Information Flow Control for Dynamic Reactive Systems

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

McKenna McCall

B.S., Computer Science, Kansas State University
B.S., Mathematics, Kansas State University
M.S., Electrical and Computer Engineering, Carnegie Mellon University

Carnegie Mellon University
Pittsburgh, PA

May, 2023

© McKenna McCall, 2023
All rights reserved.

For my daughter, Joan, who has made this journey immeasurably more fulfilling and fun.

Thesis Committee

Limin Jia, Carnegie Mellon University, Chair

Lujo Bauer, Carnegie Mellon University

Matt Fredrikson, Carnegie Mellon University

Stephen Chong, Harvard University

Acknowledgements

I feel very lucky to have many sources of mentorship and support during my PhD, but my advisor, Limin has been central to it all. Words cannot express how grateful I am for the support I have received for all these years. No matter what challenge I have faced (professional, personal, or pandemic-related), I have felt like I have someone I can go to. You are truly an inspiring mentor and I hope that I can use my career to offer guidance to others the way you have done for me.

Thank you to the rest of my committee for your feedback and support, as well as the many other incredible mentors who have helped me get here: Randy Wewer for an early introduction to propositional logic, and Mary Lathrop for encouraging me to be more ambitious in my academics. I also want to thank Dr Jessy Changstrom, who is the first person to suggest I try computer science, and Dr Jeremy LeCrone, whose classes both helped me learn how to write proofs and realize how much I enjoy writing them.

I am also incredibly thankful for my friends. Thank you to Sabrina Kunkel for introducing me to my favorite YouTube video essayist (among many other excellent recommendations) and for being an amazing listener in general. To my oldest friend, Anna Confer, you probably do not realize how much your friendship helped me get to where I am now. And an enormous thanks to my brothers and friends, Noah and Josh, for always having the best video game and streaming recommendations on-hand.

A very special thank you to Claire Le Goues and Adam Brady for being the parents of my daughter's first best friend and some of the most generous and welcoming people I have had the pleasure of knowing. From hosting our little pandemic preschool to inviting us to (several!) family gatherings, you not only helped my family weather the pandemic, but you also taught this first-generation college student more about what an academic career might look like than any workshop ever could. I am so grateful for your friendship and looking forward to many future playdates.

None of this would have been possible without the support from my family. To my husband, Mike, thank you for moving across the country and joining me on this adventure. Thank you for being the first person to encourage any and all of my ambitions, making me coffee, having the best taste in movies, and helping make everything work. Joan, thank you for sharing your delightful sense of humor and rainbow drawings with me. I am so proud and thankful to call myself your mother.

Finally, I am thankful for the funding that has afforded me all the incredible opportunities I have had over the last several years. The work in this thesis was supported in part by the CyLab Presidential Fellowship at Carnegie Mellon University and by the National Science Foundation via grants CNS1704542 and CNS1320470.

Abstract

It is common for reactive systems like web services to collect personal information and/or perform sensitive tasks, making information flow control (IFC) in these applications particularly important. Most existing work on IFC in reactive systems does not address the unique capabilities an attacker has in a dynamic setting (like using a script to simulate a user event, or creating a new HTML element), or else enforce strict noninterference which is too restrictive to be practical. Moreover, standard security definitions do not always translate cleanly to reactive settings. In this thesis, we revisit information flow control concepts like confidentiality, integrity, declassification, and endorsement from the perspective of a dynamic reactive system.

We identify new ways dynamic features can leak information via declassification and propose two strategies for mitigating these risks. The first is an extension of Secure Multi-execution (SME) that treats dynamic features specially so that they do not influence declassification. The second combines SME and taint tracking to keep track of attacker influence within SME executions. We develop a new notion of "attacker influence" which has all the advantages of a knowledge-based definition, making it an intuitive and precise way to reason about security. Robust declassification follows naturally from this new security condition because we treat declassifications as trusted behaviors in our noninterference definition. Finally, we balance the tradeoff between security and performance by developing a flexible framework which allows the seamless composition of multi-execution and taint tracking techniques. This means that event handlers from different sources can be treated differently from each other, for example, according to their relative levels of trustworthiness or complexity. We find that composition can not only balance the security and performance tradeoffs of different techniques, but some compositions actually achieve stronger security guarantees compared to using one technique alone.

Contents

1 Introduction **1**

2 Background and Related Work **3**

2.1 Reactive Systems 3

2.2 IFC Policies and Noninterference 4

2.3 Dynamic IFC Enforcement 6

2.4 Declassification 8

2.5 (Knowledge-based) Security Definitions 9

2.6 Robust Declassification and Nonmalleable IFC 11

3 Robust Declassification via Special Treatment for Dynamic Features **13**

3.1 Overview 13

3.2 Dynamic Features Leak Information 14

3.3 Dynamic Reactive Programs 17

3.4 Extended SME for Dynamic Features 24

3.5 Discussion 31

3.6 Summary 32

4 Robust Declassification via Limiting Attacker Influence **33**

4.1 Overview 33

4.2 Motivating Examples 34

4.3 SME with Dynamic Features 38

4.4 Declassification 44

4.5 Endorsement 51

4.6 Security 53

4.7 Discussion 67

4.8 Summary . . . 68

5 Compositional IFC for Reactive Systems 69
5.1 Overview . . . 69
5.2 Motivating Example . . . 70
5.3 Compositional Enforcement Framework . . . 73
5.4 Security and Weak Secrecy . . . 84
5.5 Discussion . . . 92
5.6 Summary . . . 92

6 Discussion and Future Work 93
6.1 Alternatives to IFC . . . 93
6.2 More Realistic Web Models . . . 93
6.3 Applications to Other Reactive Settings . . . 95
6.4 Extending our Compositional Framework . . . 95

A Supporting Materials for Chapter 3 97
A.1 Additional Definitions . . . 97
A.2 Soundness Proofs . . . 101
A.3 Robust Declassification Proofs . . . 106
A.4 Precision Proofs . . . 116

B Supporting Materials for Chapter 4 122
B.1 Additional Definitions . . . 122
B.2 Proofs . . . 131

C Supporting Materials for Chapter 5 150
C.1 Additional Definitions . . . 150
C.2 Complete Semantics . . . 152
C.3 Security Definitions . . . 175
C.4 Proofs . . . 188

Bibliography 235

List of Figures

2.1 Example reactive system. 3

2.2 Comparison of SME and Faceted Execution. Inputs and outputs are shown with solid arrows while the dashed arrow in the Faceted Execution diagram represents a split/join in the execution context. 7

3.1 The high execution receives the real keypress, so generates only one button with id id_i. The low execution receives the default value, so generates all n buttons. 17

3.2 Operational Semantics of Commands 19

3.3 Operational Semantics for Event Loop 20

3.4 Constructs for Defining Security Policies 21

3.5 The high execution receives all inputs, unchanged. The low execution receives L inputs and released H inputs through the release module. Note that the release module throws out the H_Δ inputs so that they do not interfere with the declassification policy. 25

3.6 Operational Semantic Rules for Single Execution 25

3.7 SME Input Rules 26

3.8 SME Output Rules 26

3.9 Projection of Traces 27

4.1 Example of dynamic features causing leaks. The dv case guarantees that the attacker copy will have a matching button (colored light blue) to capture the declassified event and leak the secret. 34

4.2 Information is allowed to flow in the direction of the arrows. The attacker can influence *U*ntrusted executions to add page elements or event handlers to try to manipulate declassification directly within an execution (blue case) or indirectly between executions (orange case). 36

4.3 Syntax for processing inputs and outputs. 38

4.4 Top-level SME rules for processing inputs and outputs, and looking up event handlers 40

4.5 Mid-level rules for processing the event queue 42

4.6 Rules for running event handers . . . 43
4.7 Additional rules for running event handers . . . 44
4.8 Updated input rule for declassification. We highlight the noteworthy changes to the existing input rule using red text. . . . 45
4.9 SME model for enforcing confidentiality and integrity information flow policies, including declassification. A secret and trusted event (mouse click) would be shared with (S, T) and (S, U) executions, and would only be shared with (P, U) and (P, T) executions if permitted by the declassification policy. . . . 45
4.10 Modified input rules for robust declassification. Noteworthy changes are shown in red text. . . . 48
4.11 Additional rule changes for robust declassification. Noteworthy changes are shown in red. . . . 49
4.12 Insecure example from Section 4.2 with robustness checks. The labels tell us the trustworthiness of the source of the page elements and event handlers, depicted here as small white labels on each page element. . . . 50
4.13 Update input rule for endorsement. We highlight the noteworthy changes to the existing input rule using red text. . . . 52
4.14 Modified input rules for transparent endorsement. Noteworthy changes are shown in red. . . . 54
4.15 Additional rule changes for transparent endorsement. Noteworthy changes are shown in red text. . . . 55
4.16 The observation ($p = c$) or behavior ($p = i$) of a trace at l (only declassification shown) . . . 56
4.17 The states above and below the dotted line are behaviorally equivalent at T even though there are different products in the (P, U) and (S, U) states. . . . 61
4.18 New rule for the behavior of a trace for robust declassification . . . 62
4.19 Additional rules for the observation ($p = c$) or behavior ($p = i$) of a trace at l to account for endorsement, downgrading, and the creation of a new page element. . . . 64
4.20 Helper functions for trace observation and behavior for endorsement and downgrading. . . . 65

5.1 Syntax for the compositional framework . . . 73
5.2 Semantics for processing inputs (user events). . . . 74
5.3 Semantics for performing outputs (communications on channels). . . . 74
5.4 Reactive system described in Chapter 2.1. . . . 76
5.5 Semantics for managing the event handler queue. . . . 78
5.6 Selected command semantics . . . 79
5.7 Storage syntax . . . 79

5.8 Event handler storage syntax for the DOM . . . 80

5.9 Example configuration . . . 83

5.10 Rules for projecting execution traces to L . . . 85

5.11 Comparison of Progress-insensitive security (top) and Weak Security (bottom) proofs. Given $T_1 \approx_L T_2$, where T_1 takes a step to ○, we want to show that T_2 can take equivalent steps ◌, and that trace equivalence maintains state equivalence ●. . . . 89

A.1 Projection of actions . . . 97

A.2 Projection of traces . . . 97

A.3 Declassified input trace . . . 99

B.1 Rules for the observation of an SME store, single store, and event handler map at l (when $p = c$) and behavior of an SME store at l (when $p = i$). . . . 123

B.2 Rules for the observation (when $p = c$) or behavior (when $p = i$) of a configuration stack at l . 123

B.3 Trace projection for output and dynamic behaviors . . . 125

B.4 Trace projection for inputs . . . 126

B.5 Helper functions for trace projection for input rules . . . 127

B.6 Helper functions for trace projection for input rules . . . 127

C.1 Event handler lookup semantics . . . 153

C.2 Event handler lookup helper functions . . . 154

C.3 Additional rules for processing the event handler queue for SME and MF . . . 155

C.4 Event handler semantics . . . 156

C.5 Additional event handler semantics for MF . . . 156

C.6 Variable assignment semantics . . . 158

C.7 Expression semantics . . . 159

C.8 Variable lookup rules . . . 161

C.9 Shared Unstructured EH storage command semantics . . . 162

C.10 Shared Tree-structured EH storage command semantics . . . 162

C.11 Rules for looking up a node in the unstructured EH storage . . . 163

C.12 Rules for updating the attribute of a node in the unstructured EH storage . . . 164

C.13 Rules for triggering an event in the unstructured EH storage . . . 165

C.14 Rules for creating a new node in the unstructured EH storage . . . 166

C.15 Rules for registering a new event handler in the unstructured EH storage . . . 167

C.16 Rules for looking up the address of a node in the tree-structured EH storage . . . 167
C.17 Rules for looking up a node in the tree-structured EH storage . . . 168
C.18 Rules for updating the attribute of a node in the tree-structured EH storage . . . 168
C.19 Rules for triggering the event handler in the tree-structured EH storage . . . 169
C.20 Rules for adding a child to a node in the tree-structured EH storage . . . 170
C.21 Rules for adding a sibling to a node in the tree-structured EH storage . . . 171
C.22 Rules for registering an event handler to a node in the tree-structured EH storage . . . 171
C.23 Rules for navigating the tree-structured SMS EH storage . . . 173
C.24 Rules for accessing the attribute of node in the tree-structured EH storage . . . 173
C.25 Rules for returning the number children a node in the tree-structured EH storage has . . . 174
C.26 Modified EH semantics for weak secrecy . . . 176
C.27 Updated event handler API semantics for weak secrecy . . . 177
C.28 Rules for configuration stack and event queue equivalence . . . 177
C.29 Low projection of a local variable store. The rules for the global variable store are the same. . . 178
C.30 Event handler store projection for unstructured event handler stores . . . 179
C.31 Value projection for unstructured EH stores . . . 180
C.32 Value projection for tree-structured EH stores . . . 180
C.33 Node projection for an unstructured EH store . . . 182
C.34 Node projection for a tree-structured EH store . . . 183
C.35 Event handler map projection . . . 183
C.36 Command and EH queue projection rules . . . 184
C.37 Projection of Traces to L Observation . . . 185

List of Tables

4.1 Knowledge definitions. Knowledge and progress knowledge are for defining a knowledge-based progress-insensitive noninterference. Release knowledge and transparent knowledge account for what is leaked to the attacker through declassification and the addition of a page element/event handler capable of transparent endorsement (respectively). Complete definitions may be found in Appendix B.1. 55

4.2 Influence definitions. Influence and progress influence are for defining an influence-based progress-insensitive noninterference. Robust influence is for defining robust declassification. Complete definitions may be found in Appendix B.1. 60

4.3 Additional knowledge and influence conditions for defining transparent endorsement. Complete definitions may be found in Appendix B.1. 66

5.1 Conversion between standard, tainted, and faceted values. 82

5.2 Requirements for Progress-Insensitive Security and Weak Secrecy. The requirements for both are similar, except that Weak Secrecy does not use requirements T3 or EH3 and the Progress-Insensitive Security requirements E1, V1, and EH1 use strong equivalence while the Weak Secrecy requirements WE1, WV1, and WEH1 use standard equivalence. 88

Chapter 1

Introduction

Online services for banking, social media, email, and shopping are becoming unavoidable, and these services typically require access to the user's personal information such as their phone number, location, or credit card details. These applications often include code from heterogeneous and untrusted sources and could potentially leak the users' sensitive data to an adversary. Attackers have been known to steal sensitive user data [51], sometimes via third-party scripts, which have been observed indiscriminately collecting data from web forms, including personal information like email addresses and passwords [86].

Information flow control (IFC) is a promising technique to ensure that applications do not leak sensitive data. Information flow policies (such as, "clicks are secret and should not be leaked to public advertisers") specify which information flows are allowed. Information flow policies may be enforced statically [78, 37, 94, 50] (at compile-time), dynamically [13, 14, 15, 38, 79] (at runtime), or via hybrid techniques [56, 75, 40, 65, 8] combining both. The canonical IFC security property is *noninterference* [43] which says that secret inputs should not influence more public outputs. The simplest form of noninterference says that public outputs (which can be used with the least restriction) should never be influenced by secret inputs (which is the data whose use is most restricted).

Many runtime mechanisms have been developed for enforcing information flow control (IFC) policies [35, 22, 88, 24, 41, 52, 20, 21]. Some prior work enforces strict noninterference [5, 73, 35, 20, 42, 45, 82], which is often too inflexible to be practical. Supporting principled *declassification*, which allows sensitive information to be leaked while maintaining an otherwise provably secure system, is important for many useful web services like website analytics. For instance, if a company wants to know which of their products are being clicked (but not purchased), they may want to track some of their customers' interactions on their site by using third-party analytics scripts. Declassification can ensure that these third parties will have access to the information they need (e.g., which products are clicked), without releasing everything.

Prior work that allows declassification by web scripts (typically modeled as a reactive system [28]) either did not prove formal properties about declassification [22, 24], or used a simplified model that is missing some dynamic JavaScript features that could be leveraged by an attacker to leak information [91, 12]. Online applications often include code from several (potentially untrusted) sources and update content dynamically. By not modeling dynamic features, researchers may miss ways that an attacker might leverage dynamic content to leak sensitive information [38]. This is especially problematic in the presence of declassification which may interact with dynamic features to leak more than intended [66, 101, 31].

Most IFC approaches from prior work use the same enforcement mechanism for all components in an application. Broadly, these can be classified as *multi-execution approaches* [38, 15, 17], or *taint tracking approaches* [13, 14, 95, 84]. *Multi-execution approaches* run code multiple times and ensure that the code executing at a particular level only outputs data at the same security level, replacing sensitive data from higher security levels with "default" values. *Taint tracking approaches* annotate data with *labels* to indicate its security level and can suppress outgoing sensitive data to publicly observable channels or abort the execution altogether to prevent leaks. These approaches differ in performance, how much they alter the semantics of safe programs, and the relative strength of their security guarantees. Given the heterogeneity of applications, a compositional enforcement mechanism where different components execute under different IFC enforcement mechanisms could offer an attractive solution to the tradeoffs of each approach.

In this thesis, we address these gaps by presenting two techniques for securely incorporating dynamic features into reactive systems. In Chapter 3, we present a monitor that treats dynamic features specially so that they cannot influence declassification. Next in Chapter 4, we will look at how traditional IFC techniques can be used to track attacker influence to ensure the dynamic features are not used by the attacker to leak more information than intended. Finally, we develop a compositional framework in Chapter 5 where different event handlers may be protected by different IFC techniques, and likewise, the resources shared between event handlers may be protected by different IFC techniques than the event handlers themselves. This is especially useful for scripts on a webpage, which may come from different sources and benefit from being treated differently. We also incorporate the techniques from Chapter 3 into the framework to handle declassification.

Thesis statement: *IFC monitors can enforce information flow policies in reactive systems. Monitors can prevent attackers from leveraging dynamic behaviors in these systems to leak information via declassification by never allowing events from dynamic features to be declassified or by restricting attacker influence. Composition can balance the tradeoffs of two approaches to IFC, multi-execution and taint tracking, improving the security guarantees of taint tracking and the run-time overhead of multi-execution.*

Chapter 2

Background and Related Work

This chapter describes relevant background and prior work. We review work on reactive systems in Section 2.1, IFC security properties in Section 2.2, and dynamic IFC techniques in Section 2.3. Then, in Section 2.4 we describe the type of declassification used in this thesis, background on knowledge-based security conditions in Section 2.5, and, finally, robust declassification and nonmalleable IFC in Section 2.6.

2.1 Reactive Systems

Reactive systems have been used to model event-driven programs [28, 20, 52, 68], such as scripts on webpages. In the reactive model, a program is a set of event handlers which are executed when a corresponding event is triggered.

We describe a simple reactive system modeling a webpage as shown in Figure 2.1. First, a user input triggers an event (step 1 in the Figure) which causes corresponding event handlers (EH) to run. Event

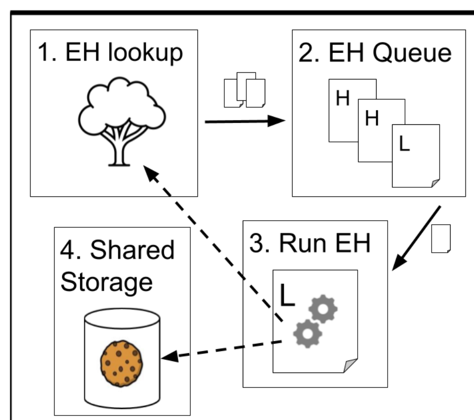


Figure 2.1: Example reactive system.

handlers may be stored in a tree (such as the DOM on a webpage) or a simpler data structure, like an unordered list. Event handlers wait in a queue (step 2) to be run. The runtime manages a single-threaded event loop to run all the event handlers in the queue. The runtime also keeps track of the system state which is *producer* (while an event handler is running) or *consumer* (when an event handler finishes). While an event handler is running (3), it may trigger new events, register new event handlers, or generate page elements by interacting with the event handler storage. These dynamic features (event handlers triggering other event handlers, registering event handlers, or creating page elements) are an important part of our reactive model. An event handler may also interact with other types of storage (like cookies or bookmarks) that persist after the event handler finishes (4). A new event is processed when all event handlers have finished running.

This model is an abstraction of the single-threaded main event loop from the JavaScript engine in browsers. Such reactive programs have been used to model the way that browsers and IFC mechanisms interact with scripts [74, 25], but prior work typically does not involve security-relevant dynamic features like dynamically-generated page elements. For the rest of this thesis, we use an extended version of this model to explore how IFC policies can be enforced even in the presence of dynamic features. We also use this model to explore how the tradeoffs of different IFC techniques might be balanced by composition.

2.2 IFC Policies and Noninterference

IFC policies specify which information flows are permitted in an application and may be about data confidentiality (which protects secret information from unauthorized access) or data integrity (which protects trusted data from unauthorized modification). Typically, this involves assigning information flow security labels to components of the application (which may include memory locations, data, events, or principals) and arranging the labels into a lattice [36] which determines the allowed information flows. There has also been work on policy inference (e.g., [92]) which requires fewer annotations than traditional policies. The simplest 2-point security lattice has the labels L (for Low confidentiality or public data) and H (for High confidentiality or private data) and partial order $L \sqsubseteq H$. Here $\sqsubseteq$ can be interpreted as "flows-to", so $L \sqsubseteq H$ means that L data is allowed to flow to H, but not the other way around.

The canonical IFC security property is *noninterference* [43], which, in this setting, says that inputs at some security level l should only influence outputs at security level l' if the input is more public than the output: $l \sqsubseteq l'$. For the simple 2-point security lattice, noninterference says L inputs may influence L or H outputs, but H inputs may only influence H outputs. If a program were to output an H input to an L channel, it would not satisfy noninterference. These kinds of information flows are called *explicit* leaks.

It is also possible to leak information *implicitly*. For instance, the following program branches on a secret, which can leak information based on the observable differences between the branches:

$\mathsf{if}\ h == 42\ \mathsf{then}\ l := k$

It is useful to define other variants of noninterference to compare IFC techniques which do (or do not) protect against various leaks. Approaches which only prevent explicit leaks and permit implicit leaks like the one above satisfy *weak* or *explicit* secrecy. Volpano [95] originally defined weak secrecy to formalize data-dependent flows as opposed to the stronger property of noninterference. Schoepe et al. [84] generalize this property as a knowledge-based property, *explicit secrecy*, to adapt to different semantics used by different languages.

The while loop in the following program may cause the system to diverge, another source of leakage:

$\mathsf{while}(h)\ \{\ ...\ \}; \mathsf{output}_L(l)$

If the loop condition h is itself a secret, whether the system eventually sends the output leaks something about h. If the output is received, h must have eventually evaluated to false. Otherwise, h must have been true. The example above satisfies *progress-insensitive* noninterference, which allows secret input events to determine whether the system makes progress and eventually produce another public output. *Progress-sensitive [9, 64, 49]* noninterference, meanwhile, does not permit such leaks. Even if these programs do not cause the system to diverge, using a secret value as a loop condition may mean that the loop executes a different number of times, leading to different execution times, depending on the secret. Like progress leaks, there are variants of noninterference allowing (*timing-insensitive*) or disallowing (*timing-sensitive* [1, 78]) such leaks. In Chapter 5, we compose multiple IFC techniques and define two security conditions to compare the relative security guarantees of different compositions: one based on progress-insensitive noninterference (the stronger guarantee), and one based on weak secrecy (the weaker guarantee).

Composition of information flow properties has been studied in the setting of event-based systems [58, 61]. McCullough, further, defined the property of restrictiveness for security of systems [63] based on what a user can infer about sensitive data, which is composable. Zakinthinos and Lee [98] showed important results about the composition of generalized noninterference, which was earlier proven to be not fully compositional [62]. Rafnsson and Sabelfeld [71] explore the composition of PINI and progress-sensitive noninterference in the context of interactive programs. Similar to existing work, we explore the composition of information flow security properties across various types of mechanisms for event handlers and shared storage. Because event handling and accesses to shared storage are not symmetric, we stipulate requirements on each component and how components interact but cannot directly compose them as homogeneously defined secure components.

2.3 Dynamic IFC Enforcement

Much work has been done on information flow control enforcement in JavaScript [48, 46, 47, 53, 34, 15, 39, 51]. Because of the dynamic nature of JavaScript, all these approaches use runtime enforcement mechanisms to enforce information flow control. Several projects have developed tools for enforcing information flow control on web scripts by modifying browser components [22, 88, 23, 35, 91, 24, 15, 33]. Methods used by these projects include taint tracking, compartmentalization, and secure multi-execution. In this section, we describe specific IFC enforcement techniques which we broadly categorize into multi-execution techniques and taint tracking techniques.

We illustrate different enforcement mechanisms developed to enforce noninterference for reactive systems via an example event handler:

$\mathsf{onKeyPress}(k)\ \{\ \mathsf{if}\ k == 42\ \mathsf{then}\ l := k\ \}$

For this example, we use a two-point security lattice with labels Low and High and partial order $L \sqsubseteq H$, meaning that information may flow from Low to High, but not from High to Low. Assume that initially $l = 0$. The event handler runs for keypress events and the identity of the keypress (*which* key was pressed) is passed as a parameter. The occurrence of the keypress events is public (L), while the parameter k, is secret (H). This means an attacker is allowed to learn that a key was pressed but not *which* key.

Multi-execution Secure multi-execution was introduced as an IFC mechanism for JavaScript on web pages [38, 35], enforcing noninterference by running multiple copies of the event handlers at each security level the event is visible at. Each performs only the outputs at matching security levels, skipping the rest. Inputs which are not visible at the given security level are replaced with default values. When a key is pressed in the example above, the execution at the H level reads the value of the key press k (the H input) and assigns 42 to the H copy of l if $k \mapsto 42$. In the L execution, the H input k is replaced with the default value, say 0, so the branch is never taken. Thus, at the end of the L execution, l remains unchanged in the L store, irrespective of the value of k.

Faceted execution [15] is a similar multi-execution technique, whose relationship to SME has been studied [26]. This technique simulates the multiple executions of SME while avoiding unnecessary redundancy by creating *facets* of a value only when the value contains a secret, e.g., $v = \langle v_h | v_l \rangle$ where v_h is the value of v observable at H and v_l is the value of v observable at L[1]. The execution splits only when necessary, such as when branching on a faceted value. Like SME, the L outputs are suppressed in the H copy and H outputs are suppressed in the L copy. After evaluating the branch, the split executions join

[1]The original faceted values [15] have the conditional format $\langle k\ ?\ v : v' \rangle$, where those that can observe principal k's private data see v and others observe v'.

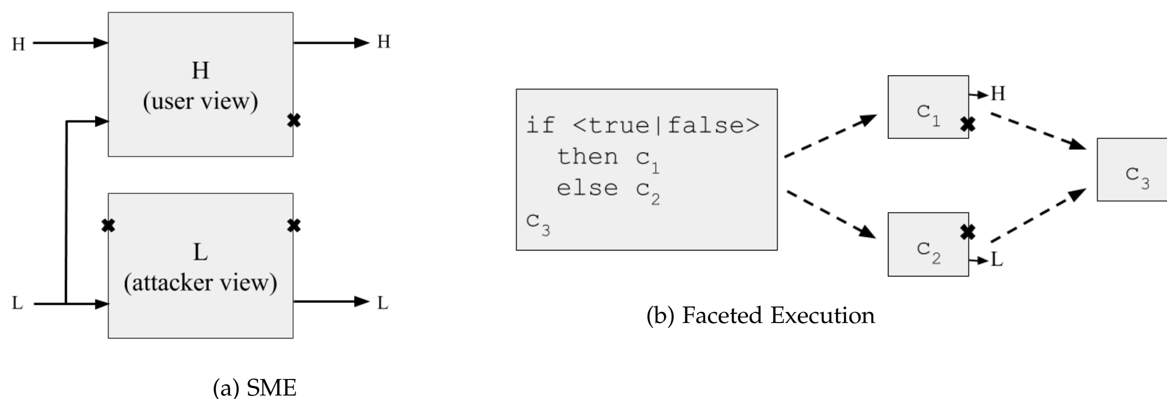


Figure 2.2: Comparison of SME and Faceted Execution. Inputs and outputs are shown with solid arrows while the dashed arrow in the Faceted Execution diagram represents a split/join in the execution context.

and continue normally as one. Multi-execution techniques typically satisfy strong security guarantees. See Figure 2.2 for an illustration of the two techniques.

In the above example, $l \mapsto 0$ initially and is observable at all levels. If the secret input k is 42, l is assigned the faceted value $l = \langle 42|0 \rangle$, meaning H observers see 42, while L observers see l's original value. If the event handler then output l, the label on the channel receiving the output would determine which value to send: 42 if the channel is H, 0 if the channel is L.

SME has been extended to be more precise to ensure SME does not alter the order of the outputs compared to the original program [99]. Later work on faceted execution discusses its extension to applications where the policy is specified separately from the code [17, 97], and how to deal with exceptions [16]. Later work combines SME and faceted execution [83] (and an optimization [4]) and proposes "generalized" multiple facets [69] to balance the security and performance tradeoffs of the two multi-execution techniques (in the first two cases), consider a more general security lattice (in the last case), and each achieves stronger (termination-sensitive) security guarantees than offered by traditional faceted execution. Ngo et al. [70] show that a perfectly sound and precise monitor satisfying termination-insensitive noninterference cannot exist and argue that multi-execution techniques actually satisfy *indirect* termination-sensitive noninterference. Later, Algehed and Flanagan [3] proved the impossibility of building a transparent and efficient black-box runtime monitor using SME.

Taint tracking Taint tracking approaches carry and propagate taint via security labels along with data. Taint tracking techniques are prone to *implicit leaks*. In the example, suppose $l \mapsto 0^L$ initially, where L is the label of l. If $k \mapsto 42^H$, then $l \mapsto 42^H$ at the end. Otherwise, the branch is not taken, and l remains 0^L. Since the value of l depends on the branch condition, the branch condition is leaked *implicitly* through l.

Since the behavior within a branch depends on the branch condition, the branch condition is leaked *implicitly* through behaviors within the branches. To secure taint tracking against implicit leaks, some approaches maintain a program context (*pc*) which keeps track of the context of the control flow decisions. These approaches abort the execution when assigning to public variables in secret contexts [13]. These approaches based on *no-sensitive-upgrade* (NSU) (or later permissive upgrade [14]) satisfy termination-insensitive noninterference. NSU semantics can be conservative and risk aborting the execution even when nothing is leaked. Some other approaches choose not to use NSU semantics, having even weaker security guarantees in favor of avoiding unexpected interruptions to the execution. For this reason, and to have different security properties to use when comparing compositions in Chapter 5, this thesis focuses on techniques which do not abort or diverge [95, 84].

2.4 Declassification

Many of the monitors described above enforce a strict *noninterference*, where secret inputs are never allowed to influence public outputs. But this is often too restrictive to be practical. For instance, an online shop might want to do analytics to learn which products users are clicking on most, or a bank might want to know a user's location to help authenticate them. Declassification [81] offers a principled way to release some information while maintaining an otherwise provably secure system.

Several IFC approaches for browsers have been proposed that build on existing IFC techniques to allow information to be declassified. Some work uses declassification labels [22] or privileges [88] to specify exceptions to information flow policies. WebPol policies [24] allow the host of a website to specify which page elements and user-generated events can be declassified to which domains. Mash-IF [57] allows user-specified declassification and proposes a technique for analyzing scripts to generate declassification rules (if the user consents to the information flow).

To allow scripts that depend on approximated or aggregated secret values (e.g., analytical scripts) to run correctly in SME, Vanhoef et al. proposed stateful declassification policies [91]. In their system, a projection function specifies what information from a secret event can be declassified. In addition, a stateful release function maintains the aggregate information about all secret events seen so far for eventual declassification (e.g., total number of clicks). Example stateful policies include whether the user pressed a specific shortcut key can be released, the average of the coordinates of mouse clicks can be released, and after the user clicks on the "AGREE" button, the GPS reading can be released.

Other work has identified techniques for declassification in SME. One model treats declassifications as non-blocking outputs from a secret execution to a more-public execution [72]. Declassification in SME

typically assumes that the program can handle both the real and declassified inputs, but these assumptions may not always be realistic since a program may not be aware of the context its running in (or that it is running in SME at all). Asymmetric SME [29] addresses this by running different copies of the program at each security level, ensuring that the copy that receives declassified values can adapt to these inputs.

In this thesis, we use stateful declassification based on Vanhoef et al. [91] and assume that multi-execution techniques run the same code.

2.5 (Knowledge-based) Security Definitions

An intuitive way to reason about information flows is to focus on the knowledge that an attacker gains by observing public data. In the reactive setting, an attacker can observe public inputs (i.e., events) and public outputs (i.e., behaviors generated by event handlers). The *attacker's knowledge* is the set of all possible inputs that could have produced the public outputs they observed. As the attacker makes more observations, they learn more information which makes them more confident about the possible secret inputs. Knowledge-based security conditions have been used in a variety of settings. Balliu [18] highlight the usefulness of a knowledge-based condition in interactive/reactive and nondeterministic settings (like the one we consider here) and explore the relationship between trace-based and knowledge-based security conditions. Other work defines a knowledge-based security condition for dynamic policies [7, 2] (our *progress knowledge* is based on some of this work [7]). There are also knowledge-based explicit secrecy security conditions [84].

For explanatory purposes, we write T to denote an execution trace and τ to denote input/output sequences. The secrets in reactive systems are sequences of user inputs. Let us write $T \approx_L T'$ to denote that two traces are observationally equivalent at the level L. Informally, an attacker's knowledge, written $\mathcal{K}(T, \sigma_0)$, is the set of possible input sequences that could produce an output trace that is observationally equivalent at L to T from the initial configuration σ_0.

We define $\mathsf{in}(T)$ and $\mathsf{out}(T)$ to be the input and output actions in T, respectively. We denote $\mathsf{runs}(\sigma_0)$ as the set of execution traces starting from the initial state σ_0.

$$\mathcal{K}(T, \sigma_0) = \{\tau_i \mid \exists T' \in \mathsf{runs}(\sigma_0), T \approx_L T', \tau_i = \mathsf{in}(T')\}$$

The security property that we are interested in enforcing says that interacting with the system does not reveal anything about the user's secret inputs to the attacker. It is defined as follows:

Definition 1 (Security)**.** *We say a configuration σ_0 is secure against attackers at level L, if for all traces τ, action α, s.t. $\tau \cdot \alpha \in \mathsf{runs}(\sigma_0)$, $\mathcal{K}(\tau, \sigma_0) \subseteq_{\preceq} \mathcal{K}(\tau \cdot \alpha, \sigma_0)$.*

Here, $S_1 \subseteq_{\preceq} S_2$ means that every element in S_1 is a prefix of an element in S_2; traces in S_2 may be longer because α might be an input event. This property is weaker than standard noninterference, which requires that a low observer know nothing about the high inputs and that the knowledge set includes all possible secret user inputs. However, this is too restrictive, as our program is not input-total: events must be associated with existing elements, reducing the number of possible inputs.

We illustrate Definition 1 through an example. Consider a webpage with an image (*img*) containing the numbers "1" through "8". Whether the user clicks the image ($img.\mathsf{click}(v)$) is public but where the user clicks on the image (i.e., the value of v) is secret. Assume the attacker knows the program and that the image displays the numbers 1 through 8. This means the attacker knows that $img.\mathsf{click}(9)$ is not a possible input. We allow the attacker to know this type of information, even though it means they have learned something about the secret. After the user clicks 2 on the image, the attacker's knowledge is that the user might have clicked on any number between 1 and 8:

$$\mathcal{K}([img.\mathsf{click}(2)], \sigma_0, \mathcal{P}) = \{[img.\mathsf{click}(1)], ..., [img.\mathsf{click}(8)]\}$$

If the program later output v (which contains the number the user clicked) in response to a different event ($id.\mathsf{Ev}()$), then the attacker's knowledge after receiving the output would be:

$$\mathcal{K}([img.\mathsf{click}(2), id.\mathsf{Ev}()], \sigma_0, \mathcal{P}) = \{[img.\mathsf{click}(2), id.\mathsf{Ev}()]\}$$

Then, some of the shorter traces in the older knowledge set (before the output is received) are not prefixes of traces in the new set (after the output is received):

$$\{[img.\mathsf{click}(1)], \cdots, [img.\mathsf{click}(8)]\} \not\subseteq_{\preceq} \{[img.\mathsf{click}(2), id.\mathsf{Ev}()]\}$$

The program is not secure using our definition.

Knowledge-based security with declassification For a program to satisfy noninterference, the attacker should not be able to refine their knowledge about the secret inputs throughout the program's execution, but this definition does not allow for controlled leaks of private information through declassification. Instead, we use a *gradual release* property [11] which ensures that the attacker's knowledge is not refined, outside of what is permitted (i.e., what is released) by the declassification policy.

The gradual release property has been applied to systems that allow flexible declassification. For instance, Banerjee et al. proposed expressive declassification policies defined by agreements of initial state written as flowspecs, which specify precisely how much information may be revealed about confidential variables [19]. They also present a type system for enforcing knowledge-based security, which is defined

as a conditional gradual release property. Askarov and Chong [7] also present a definition of knowledge which reasons about initial configurations. Like us, they refine it to *progress knowledge* which restricts the set of configurations to those that can produce another observable event.

A weaker form of noninterference that allows implicit leaks is *weak secrecy* [95] or *explicit secrecy* [84]. Weak secrecy only allows information leaks through branch predicates. Consider the program:

$$\mathsf{if}\ h = 0\ \mathsf{then}\ l := 1\ \mathsf{else}\ l := 0; \mathsf{output}(L, l)$$

The program above satisfies weak secrecy, as it can be re-written as a secure program without "high" branches: $l := 1; output(L, l)$ and $l := 0; output(L, l)$. Because both programs are secure, the original program satisfies weak secrecy.

Other work [10] proposes a notion of integrity that is the dual to the knowledge-based condition for confidentiality called "attacker impact" and explores the relationship between integrity and confidentiality as "attacker control". While attacker knowledge is the set of inputs leading to the same public observations, attacker impact is the set of attacks (which are blocks of attacker-controlled code) leading to the same trusted events. If an attacker does not have influence over trusted events, their impact is not refined throughout execution; that is, each attack is equally powerless. Meanwhile, attacker control is the set of attacks with a similar influence on knowledge. Then a system satisfies robust declassification if the attacker's release control (i.e., the set of possible attacks resulting in *any* declassification) is smaller than the attacker's control after a declassification (i.e., the set of possible attacks resulting in *a particular* declassification). If an attacker has influence over declassification, then different attacks would lead to different declassifications and the attacker's control would be refined when the system makes a particular declassification. In Chapter 4, we define *attacker influence*, which is similar to attacker impact and control, but rather than tying attacker influence to knowledge, we show that robust declassification follows directly from the security condition for integrity when we treat declassification as a trusted action which should not be influenced by the attacker.

2.6 Robust Declassification and Nonmalleable IFC

In general, a system which does not allow *active* attackers (i.e., ones who influence the system in some way) to leak more than *passive* attackers (i.e., ones who do not influence the system) through declassification is called *robust* [101, 32, 31]. The concept of robust declassification was introduced by Zdancewic et al. to ensure low integrity attackers cannot manipulate declassification operations [101]. Later work develops a type system for enforcing robust declassification and *qualified robustness* [66] to account for

endorsement (which is the integrity dual to declassification for confidentiality). Similar to robust declassification, transparent endorsement [31] says that a principal can only supply data for endorsement if the data is observable to them. Otherwise, they could use data without ever actually being granted access to it.

Our robust declassification condition in Chapter 4 is based on qualified robustness and we also prove a qualified version of transparent endorsement which accounts for declassification. In Chapter 3, we do not have an explicit integrity label for attackers. Instead, we assume scripts have low integrity and therefore actions performed by scripts are considered to have low integrity.

Chapter 3

Robust Declassification via Special Treatment for Dynamic Features

In this chapter, we describe a technique for robust declassification where we treat events from dynamic features (which may have been added by an attacker) specially, so they do not influence declassification. [1]

3.1 Overview

One of the challenges of IFC is *declassification*, which allows sensitive information to be intentionally released while maintaining an otherwise provably secure system. Principled declassification is particularly important online, as many useful scripts, such as web analytics services, only work when they are allowed to access some sensitive data. For example, a company may be interested in knowing where their website is most popular, so the script will need to access visitor locations. Prior work that allows declassification by web scripts either did not prove formal properties about declassification [22, 24], or used a simplified model that is missing some dynamic JavaScript features that could leak information [91].

Ignoring dynamic features of scripts—such as user action simulation, new DOM element generation, and new event handler registration—is problematic because they can be used to leak information, especially when they interfere with trusted declassification operations. For instance, consider a declassification policy that allows a user's GPS location to be sent to a server only after the user clicks on the "AGREE" button. If the IFC mechanism does not distinguish between a user-generated click and a script-simulated click, the user's GPS location will be leaked to the server without user consent, which is a violation of the policy. This is an example of a lack of *robust declassification* [66], in which an active attacker can abuse declassification components and trick the system into leaking more information than intended. Another

[1] This chapter is based on published work [60].

dynamic feature of scripts that may leak information is DOM element generation. A script may change which fields are present on a page based on a secret value. Since a user can only trigger events for elements which are present on the page, observing which events are triggered will leak information.

To reason about declassification precisely, we appeal to the concept of *gradual release* [11], which allows us to say a system is secure if the attacker's *knowledge* remains constant outside of declassification and to quantify over released information at declassification points.

We aim to provably secure sensitive user information in the browser context, while maintaining the flexibility of declassification, even in the presence of active attackers—those who can simulate user actions, generate new DOM elements, and register new event handlers. Few papers have examined this problem before. Our key insight is that script-generated events and objects need to be prevented from affecting the declassification mechanism.

This chapter makes the following contributions: We show through examples that naïvely including dynamic components to otherwise secure models introduces information leaks. We extend prior work on secure multi-execution (SME) with declassification [91] and design new SME rules that treat script-generated content specially to ensure that declassification policies cannot be manipulated by them. Instead of trace-based definitions, we use a knowledge-based progress-insensitive definition of security and prove that our enforcement mechanism is sound. This way, the properties of our system can be described by changes in an attacker's knowledge—a natural way to model what an attacker learns by observing a system. We prove that our enforcement mechanism is precise (does not alter the semantics of "good" programs) and has robust declassification.

The rest of this chapter is organized as follows. We present examples where dynamic features interfere with declassification in Section 3.2. In Section 3.3, we introduce our dynamic reactive program model and introduce declassification. Our SME system and its formal properties are presented in Section 3.4. We discuss specific aspects of our system in Section 3.5. Detailed definitions, lemmas, and proofs can be found in Appendix A.

3.2 Dynamic Features Leak Information

We illustrate potential security problems caused by interactions between dynamic features of scripts and declassification and motivate our multi-DOM model.

3.2.1 Scripts Interfering with Declassification

One of the drawbacks of the reactive programming model from the prior work discussed in Chapter 2.1 is that it is overly simplified and omits many security-relevant dynamic features. The dynamic features that we focus on are user event simulation, new DOM element generation, and new event handler registration. We chose these features because of the clear risk they pose to IFC. We leave modeling event bubbling, preemptive events, and DOM element removal to future work. Next we show how these features interfere with declassification if not treated carefully.

Script-simulated events First, in the presence of script-simulated events, the implementation of declassification policies needs to consider the provenance of events. In particular, events generated by scripts should not affect when and what information is declassified. Consider the following scenario in which the declassification policy allows the release of the average coordinates of every two clicks. A script simulates a click at a constant location l once the user clicks on the webpage. The script knows l and the average of l and the location of the user's click, from which computing the coordinates of the user's click is trivial.

Consider another declassification policy that allows the release of a GPS reading after the user clicks on a button authorizing it. Scripts can simulate a click on that button to cause the information to be released. These examples show that declassification policies should not be affected by script operations. Allowing untrusted and potentially attacker-controlled scripts to decide what is declassified violates the principle of *robust declassification* [101], which requires that an active attacker cannot learn more than a passive attacker. An active attacker not only observes the system behavior but can also modify it. The enforcement mechanism must distinguish between events triggered by the user and events triggered by scripts to ensure robust declassification.

Dynamically-generated elements Dynamically-generated elements can create channels that leak information if their creation depends on a secret. Consider the policy: button click events are visible to public scripts and keypress events are secret and not visible to public scripts. Consider the following script. For now, assume *secret* stores the code of the key that user has pressed and that $\mathsf{new}(id, t, e)$ generates a new object of type t identified by id with attributes e, and $\mathsf{addEh}(id, onClick\{c\})$ registers an event handler with body c for click events from the object identified by id.

$$\begin{aligned}
&\mathsf{case}\ \mathit{secret}\ \mathsf{of}\\
&\mid 1 \Rightarrow \mathsf{new}(id_1, \mathsf{Button}, e);\ \mathsf{addEh}(id_1, \mathit{onClick}\{c_1\})\\
&\ldots\\
&\mid n \Rightarrow \mathsf{new}(id_n, \mathsf{Button}, e);\ \mathsf{addEh}(id_n, \mathit{onClick}\{c_n\})
\end{aligned}$$

where $c_i = \mathsf{output}$ `attacker.com` i.

Here, depending on the value of *secret*, a different button will be generated with a distinct event handler. The user only sees one button, which depends on the value of *secret*; if they pressed key i, (i.e., $secret = i$), the user sees a button with the ID id_i. Once the user clicks on the button with ID id_i, the *onClick* event handler associated with that button will be triggered, sending the value i to the attacker. Thus, the attacker will receive the value of *secret*, revealing which key the user pressed.

Extending SME If we naïvely extend the stateful declassification mechanisms for SME to handle these new features, we may be too restrictive and risk altering the semantics of legitimate programs, making it less practical; or we may not be restrictive enough, making it vulnerable to exploitation by attackers. In FlowFox [35] (Firefox with SME support), all DOM APIs are labeled as low, which means that the high execution cannot add new elements to the DOM since low outputs are suppressed from the high execution. This is very restrictive, as websites frequently use JavaScript to modify parts of the page based on private user data. For example, a page may highlight a password field which is too weak on a registration page. The password field is secret, so the high execution needs to modify the DOM to highlight the field. Since this output is suppressed, the DOM will not be updated and the user will not see the field change.

To remove this restriction, we give each execution its own copy of the DOM. However, if we freely allow the high execution to add elements then the leak in the second example can still be exploited. The low execution receives a default value (denoted dv) instead of *secret*, so the attacker adds the following:

$$\begin{aligned}
\mid dv \Rightarrow\ &\mathsf{new}(id_1, \mathsf{Button}, e);\ \mathsf{addEh}(id_1, \mathit{onClick}\{c_1\});\\
&\ldots\\
&\mathsf{new}(id_n, \mathsf{Button}, e);\ \mathsf{addEh}(id_n, \mathit{onClick}\{c_n\})
\end{aligned}$$

The high execution has a copy of this script which knows the real value of *secret*. It generates a single button for the user whose ID depends on the value of *secret*, like before. But this time, the low execution executes the branch for the default value, generating n buttons, one for each possible value of *secret*. The resulting view for each execution is shown in Figure 3.1. The user never sees the buttons from the low execution and the attacker does not see which button was generated for the user, but when the

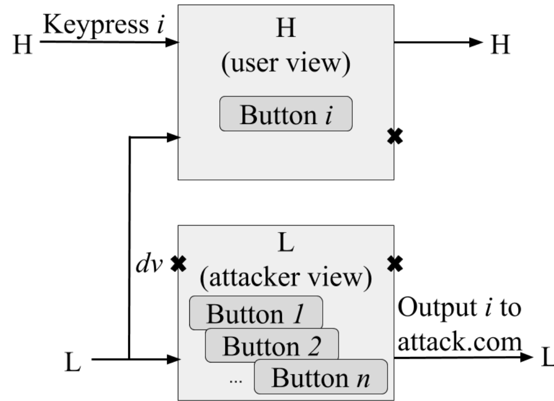


Figure 3.1: The high execution receives the real keypress, so generates only one button with id id_i. The low execution receives the default value, so generates all n buttons.

declassification policy releases the button click event, the low execution is guaranteed to have a matching button to capture the event since every possible button is present. The value of *secret* is leaked to the attacker just as before. In Section 3.4, we show how to stop leaks through dynamically generated elements. Next, we show informally that this example violates a knowledge-based security property.

3.3 Dynamic Reactive Programs

To design an IFC enforcement mechanism which prevents leaks due to dynamic features, we need to design a language model that includes those features. We first present the syntax and semantics of our dynamic reactive programs. We then introduce security relevant constructs. Finally, we explain stateful declassification and extend both the language and security definitions to accommodate declassification.

3.3.1 Syntax

The syntax of our language is shown below. We write Ev to denote events such as click and mouseover. Event handlers, denoted *eh*, always have names of the form *onEv*, where *Ev* is the name of the event. One difference between our model and prior work [28] is that we make explicit the object that events are associated with. For instance, $b1.\mathsf{click}(v)$ corresponds to the user clicking on a button with the identifier *b1*. The body of an event handler is a command c. We allow event handlers to trigger other events, generate new objects, and register event handlers. It is common for scripts to generate new DOM elements and simulate events.

$$
\begin{array}{llll}
\textit{Event:} & \mathsf{Ev} & ::= & ... \\
\textit{Event handler:} & eh & ::= & onEv(x)\{c\} \\
\textit{Command:} & c & ::= & \mathsf{skip} \mid c_1; c_2 \mid x := e \mid \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid \mathsf{while}\ e\ \mathsf{do}\ c \mid \mathsf{output}\ ch\ e \\
& & \mid & \mathsf{trigger}\ id.\mathsf{Ev}(e) \mid \mathsf{new}(id, t, e) \mid \mathsf{addEh}(id, eh) \\
\textit{Event handler map} & M & ::= & \cdot \mid M, \mathsf{Ev} \mapsto \{eh_1, \cdots, eh_k\} \\
\textit{State} & \sigma & ::= & \cdot \mid \sigma, x \mapsto v \mid id \mapsto (v, M)
\end{array}
$$

Command c includes the following actions: $\mathsf{output}\ ch\ e$ evaluates e and sends the result to URL ch, $\mathsf{trigger}\ id.\mathsf{Ev}(e)$ allows the script to simulate an event Ev with parameter e associated with an object identified by id, $\mathsf{new}(id, t, e)$ generates a new object identified as id of type t (e.g., button, form) with attributes e, and $\mathsf{addEh}(id, eh)$ registers a new event handler eh to the object id. We allow multiple event handlers to be registered per event. We write M to denote a mapping from an event to the set of registered event handlers for this event. We define the system state, σ, to be a mapping from variables to values and named objects to tuples, which model the attributes and event maps associated with the objects. For instance, a button *b1* can be associated with several mouse events, each of which could have multiple registered event handlers. Because new objects and event handlers can be added at run time, we do not have a fixed program. Instead, given a state σ, we can view all the event handlers in σ as the program of σ.

3.3.2 Operational Semantics

To define the operational semantics for our language, we first introduce a few runtime constructs.

$$
\begin{array}{llll}
\textit{Events} & E & ::= & \cdot \mid E, id.\mathsf{Ev}(v) \\
\textit{Non-silent actions} & a & ::= & id.\mathsf{Ev}(v) \mid ch(v) \\
\textit{Actions} & \alpha & ::= & a \mid \bullet \\
\textit{Execution state} & s & ::= & P \mid C \mid LC \\
\textit{Configurations} & \kappa & ::= & \sigma, c, s, E \\
\textit{Action traces} & \tau & ::= & \cdot \mid \tau\, \alpha \\
\textit{Execution traces} & t & ::= & \kappa \mid \kappa \xrightarrow{\alpha} t
\end{array}
$$

We write E to denote the set of events generated by the event handlers. As we discussed in Section 3.2, these events cannot be mixed with user input events. Therefore, we collect them in a separate context and process them once they are generated. We write a to denote input and output actions, $\bullet$ to denote silent actions, and α to denote all actions. An action trace, denoted τ is a sequence of actions. To model single-threaded execution, the runtime semantics keeps track of the execution state: producer, denoted P,

$\boxed{\sigma, c \xrightarrow{\alpha} \sigma', c', E}$

$$\frac{}{\sigma, \mathsf{skip}; c \xrightarrow{\bullet} \sigma, c, \cdot}\ \textsc{skip} \qquad \frac{\sigma, c_1 \xrightarrow{\alpha} \sigma', c_1', E}{\sigma, c_1; c_2 \xrightarrow{\alpha} \sigma', c_1'; c_2, E}\ \textsc{seq} \qquad \frac{[\![e]\!]_\sigma = v}{\sigma, x := e \xrightarrow{\bullet} \sigma[x \mapsto v], \mathsf{skip}, \cdot}\ \textsc{assign}$$

$$\frac{[\![e]\!]_\sigma = \mathsf{true}}{\sigma, \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \xrightarrow{\bullet} \sigma, c_1, \cdot}\ \textsc{if-true} \qquad \frac{[\![e]\!]_\sigma = \mathsf{false}}{\sigma, \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \xrightarrow{\bullet} \sigma, c_2, \cdot}\ \textsc{if-false}$$

$$\frac{[\![e]\!]_\sigma = \mathsf{true}}{\sigma, \mathsf{while}\ e\ \mathsf{do}\ \ c\ \xrightarrow{\bullet} \sigma, c; \mathsf{while}\ e\ \mathsf{do}\ \ c\ , \cdot}\ \textsc{while-true} \qquad \frac{[\![e]\!]_\sigma = \mathsf{false}}{\sigma, \mathsf{while}\ e\ \mathsf{do}\ \ c\ \xrightarrow{\bullet} \sigma, \mathsf{skip}, \cdot}\ \textsc{while-false}$$

$$\frac{[\![e]\!]_\sigma = v}{\sigma, \mathsf{output}\ ch\ e \xrightarrow{ch(v)} \sigma, \mathsf{skip}, \cdot}\ \textsc{output} \qquad \frac{[\![e]\!]_\sigma = v}{\sigma, \mathsf{trigger}\ id.\mathsf{Ev}(e) \xrightarrow{\bullet} \sigma, \mathsf{skip}, id.\mathsf{Ev}(v)}\ \textsc{event-trigger}$$

$$\frac{[\![e]\!]_\sigma = v}{\sigma, \mathsf{new}(id, t, e) \xrightarrow{\bullet} \sigma[id \mapsto (v, \cdot)], \mathsf{skip}, \cdot}\ \textsc{new}$$

$$\frac{\begin{array}{c}\epsilon = \epsilon', ev \mapsto EH \\ \sigma = \sigma', id \mapsto (v, \epsilon) \qquad eh = onEv(x)\{c\} \qquad \sigma_1 = \sigma', id \mapsto (v, (\epsilon', ev \mapsto EH \cup \{eh\}))\end{array}}{\sigma, \mathsf{addEh}(id, eh) \xrightarrow{\bullet} \sigma_1, \mathsf{skip}, \cdot}\ \textsc{add-eh}$$

Figure 3.2: Operational Semantics of Commands

consumer, denoted C, and local consumer, denoted LC. The system is in producer state when an event handler is executing. The system is in consumer state when it is ready to process user inputs (i.e., no event handler is executing and no script generated events are left to be processed). The system is in local consumer state when it is ready to process script-generated events (i.e., no event handler is executing and some script-generated events still need to be processed).

We define two sets of small-step operational semantics: one for commands from event handlers for a single event and the other for managing the event loop of consumer and producer state. We write $\sigma, c \xrightarrow{\alpha} \sigma', c', E$ to denote the execution rules of a command c under the store σ, which returns an updated store σ', a new command c', and a list of events E generated while evaluating c. The outer-level rules manage the event loop and are of the form: $\sigma, c, s, E \xrightarrow{\alpha} \sigma', c', s', E'$, where σ, c, and E have the same meaning as before and s is the state of the event loop (consumer, producer, or local consumer).

Most of the rules in Figure 3.2 are straightforward. Expression semantics are standard, so we omit those rules. We summarize the ones responsible for the dynamic features we aim to model. Rule OUTPUT evaluates e under the store σ and sends the result to the URL ch. Rule EVENT-TRIGGER evaluates e under the store σ and passes the result as a parameter to the event ev associated with the object identified by id. This event is added to the event queue. Rule NEW adds a new object to the store of type t and identified

$\boxed{\kappa \xrightarrow{\alpha} \kappa'}$

$$\frac{}{\sigma, \mathsf{skip}, P, \cdot \xrightarrow{\bullet} \sigma, \mathsf{skip}, C, \cdot} \text{ PtoC} \qquad \frac{E \neq \cdot}{\sigma, \mathsf{skip}, P, E \xrightarrow{\bullet} \sigma, \mathsf{skip}, LC, E} \text{ PtoLC}$$

$$\frac{\sigma(id.ev(v)) = c}{\sigma, \mathsf{skip}, C, \cdot \xrightarrow{id.\mathsf{Ev}(v)} \sigma, c, P, \cdot} \text{ CtoP-Usr-Input} \qquad \text{CtoP-Script-Input} \; \frac{\sigma(id.ev(v)) = c}{\sigma, \mathsf{skip}, LC, (id.\mathsf{Ev}(v), E) \xrightarrow{\bullet} \sigma, c, P, E}$$

$$\frac{\sigma, c \xrightarrow{\alpha} \sigma', c', E'}{\sigma, c, P, E \xrightarrow{\alpha} \sigma', c', P, (E, E')} \text{ P}$$

Figure 3.3: Operational Semantics for Event Loop

by id. The attributes are determined by evaluating e under the store σ. No event handlers are associated with an object when it is created. Rule ADD-EH looks up an object id in the store σ and adds the event handler eh to its set of registered event handlers.

We summarize the operational semantic rules for event loops in Figure 3.3. Rule PtoC says that if there are no more commands to execute or events to process and the execution is in producer state, then it is ready to process user inputs and switches to consumer state. Note that this is the only rule for switching to consumer state, ensuring that no user input is processed until all events are processed. Rule PtoLC says that if there are no commands left to execute, but there are events to process, and the execution is in the producer state, then it is ready to process script generated events and switches to local consumer state. Rule CtoP-Usr-Input receives a user-initiated event ev associated with object id and parameters v. The execution switches to producer state, the body of the event handler c is looked up in the store σ and is executed next. Rule CtoP-Script-Input begins with the execution in local consumer state, indicating that there are script-generated events to process. The execution switches to producer state and the body of the event on the front of the queue, c, is looked up in the store, σ, to be executed next. Finally, rule P is responsible for executing individual commands. It takes one step in the command operational semantics and updates the store, command, and event queue, remaining in the producer state.

3.3.3 Security Policies

Before introducing declassification policies, we define our security lattice. Figure 3.4 summarizes all the constructs needed for defining security policies.

We assume a simple security lattice that has two labels H and L and a partial order $L \sqsubseteq H$. As shown in our motivating examples, events associated with dynamically-generated objects should not influence

Security label:	ℓ	$::=$	$H \mid H_\Delta \mid L$
Initial IDs:	Γ	$::=$	$\cdot \mid \Gamma, id$
Label map:	m_l	$:$	$(\mathsf{eventName} + \mathsf{chName}) \rightarrow \mathsf{arg} \rightarrow \mathsf{Lab}$
Policy context:	$\mathcal{P}$	$::=$	(Γ, m_l)
Command:	c	$::=$	$\cdots \mid x := \mathsf{declassify}(\iota, e)$
Declassification function:	$\mathcal{D}$	$:$	$(\mathsf{state} \times \mathsf{event}) \rightarrow (\mathsf{state} \times \mathsf{release\ option} \times \mathsf{event\ option})$
Release module:	$\mathcal{R}$	$::=$	$(\rho, \mathcal{D})$
Released value:	r	$::=$	$\mathsf{none} \mid \mathsf{some}(\iota, v)$
Release channel:	d	$::=$	$\cdot \mid d, (\iota, v)$

Figure 3.4: Constructs for Defining Security Policies

declassification. To enforce this, we augment our security labels with another label: H_Δ for events that are associated with such objects. These events should not be observable by low observers, nor should they be subject to declassification. In the security lattice, we treat H_Δ the same as label H. Since we do not allow dynamically generated objects to have any effect on low outputs, it is possible that we will change the behavior of otherwise benign programs. This effects how we reason about the *precision* of our enforcement mechanism, which says that the semantics of good programs should not be altered. See Section 3.4.3 for information about our precision theorem and Section 3.5 for further discussion.

We use a label context $\mathcal{P}$ to map events and network outputs to their security labels. The label context needs to map events associated with dynamically added objects correctly, therefore, we split the label mapping into two parts: Γ which records all the object IDs that are in the initial configuration (IDs of elements that the attacker knows for sure exist by reading the program), and m_l which is a function that takes an event name and the argument of the event as input and returns the corresponding security label. In other words, m_l decides the label of events and network outputs. For events, m_l uses the event type and event argument alone, not the ID of the object that the event is associated with. For network outputs, m_l takes as input the channel name and the value to be sent to that channel as arguments. We can decide the security label of a non-silent action given a label context $\mathcal{P}$. The judgment $\mathcal{P} \vdash a : \ell$ means that a non-silent action a has security label ℓ with regard to the label context $\mathcal{P}$. It is defined as follows:

$$\frac{id \notin \Gamma}{(\Gamma, m_l) \vdash id.\mathsf{Ev}(v) : H_\Delta} \qquad \frac{id \in \Gamma \qquad m_l(\mathsf{Ev}, v) = \ell}{(\Gamma, m_l) \vdash id.\mathsf{Ev}(v) : \ell} \qquad \frac{m_l(ch, v) = \ell}{(\Gamma, m_l) \vdash ch(v) : \ell}$$

To decide the label of an event $id.\mathsf{Ev}(v)$, we first check whether id is in Γ. If it is not, the label for this event is H_Δ. Otherwise, we apply m_l: $m_l(\mathsf{Ev}, v)$. Instead of using the judgment, we write $\mathcal{P}(a)$ to denote the security label of a given $\mathcal{P}$. For instance, a label context $\mathcal{P}$ with $\Gamma = \{button_0\}$, $m_l(\mathsf{click}, _) = H$ means that initially there is only $button_0$ on the page, and all click events are H. Then, if we use this label context $\mathcal{P}$ in the example in Section 3.2, we have $\mathcal{P}(button_0.\mathsf{click}(v)) = H$ and $\mathcal{P}(id_1.\mathsf{click}(v)) = H_\Delta$.

3.3.4 Declassification

Many useful scripts, such as Google Analytics, are not secure using the strict definition of noninterference, as they are designed to collect some private information about user actions. Therefore, we need to extend our model to include declassification.

We add a declassification command, where ι is the identifier of the declassification. We assume that each declassification command in a program has a unique location ι. Intuitively, declassification commands are used to wrap expressions that compute aggregates of secrets (e.g., max, min, average, total number of events, etc.). For instance, to track how much content a user reads on a page, a script may want to know how many times the space key is pressed. Each time the space key is pressed, the event handler for the key press event increments a global variable *numPress*. When the user navigates away from the page, the unload event handler will be triggered, which contains the following command to access the number of times that the user pressed the space bar: $x := \mathsf{declassify}(\iota, numPress)$.

Generalizing ideas from [91], we define operational declassification policies. We write $\mathcal{R}$ to denote such policies. $\mathcal{R}$ is a pair of a state ρ and a function $\mathcal{D}$. $\mathcal{D}$ takes as input an event and the state ρ and returns a tuple containing the value to be released (r), an event to be released, and the new state. The value to be released can either be none, indicating nothing is to be released, or $\mathsf{some}(\iota, v)$, indicating value v is to be released to declassification location ι.

We call $\mathcal{R}$ an operational policy because it specifies how declassification should work but does not provide a declarative specification as to precisely what is released. One could imagine defining a specification similar to a flow spec, specified in [19], where a formula over two traces is used to specify the declassification policy. Then, static analysis is needed to check that the operational policies satisfy the declarative specification. We leave declarative policy specification to future work.

The run-time state is augmented by a channel d for communicating declassified values. d contains mappings of a declassification location to a value. We define the $\mathsf{update}(d, r)$ and $\mathsf{read}(d, \iota)$ operations to update the value in d and read the released value from d, respectively. When r is none, the update operation just returns d unchanged.

We augment the operational semantics to handle declassification. We add the release channel d to the left of the arrow for all the local execution rules in Figure 3.2. We also add the following Declassify rule to the local execution rules. It reads from the declassification channel d the value that ι is mapped to and assigns it to x. Here, e is not evaluated, as the release policy module is supposed to evaluate e on the scripts' behalf, which we explain further towards the end of this section.

$\boxed{d, \sigma, c \xrightarrow{\alpha} \sigma', c', E}$

$$\frac{\mathsf{read}(d, \iota) = v}{d, \sigma, x := \mathsf{declassify}(\iota, e) \xrightarrow{\bullet} \sigma[x \mapsto v], \mathsf{skip}, \cdot}\ \textsc{declassify}$$

We also add the release channel d to the left of the rules governing local script input and output; that include all the rules in Figure 3.3, except the CToP-User-Input rule. The resulting set of rules may be found in Section 3.4, Figure 3.6 (they will be re-used for defining SME rules in that section).

The remaining rules, summarized below, use a new judgment $\mathcal{P} \vdash \mathcal{R}, d, \kappa \xrightarrow{\alpha} \mathcal{R}', d', \kappa'$. These rules are the new outer-most level input/output rules.

$\boxed{\mathcal{P} \vdash \mathcal{R}, d, \kappa \xrightarrow{\alpha} \mathcal{R}', d', \kappa'}$

$$\frac{\sigma(id.ev(v)) = c \qquad \mathcal{P}(id.\mathsf{Ev}(v)) \in \{L, H_\Delta\}}{\mathcal{P} \vdash \mathcal{R}, d, \sigma, \mathsf{skip}, C, \cdot \xrightarrow{id.\mathsf{Ev}(v)} \mathcal{R}, d, \sigma, c, P, \cdot}\ \textsc{In-L}$$

$$\frac{\sigma(id.ev(v)) = c \qquad \mathcal{P}(id.\mathsf{Ev}(v)) = H \qquad \mathcal{R} = (\rho, \mathcal{D}) \qquad \mathcal{D}(id.ev(v)) = (r, _, \rho') \qquad d' = \mathsf{update}(d, r)}{\mathcal{P} \vdash \mathcal{R}, d, \sigma, \mathsf{skip}, C, \cdot \xrightarrow{id.\mathsf{Ev}(v)} (\rho', \mathcal{D}), d', \sigma, c, P, \cdot}\ \textsc{In-H}$$

$$\frac{d, \kappa \xrightarrow{\alpha} \kappa'}{\mathcal{P} \vdash \mathcal{R}, d, \kappa \xrightarrow{\alpha} \mathcal{R}, d, \kappa'}\ \textsc{Out}$$

Our release function is only applied to events that are labeled H. Therefore, the runtime state includes the label context $\mathcal{P}$. The purpose of the additional rules is to compute aggregates of secret inputs using the release module, which produces the release value. Rule In-L applies when the input event is not declassified because it is either a low input (labeled L) or is not supposed to be declassified because it may leak information (labeled H_Δ). If the input event is labeled H, rule In-H applies. The declassification function $\mathcal{D}$ is applied to the current state of the release module and the input event, and returns a new state and a release value r. The declassification channel d is updated to the new release value. Note that update will not change d if r is none. Finally, rule Out applies when the system is in producer or local consumer state. It makes use of the rules in Figure 3.3.

For our example policy which releases the total number of space key presses, the state ρ can be the number of space key presses so far and the declassification function increments ρ by 1 if the input event is a space bar key press event. The analytical script's event handler for key press computes its own version in the global variable *numPress*. In Section 3.4.3, we formally define a *compatibility* condition to make sure that the release policy is true to the declassified expressions (i.e., it computes what e evaluates to). In this

example, d should be the same as the value of *numPress* when the number of key presses is released. This way, the high execution does not need the declassify primitive and the low execution relies on the release module to compute declassified values.

3.4 Extended SME for Dynamic Features

In this section, we explain how to extend secure multi-execution rules to constrain dynamic features so they cannot be leveraged by attackers to leak information. The key idea here is that all inputs related to dynamically generated elements should be separated from the release module. We define security for our dynamic reactive programs as a conditional gradual release property and prove that our rules are sound. Finally, we prove the precision and robust declassification theorems for our system. Detailed proofs and supporting definitions may be found in Appendix A.

3.4.1 SME with Declassification

We write Σ to denote the secure multi-execution configuration. Σ is composed of the release policy $\mathcal{R}$, the declassification channel d, and two execution configurations κ_L and κ_H executing at security levels L and H, respectively.

$$\begin{array}{llll} \textit{SME Configuration} & \Sigma & ::= & \mathcal{R}, d; \kappa_L; \kappa_H \\ \textit{SME Execution Traces} & T & ::= & \Sigma \mid \Sigma \stackrel{\alpha}{\Longrightarrow} T \end{array}$$

We write $\mathcal{P} \vdash \Sigma \stackrel{\alpha}{\Longrightarrow} \Sigma'$ to denote the small step operational semantics for SME, which consists of rules that coordinate between the high and low executions. We write T to denote the execution traces of SME, which is a sequence of transitions.

Summaries of the SME rules are shown in Figure 3.7 and 3.8 and handle user inputs and outputs, respectively. If the input event's label is H, it is subject to declassification. We need to apply the release policy to the event to update the state of the release module, update the declassification channel, and compute the projected event that the low execution is allowed to see (if any). Rule SMEI-NR1 states that if the low execution is not allowed to see the input event (the projected event is emp), then low execution stays in the consumer state, and event handlers associated with this input event are scheduled to run in only the high execution. Rule SMEI-R applies when the H input event is projected to e_L. In this case, both the high and low execution move to producer state and start executing the event handlers for the events that they see.

Rule SMEI-NR2 applies when the input event's label is H_Δ, indicating that this input potentially interferes with the release policies. Therefore, the release module remains the same and the low execution stays

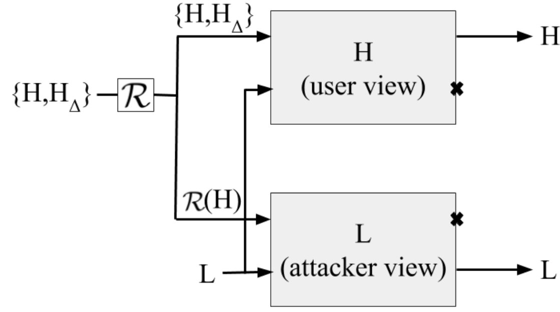


Figure 3.5: The high execution receives all inputs, unchanged. The low execution receives L inputs and released H inputs through the release module. Note that the release module throws out the H_Δ inputs so that they do not interfere with the declassification policy.

in consumer state. The last input rule SMEI-L states that when the input event is labeled L, both executions see the same input and execute event handlers matching that event. A depiction of the relationship between H and H_Δ labels and declassification may be found in Figure 3.5.

The output rules make use of consumer and producer states, which are defined as follows.

$$\begin{array}{lll} \mathsf{producer}(\kappa) & \text{iff} & \exists \sigma, c, E \text{ s.t. } \kappa = (\sigma, c, P, E) \\ \mathsf{consumer}(\kappa) & \text{iff} & \exists \sigma \text{ s.t. } \kappa = (\sigma, \mathsf{skip}, C, \cdot) \end{array}$$

The output rules make sure that (1) the execution that is not in consumer state runs using single execution rules (shown in Figure 3.6), (2) the low execution runs first, (3) outputs produced by an execution with the same label are allowed and (4) outputs produced by an execution with a different label are suppressed.

$\boxed{d, \kappa \xrightarrow{\alpha} \kappa'}$

$$\frac{E \neq \cdot}{d, \sigma, \mathsf{skip}, P, E \xrightarrow{\bullet} \sigma, \mathsf{skip}, LC, E}\ \textsc{ProLC} \qquad \frac{}{d, \sigma, \mathsf{skip}, P, \cdot \xrightarrow{\bullet} \sigma, \mathsf{skip}, C, \cdot}\ \textsc{ProC}$$

$$\frac{\sigma(id.ev(v)) = c}{d, \sigma, \mathsf{skip}, LC, (id.\mathsf{Ev}(v), E) \xrightarrow{\bullet} \sigma, c, P, E}\ \textsc{LCtoP} \qquad \frac{d, \sigma, c \xrightarrow{\alpha} \sigma', c', E'}{d, \sigma, c, P, E \xrightarrow{\alpha} \sigma', c', P, (E, E')}\ \mathrm{P}$$

Figure 3.6: Operational Semantic Rules for Single Execution

Notice that we use H_Δ to label all events that are related to elements that are not in the initial configuration so that these events will not be mistakenly passed to the release module for declassification. Going back to our examples in Section 3.2, this means that the click event of newly generated buttons will not be released to the low execution, even though the declassification function maps all click events to L. Thus, we effectively protect the integrity of the declassification policy, since the events that are fed to $\mathcal{R}$ are not influenced by the attacker. Moreover, the simulated click on the “agree to share GPS location” button will

$\boxed{\mathcal{P} \vdash \Sigma \stackrel{\alpha}{\Longrightarrow} \Sigma'}$

$$\frac{\mathcal{P}(id.\mathsf{Ev}(v)) = H \qquad \mathcal{D}(\rho, id.\mathsf{Ev}(v)) = (r, \mathsf{emp}, \rho') \qquad d' = \mathsf{update}(d, r) \qquad \sigma_H(id.\mathsf{Ev}(v)) = c_H}{\mathcal{P} \vdash (\rho, \mathcal{D}), d; \sigma_L, \mathsf{skip}, C, \cdot; \sigma_H, \mathsf{skip}, C, \cdot \stackrel{id.\mathsf{Ev}(v)}{\Longrightarrow} (\rho', \mathcal{D}), d'; \sigma_L, \mathsf{skip}, C, \cdot; \sigma_H, c_H, P, \cdot} \textsc{SmeI-NR1}$$

$$\frac{\mathcal{P}(id.\mathsf{Ev}(v)) = H_\Delta \qquad \sigma_H(id.\mathsf{Ev}(v)) = c_H}{\mathcal{P} \vdash \mathcal{R}, d; \sigma_L, \mathsf{skip}, C, \cdot; \sigma_H, \mathsf{skip}, C, \cdot \stackrel{id.\mathsf{Ev}(v)}{\Longrightarrow} \mathcal{R}, d; \sigma_L, \mathsf{skip}, C, \cdot; \sigma_H, c_H, P, \cdot} \textsc{SmeI-NR2}$$

$$\frac{\begin{array}{c}\mathcal{P}(id.\mathsf{Ev}(v)) = H \\ \mathcal{D}(\rho, id.\mathsf{Ev}(v)) = (r, e_L, \rho') \qquad d' = \mathsf{update}(d, r) \qquad \sigma_L(e_L) = c_L \qquad \sigma_H(id.\mathsf{Ev}(v)) = c_H\end{array}}{\mathcal{P} \vdash (\rho, \mathcal{D}), d; \sigma_L, \mathsf{skip}, C, \cdot; \sigma_H, \mathsf{skip}, C, \cdot \stackrel{id.\mathsf{Ev}(v)}{\Longrightarrow} (\rho', \mathcal{D}), d'; \sigma_L, c_L, P, \cdot; \sigma_H, c_H, P, \cdot} \textsc{SmeI-R}$$

$$\frac{\mathcal{P}(id.\mathsf{Ev}(v)) = L \qquad \sigma_L(id.\mathsf{Ev}(v)) = c_L \qquad \sigma_H(id.\mathsf{Ev}(v)) = c_H}{\mathcal{P} \vdash \mathcal{R}, d; \sigma_L, \mathsf{skip}, C, \cdot; \sigma_H, \mathsf{skip}, C, \cdot \stackrel{id.\mathsf{Ev}(v)}{\Longrightarrow} \mathcal{R}, d; \sigma_L, c_L, P, \cdot; \sigma_H, c_H, P, \cdot} \textsc{SmeI-L}$$

Figure 3.7: SME Input Rules

$\boxed{\mathcal{P} \vdash \Sigma \stackrel{\alpha}{\Longrightarrow} \Sigma'}$

$$\frac{\neg\mathsf{consumer}(\kappa_L) \qquad \mathsf{producer}(\kappa_H) \qquad d, \kappa_L \stackrel{\alpha}{\longrightarrow} \kappa_L' \qquad \mathcal{P}(\alpha) = L}{\mathcal{P} \vdash \mathcal{R}, d; \kappa_L; \kappa_H \stackrel{\alpha}{\Longrightarrow} \mathcal{R}, d; \kappa_L'; \kappa_H} \textsc{SmeO-LL}$$

$$\frac{\neg\mathsf{consumer}(\kappa_L) \qquad \mathsf{producer}(\kappa_H) \qquad d, \kappa_L \stackrel{\alpha}{\longrightarrow} \kappa_L' \qquad \mathcal{P}(\alpha) = H \text{ or } \alpha = \bullet}{\mathcal{P} \vdash \mathcal{R}, d; \kappa_L; \kappa_H \stackrel{\bullet}{\Longrightarrow} \mathcal{R}, d; \kappa_L'; \kappa_H} \textsc{SmeO-LH}$$

$$\frac{\neg\mathsf{consumer}(\kappa_H) \qquad \mathsf{consumer}(\kappa_L) \qquad d, \kappa_H \stackrel{\alpha}{\longrightarrow} \kappa_H' \qquad \mathcal{P}(\alpha) = H}{\mathcal{P} \vdash \mathcal{R}, d; \kappa_L; \kappa_H \stackrel{\alpha}{\Longrightarrow} \mathcal{R}, d; \kappa_L; \kappa_H'} \textsc{SmeO-HH}$$

$$\frac{\neg\mathsf{consumer}(\kappa_H) \qquad \mathsf{consumer}(\kappa_L) \qquad d, \kappa_H \stackrel{\alpha}{\longrightarrow} \kappa_H' \qquad \mathcal{P}(\alpha) = L \text{ or } \alpha = \bullet}{\mathcal{P} \vdash \mathcal{R}, d; \kappa_L; \kappa_H \stackrel{\bullet}{\Longrightarrow} \mathcal{R}, d; \kappa_L; \kappa_H'} \textsc{SmeO-HL}$$

Figure 3.8: SME Output Rules

also not be given to the release module. This event will be placed in the local event queue, E, and will not affect the declassification state, so the GPS location will not be leaked.

3.4.2 Soundness

In Section 2.5, we informally discussed knowledge and security based on an initial configuration σ_0. Here, we define these terms based on execution traces, T, for SME. First, we define $\mathsf{iruns}(\sigma_0, \mathcal{P}, \mathcal{R})$ to be the set of SME execution traces starting from the initial state $\Sigma_0 = (d_0, \mathcal{R}; (\sigma_0, \mathsf{skip}, C, \cdot); (\sigma_0^-, \mathsf{skip}, C, \cdot))$, where σ_0^- denotes the same store as σ_0 with all the declassification commands removed and d_0 as the default

$\boxed{T \Downarrow_L^{\mathcal{P}} = \tau}$

$$\frac{}{(\cdot) \Downarrow_L^{\mathcal{P}} = \cdot} \qquad \frac{T' \in \mathsf{runs}(\Sigma', \mathcal{R}', \mathcal{P}) \qquad \Sigma \not\approx_L \Sigma' \qquad \alpha \in \mathsf{in}(T)}{(\mathcal{P} \vdash \Sigma \xrightarrow{\alpha} T') \Downarrow_L^{\mathcal{P}} = \mathcal{R}_{\mathcal{P}}(\alpha) :: T' \Downarrow_L^{\mathcal{P}}}$$

$$\frac{T' \in \mathsf{runs}(\Sigma', \mathcal{R}', \mathcal{P}) \qquad \Sigma \not\approx_L \Sigma' \qquad \alpha \notin \mathsf{in}(T)}{(\mathcal{P} \vdash \Sigma \xrightarrow{\alpha} T') \Downarrow_L^{\mathcal{P}} = \alpha :: T' \Downarrow_L^{\mathcal{P}}} \qquad \frac{T' \in \mathsf{runs}(\Sigma', \mathcal{R}', \mathcal{P}) \qquad \Sigma \approx_L \Sigma'}{(\mathcal{P} \vdash \Sigma \xrightarrow{\alpha} T') \Downarrow_L^{\mathcal{P}} = T' \Downarrow_L^{\mathcal{P}}}$$

Figure 3.9: Projection of Traces

declassification channel that maps all possible declassification location ι to a default value. We call Σ_0 an initial SME configuration from σ_0.

The knowledge of an attacker, $\mathcal{K}(T, \sigma_0, \mathcal{P}, \mathcal{R})$ is the set of possible input traces that could produce an execution trace that is observationally equivalent to T.

Definition 2 (Attacker Knowledge)**.** *An attacker's knowledge after observing trace T beginning from state σ_0 with policy context $\mathcal{P}$ and release module $\mathcal{R}$ denoted $\mathcal{K}(T, \sigma_0, \mathcal{P}, \mathcal{R})$ is defined as the set of traces τ_i s.t. $\exists T' \in$ $\mathsf{iruns}(\sigma_0, \mathcal{P}, \mathcal{R})$ with $T \approx_L^{\mathcal{P}} T'$ and $\tau_i = \mathsf{in}(T')$*

To determine when two execution traces are observationally equivalent, we must first determine when two configurations are observationally equivalent and define the observation of a trace.

We consider two SME configurations, $\Sigma_1 = \mathcal{R}_1, d_1; \kappa_{L1}; \kappa_{H1}$ and $\Sigma_2 = \mathcal{R}_2, d_2; \kappa_{L2}; \kappa_{H2}$, observationally equivalent whenever their low executions are in the same state and they are affected by declassification equivalently ($\mathcal{R}_1 = \mathcal{R}_2$, $d_1 = d_2$, and $\kappa_{L1} = \kappa_{L2}$). It follows that the observation at the level L of a trace, T, under the label context $\mathcal{P}$, denoted $T \Downarrow_L^{\mathcal{P}}$, is the sequence of inputs and outputs that results in some change in the low execution or declassification policy. Examining our SME rules reveals that this observation is the declassified high inputs, the low inputs, and the low outputs. Formally, $T \Downarrow_L^{\mathcal{P}}$ is defined in Figure 3.9. Here :: denotes concatenation, and $\mathsf{runs}(\Sigma, \mathcal{R}, \mathcal{P})$ is the set of execution traces beginning from Σ with release policy $\mathcal{R}$ and label context $\mathcal{P}$.

We define our security property for SME, which states that the attacker cannot gain more knowledge about secret user inputs as the system runs, except for what has been released. Formally:

Definition 3 (Security)**.** *A configuration σ_0 is secure w.r.t. the label context $\mathcal{P}$ and release policy $\mathcal{R}$ against attackers at level L, if for all traces T, actions α, and configurations Σ s.t. $(T \xrightarrow{\alpha} \Sigma) \in \mathsf{iruns}(\sigma_0, \mathcal{P}, \mathcal{R})$, $\mathcal{K}(T \xrightarrow{\alpha} \Sigma, \sigma_0, \mathcal{P}, \mathcal{R}) \supseteq_{\preceq} \mathcal{K}(T, \sigma_0, \mathcal{P}, \mathcal{R})$*

Definition 3 is progress sensitive. For instance, if a confidential value determines whether the execution reaches a consumer state, then it is not secure under this definition. The attacker can refine her knowledge

about the confidential value based on whether the system is making progress to process inputs. Our SME rules are not secure by Definition 3. To prove this statement for our rules, we want to show that all the shorter traces in $\mathcal{K}(T, \sigma_0, \mathcal{P}, \mathcal{R})$ are prefixes of longer traces in $\mathcal{K}(T \overset{\alpha}{\Longrightarrow} \Sigma, \sigma_0, \mathcal{P}, \mathcal{R})$. Consider the situation where α is an input event. If the shorter trace is currently processing an event handler containing an infinite loop, it will never return to a consumer state to accept input. Therefore, this trace is not a prefix of a longer trace in $\mathcal{K}(T \overset{\alpha}{\Longrightarrow} \Sigma, \sigma_0, \mathcal{P}, \mathcal{R})$.

Instead, we consider progress-insensitive security. We define a trace which makes progress as:

$$\mathsf{prog}(T, \mathcal{P}) \text{ iff } T = \mathcal{P} \vdash \Sigma_0 \Longrightarrow^* \Sigma \text{ and } \exists T' \text{ s.t. } T' = \mathcal{P} \vdash \Sigma \Longrightarrow^* \Sigma_C \text{ and } \mathsf{consumer}(\Sigma_C)$$

And we limit our set of knowledge to the traces that make progress.

Definition 4 (Progress Knowledge)**.** *An attacker's knowledge after observing T beginning from state σ_0 with policy context $\mathcal{P}$, release module $\mathcal{R}$, when they know the system will continue to make progress, denoted $\mathcal{K}_p(T, \sigma_0, \mathcal{P}, \mathcal{R})$ is defined as the set of traces τ_i s.t. $\exists T' \in \mathsf{iruns}(\sigma_0, \mathcal{P}, \mathcal{R})$ with $T \approx_L^{\mathcal{P}} T'$, $\tau_i = \mathsf{in}(T')$, and $\mathsf{prog}(T', \mathcal{P})\}$*

Then, we can update our definition of security to be progress-insensitive by limiting the shorter trace to those capable of making progress.

Definition 5 (Progress Insensitive Security)**.** *A configuration σ_0 is secure w.r.t. the label context $\mathcal{P}$ and release policy $\mathcal{R}$ against an attacker at level L, if for all traces T, actions α, and configurations Σ s.t. $(T \overset{\alpha}{\Longrightarrow} \Sigma) \in \mathsf{iruns}(\sigma_0, \mathcal{P}, \mathcal{R})$, $\mathcal{K}(T \overset{\alpha}{\Longrightarrow} \Sigma, \sigma_0, \mathcal{P}, \mathcal{R}) \supseteq_{\preceq} \mathcal{K}_p(T, \sigma_0, \mathcal{P}, \mathcal{R})$*

We prove that our SME rules are sound, formally:

Theorem 6 (Soundness)**.** *$\forall \mathcal{P}, \mathcal{R}, \sigma_0$, s.t. σ_0 is secure w.r.t. the label context $\mathcal{P}$ and release policy $\mathcal{R}$ against an attacker at level L.*

As stated previously, to prove this statement, we want to show that all the shorter traces in $\mathcal{K}(T, \sigma_0, \mathcal{P}, \mathcal{R})$ are a prefix of a longer trace in $\mathcal{K}(T \overset{\alpha}{\Longrightarrow} \Sigma, \sigma_0, \mathcal{P}, \mathcal{R})$. This is to say, any shorter trace can be expanded to an execution which is observationally equivalent to $\mathcal{P} \vdash T \overset{\alpha}{\Longrightarrow} \Sigma$. If α is not observable (e.g., high output), the shorter trace is already observationally equivalent to $\mathcal{P} \vdash T \overset{\alpha}{\Longrightarrow} \Sigma$. Otherwise, we need to show that the shorter trace can take an equivalent step. This is the intuition behind the following lemma:

Lemma 7 (Strong One-step)**.** *If $T_1 = \mathcal{P} \vdash \Sigma_1 \overset{\alpha}{\Longrightarrow} \Sigma_1'$ with $\Sigma_1 \not\approx_L \Sigma_1'$, $\Sigma_1 \approx_L \Sigma_2$, and $\mathsf{prog}(\Sigma_2, \mathcal{P})$ then $\exists \Sigma_2', \tau$ s.t. $T_2 = \mathcal{P} \vdash \Sigma_2 \Longrightarrow^* \Sigma_2'$ with $T_1 \approx_L^{\mathcal{P}} T_2$ and $\Sigma_1' \approx_L \Sigma_2'$*

In addition to α being observable, this lemma requires that the two traces be in equivalent states before the step. This follows from an additional lemma:

Lemma 8 (Equivalent Traces give Equivalent States). *If* $T_1 = \mathcal{P} \vdash \Sigma_1 \Longrightarrow^* \Sigma_1'$ *and* $T_2 = \mathcal{P} \vdash \Sigma_2 \Longrightarrow^* \Sigma_2'$ *with* $\Sigma_1 \approx_L \Sigma_2$ *and* $T_1 \approx_L^{\mathcal{P}} T_2$*, then* $\Sigma_1' \approx_L \Sigma_2'$.

Lemma 7 is proven by examining each case of $\mathcal{E} :: \mathcal{P} \vdash \Sigma_1 \stackrel{\alpha}{\Longrightarrow} \Sigma_1'$, where α is observable, and showing that the second trace can take an equivalent step. Lemma 8 is proven by induction over the length of the trace T_1.

3.4.3 Precision

One desirable property of SME is precision, which states that the semantics of *good* programs should not be altered. Good programs are those that are compatible with the declassification policies and do not leak information outside of what is released by declassification. The formal definitions of *compatibility* and *no leak outside declassification* are very similar to those in prior work [91].

Definition 9 (Compatibility). *We say that a state* σ *is compatible with a release policy* $\mathcal{R}$ *and label context* $\mathcal{P}$*, when for all* τ $\mathcal{P} \vdash d_0, \mathcal{R}; \kappa \stackrel{\tau}{\longrightarrow}^* d', \mathcal{R}'; \kappa'$ *iff* $\kappa \stackrel{\tau}{\longrightarrow}^* \kappa'$ *where* d_0 *is the initial release channel,* $\kappa = (\sigma, \mathsf{skip}, \mathsf{C}, \cdot)$.

We use the judgement $\kappa \longrightarrow^* \kappa'$ to denote program execution without SME. Definition 9 confirms that the release function computes the same declassified values as the script would if it ran without SME. We say that a script does not leak outside of declassification if release policies that affect the inputs the same way always produce the same outputs. If the outputs differed, it must be the case that the secret inputs influenced the outputs, outside of what was declassified. We write $\tau|_\ell^{\mathcal{P}}$ to denote the projection of an action sequence to label ℓ under the label context $\mathcal{P}$, and $\mathcal{R}_{\mathcal{P}}^*(\tau)$ to denote repeatedly applying $\mathcal{R}$ to each input event in τ with label context $\mathcal{P}$.

Definition 10 (No Leak Outside Declassification). *We say that a state* σ *is has no leak outside declassification, if for all label context* $\mathcal{P}$*, release policies* $\mathcal{R}$*,* $\mathcal{R}'$*,* $\mathcal{R}_1$*,* $\mathcal{R}_1'$ *and traces* τ_{i1}*,* τ_{i2}*, s.t.* $\mathcal{R}_{\mathcal{P}}^*(\tau_{i_1}) = \mathcal{R}_{\mathcal{P}}'^*(\tau_{i_2})$*, for all* τ_1 *and* τ_2 $\mathcal{P} \vdash t_1 = d_0, \mathcal{R}; \kappa \longrightarrow^* d', \mathcal{R}_1; \kappa'$ *and* $\mathcal{P} \vdash t_2 = d_0, \mathcal{R}'; \kappa \longrightarrow^* d', \mathcal{R}_1'; \kappa'$*, and* $\mathsf{in}(t_1) = \tau_{i1}$*,* $\mathsf{in}(t_2) = \tau_{i2}$*, it is the case that* $\mathsf{out}(t_1)|_L^{\mathcal{P}} = \mathsf{out}(t_2)|_L^{\mathcal{P}}$.

We say that an execution trace is a complete run if it starts and finishes in consumer state.

$$T \text{ is a complete run iff } \mathcal{P} \vdash T = \Sigma \Longrightarrow^* \Sigma' \text{ and consumer}(\Sigma) \text{ and consumer}(\Sigma')$$

We prove the following precision theorem. Similar to prior work on SME, our precision theorem concerns observations at each security level.

Theorem 11 (Precision)**.** *For all $\mathcal{P}$, $\mathcal{R}$, σ and κ_1, $\kappa_1 = (\sigma, \mathsf{skip}, C, \cdot)$, σ is compatible with $\mathcal{P}$ and $\mathcal{R}$, and does not leak outside declassification, then for all complete runs T and t s.t. $\mathcal{P} \vdash T = \Sigma_1 \Longrightarrow^* \Sigma_2$, $t = \kappa_1 \longrightarrow^* \kappa_2$, and $\Sigma_1 = d, \mathcal{R}; \kappa_1; \kappa_1$, and $\mathsf{in}(T) = \mathsf{in}(t)$ imply $\mathsf{out}(T)|^{\mathcal{P}}_H = \mathsf{out}(t)|^{\mathcal{P}}_H$ and $\mathsf{out}(T)|^{\mathcal{P}}_L = \mathsf{out}(t)|^{\mathcal{P}}_L$.*

One interesting observation here is that this precision theorem is fairly weak as it requires both the SME and single execution traces exist. In Section 3.5, we show that programs are not precise using a stronger definition due to dynamic features.

3.4.4 Robust Declassification

Robust declassification requires that active attackers cannot learn more than passive attackers. We say that σ_2 contains more active components than σ_1 ($\sigma_1 <_A \sigma_2$) if it contains more script-generated event handlers and objects but is otherwise the same.

For our robustness theorem, we consider an interleaving of inputs to σ_1 with additional inputs (corresponding to the additional components, denoted τ_Δ) as the input to σ_2. We formally define an interleaving of two traces as follows:

$$\frac{}{\tau_1 \bowtie \cdot = \tau_1} \qquad \frac{\tau_1 = \tau_1' :: \tau_1'' \qquad \tau_2 = \alpha :: \tau_2'}{\tau_1 \bowtie \tau_2 = \tau_1' :: \alpha :: (\tau_1'' \bowtie \tau_2')}$$

We also define the following relation for A and B, sets of traces:

$$A \subseteq_{\ll} B \text{ iff } \forall \tau \in A, \exists \tau', \tau_\Delta \text{ with } \tau' \in B, \text{ and } \tau \bowtie \tau_\Delta = \tau'$$

Because the additional inputs to σ_2 are from script-generated components, all these inputs have the label H_Δ. We denote this formally as $\mathsf{dom}(\tau_\Delta) \cap \Gamma = \emptyset$, meaning that the objects associated with inputs from dom were added to the system. We define the domain of a set of inputs:

$$\frac{\tau = \tau' :: \alpha}{\mathsf{dom}(\tau) = \mathsf{dom}(\alpha) \cup \mathsf{dom}(\tau')} \qquad \frac{\alpha = id.ev(v)}{\mathsf{dom}(\alpha) = \{id\}} \qquad \frac{\alpha = ch(v)}{\mathsf{dom}(\alpha) = \{\ \}}$$

One caveat is that we need to account for non-progress behavior (divergent in non-consumer state) introduced by the additional event handlers in σ_2. We consider an execution trace divergent if it never reaches a consumer state.

Theorem 12 (Robust Declassification)**.** *$\forall \sigma_1, \sigma_2, \mathcal{P}, \mathcal{R}$ s.t. $\sigma_1 <_A \sigma_2$, and $\forall T_1 \in \mathsf{iruns}(\sigma_1, \mathcal{P}, \mathcal{R})$ s.t. T_1 is a complete run, $\forall T_2 \in \mathsf{iruns}(\sigma_2, \mathcal{P}, \mathcal{R})$ s.t. T_2 is a complete run, with $\tau_i = \mathsf{in}(T_1)$, $\mathsf{in}(T_2) = \tau_i \bowtie \tau_\Delta$, $\mathsf{dom}(\tau_\Delta) \cap \Gamma = \emptyset$, $\mathcal{K}(T_1, \sigma_1, \mathcal{P}, \mathcal{R}) \subseteq_{\ll} \mathcal{K}(T_2, \sigma_2, \mathcal{P}, \mathcal{R})$ or σ_2 diverges.*

We prove this by defining a simulation relation between the configurations in T_1 and T_2. As mentioned earlier, the additional input to T_2 will not affect the state of the release module and can only be processed by the high execution. Therefore, after processing these inputs, the configurations in T_2 still relate to the same configurations in T_1.

Allowing active attackers to cause the system to enter a state where it cannot receive inputs is consistent with our progress-insensitive definition of the attacker's knowledge, which allows the system to leak information through whether it makes progress.

Going back to our example in Section 3.2, we can instantiate σ_2 as the configuration including the event handler with the problematic branching statement, and σ_1 as this configuration minus this event handler. The additional events will be $id_2.\mathsf{click}(v)$. If $id_2.\mathsf{click}(v)$ were given to the low execution, then the knowledge of the active attacker refines that of the passive one, as it knows the previous input must be 2. However, because the button with ID id_2 was added to the system, the event is not given to the low execution, so the active attacker learns no more than the passive one. Robust declassification ensures that this is always the case. The attacker that generates objects and registers event handlers learns no more than the attacker who merely watches the system run.

3.5 Discussion

Precision Our precision theorem is weak in the sense that we require the program leak no information outside of what is released by declassification. Consider a program that generates new elements and event handlers (denoted $\Delta\sigma$), which output to low channels when triggered. If all the events associated with these new items are otherwise low events, then this is a benign program since there is no secret involved. However, it does not satisfy the *no leak outside of declassification* condition. The reason is that the events associated with $\Delta\sigma$ are given the H_Δ label by our system and are expected to have no effects on low outputs, which is not the case here. Our SME rules will suppress legitimate low outputs from this program, as a result. However, SME cannot do much better because the run-time has no way of knowing whether $\Delta\sigma$ depends on secrets or not.

Integrity and Endorsement Dual to confidentiality is integrity, whose non-interference property states that untrusted (low integrity) data cannot affect trusted (high integrity) data. Considering integrity in our system would provide an opportunity for more fine-grained declassification policies. For instance, instead of preventing any script-generated input from affecting declassification, the trustworthiness (i.e., integrity) of the source could be considered. User inputs (high integrity) should be allowed to influence

declassification policies whereas scripts (low integrity) should not. This connection between robust declassification and integrity has been studied [66, 101, 31]. In the next Chapter, we explore how to extend our model to include integrity.

3.6 Summary

In this chapter, we investigated how dynamic features of JavaScript can be used to leak information by abusing declassification policies. We designed new SME rules to enforce strict separation between dynamically generated components and the declassification module. To state security properties in the presence of declassification policies, we use a knowledge-based progress-insensitive definition of security and prove that our enforcement mechanism is sound. We also prove precision and robust declassification properties of our SME rules.

Chapter 4

Robust Declassification via Limiting Attacker Influence

In this chapter we describe a technique for robust declassification where we keep track of which elements were added by untrusted code and prevent events associated with those elements from being declassified.

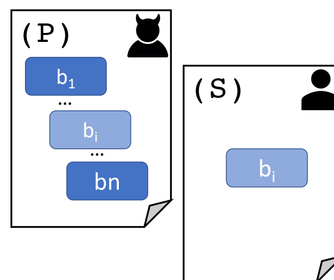
4.1 Overview

In the previous chapter, we showed that the dynamic features on webpages (including adding new page elements to the DOM or registering new event handlers) can leak sensitive information. The problem was that an untrusted party with the ability to add code to the page (such as via a third-party script) could leverage dynamic features to influence when declassification happens. The solution presented there ensures that an attacker cannot influence declassification via dynamic elements by disallowing events from dynamic features from being declassified. However, while this technique is provably secure, it risks altering the behavior of secure programs involving dynamic features and could prevent declassification in the benign example we describe above.

In this chapter, we develop a more fine-grained technique for protecting dynamic features from leaking secrets due to declassification by keeping track of the attacker's influence on the page and only preventing declassification when it involves code which is not trusted by the user. This chapter makes the following contributions: We present an example to show that preventing declassification for all dynamic elements is too restrictive to be practical in some situations (Section 4.2.2). We highlight leaks due to interactions *between* executions which can arise even if the user trusts the copy of the page they are interacting with (Section 4.2.3). This is different from the example in Section 3.2 where the user is interacting with attacker-controlled code directly. In Section 4.3 we present SME semantics based on the ones described

```
onKeypress(secret) :
  case secret :
    | 1 : new(b_1);
          addEH(b_1, onClick{output_P(1)});
    ...
    | n : new(b_n);
          addEH(b_n, onClick{output_P(n)});
    | dv :new(b_1);
          addEH(b_1, onClick{output_P(1)});
          ...
          new(b_n);
          addEH(b_n, onClick{output_P(n)});
```

(a) Event handler added by the attacker.



(b) Resulting attacker view (P) and user view (S) of the page.

Figure 4.1: Example of dynamic features causing leaks. The dv case guarantees that the attacker copy will have a matching button (colored light blue) to capture the declassified event and leak the secret.

in the previous chapter and in Section 4.4 we extend the model to incorporate techniques from [101] to achieve a more permissive security monitor—without sacrificing security. This new monitor composes taint tracking with SME. Tracking the trustworthiness of page elements and their event handlers allows us to prevent attacker-controlled code from influencing declassification, without entirely sacrificing the ability to perform declassification in untrusted executions. We present novel security conditions where robust declassification naturally follows from our influence-based noninterference property (Section 4.6). In this chapter, we focus on robust declassification for ease of understanding, but the same techniques can be used to show transparent endorsement [31].

4.2 Motivating Examples

In Chapter 3, we show that an attacker can leak additional information when events from dynamically generated elements are declassified. We review an example of the leaks due to dynamically generated features and the proposed solution from Section 3.3.

4.2.1 Leaks when declassifying dynamic elements

Consider a 2-point security lattice with elements - $\{P, S\}$ where $P \sqsubseteq S$, and a web page with a policy that declassifies click events and occurrences of keypress events (but *which* key was pressed should remain secret). The P copy of the page is visible to the attacker and receives only the public (or declassified) events, while the S copy is visible to the user and receives all the events. Suppose the attacker registers the event handler shown in Figures 4.1a which runs whenever a key is pressed and adds a different button to the page, depending on which key was pressed (which key was pressed is stored in *secret*). If the user

presses key i on their keyboard, this event handler would add button b_i in the S copy of the page based on the actual value of the secret. The P copy of the page receives the event with a default value dv to hide which key was pressed, so the event handler adds *all* possible buttons to the page. When the user clicks on b_i on their view of the page (the S copy), it is also declassified to the P copy, which is guaranteed to have a matching button to capture the event. The on*Click* event handler executes the statement output(P, i). Since outputs to P channels are allowed in the P execution, this leaks which key was pressed to the attacker.

To prevent this unintended leak, we used an additional label S_Δ for dynamically added secrets and restrict declassification to only those secrets that are labeled S, i.e., events dispatched on elements labeled S_Δ are never declassified to P. Accordingly, in the previous example, the button b_i added in the S copy is labeled S_Δ; hence, the mouse-click on b_i is not declassified to P, thereby preventing the attacker from learning which key was pressed. However, while this solution prevents unintentional leaks, it can be too restrictive to be practical, as we show below.

4.2.2 Restrictiveness of technique from Chapter 3

Consider a situation where an online shop wants to know which of their products are receiving the most attention (but not necessarily purchased). They use JavaScript to dynamically display products on their site depending on what the user has searched for. To measure product popularity, they use a third-party analytics library to track which products users are clicking on their site. Because they do not want the third-party library to have access to *all* of the user's private information, they treat the script as *P*ublic, meaning they can only communicate with the analytics service via *P*ublic channels through *P*ublic SME executions. To give the analytics library access to the relevant click information, the shop employs a declassification policy where the coordinates of individual clicks are *S*ecret, but which product is clicked may be declassified to the analytics library. With the solution described above, everything dynamically loaded on the page (even ones not controlled by the attacker) are labeled S_Δ and excluded from declassification, and thus, the online shop will not be able to perform their analytics unless the shop gives them unrestricted access to *S*ecret information, which is antithetical to the goals of IFC.

The reason the earlier example (Figure 4.1b) leaked more than intended is that the attacker leveraged the declassification policy to leak information by adding buttons to the page. Meanwhile, the products added to the web page described above are added by the shop itself. The underlying problem is not the dynamic page elements, but their *source*. Instead of disallowing *any* dynamic features to influence declassification, an intuitive fix would simply restrict the attacker's influence. This involves protecting the *integrity* of the data, which is dual to *confidentiality*.

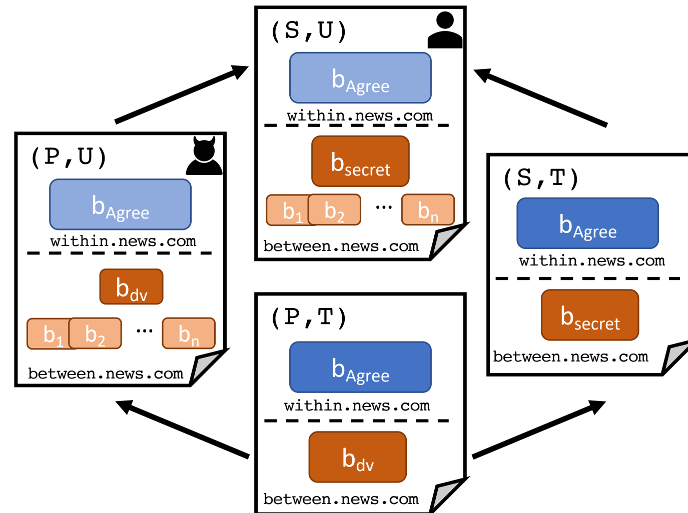


Figure 4.2: Information is allowed to flow in the direction of the arrows. The attacker can influence *U*ntrusted executions to add page elements or event handlers to try to manipulate declassification directly within an execution (blue case) or indirectly between executions (orange case).

4.2.3 Tracking integrity in SME

Consider a simple 4-point security lattice with 2 confidentiality labels (*P*ublic and *S*ecret) and 2 integrity labels (*T*rusted and *U*ntrusted). Information is allowed to flow from *P*ublic to *S*ecret, as well as *T*rusted to *U*ntrusted. Then the complete security lattice is a diamond with (P, T) at the bottom, (S, U) at the top, and the other labels (P, U) and (S, T) in between. SME can be used to enforce information flow policies drawn from this lattice by running one execution for each of these 4 security levels as shown in Figure 4.2. In this model, the attacker, and other *U*ntrusted parties, like *ad.com*, are only able to add code to the *U*ntrusted executions, while *T*rusted parties (like *news.com* in our examples) may add code to any of the executions. In our examples, the (P, U) execution can communicate with the attacker via *ad.com* and the user is shown the (S, U) version of the webpage.

In the following examples, we show that attackers have influence over the decision to declassify whether the user interacts directly with their code (similar to the leaks via *Dynamically generated elements* from Chapter 3.2) or indirectly, when a declassification triggers their code in *another* execution. We will return to these examples in later sections as well.

Leaks within an execution Suppose a user visits a webpage (*within.news.com*) which explains that it will share their account preferences with advertisers (*ad.com*), but only if they click the "Agree" button (identified in the code as b_{Agree}) to consent. When the page loads, *ad.com* adds a large b_{Agree} button at the top of the page with the text "Click me!", as in Figure 4.2, where the buttons coming from *ad.com* are light

blue and the ones from *within.news.com* are dark blue. A curious user may click the button, not realizing it will share their settings. We call this a leak *within* an execution because the user is interacting directly with attacker-controlled code. This is similar to the example from Section 3.2 where the attacker adds elements to the page the user is interacting with directly.

Leaks between executions Consider another webpage (*between.news.com*) which has a policy that keypress events are Secret, but clicks may be declassified from Secret to Public. *news.com* installs an event handler which adds a different button to the page, depending on which key the user presses (similar to onKeypress in Figure 4.1, without the dv case). Meanwhile, *ad.com* adds all the possible buttons to the page and registers an event handler which is triggered by a click to send them a message, telling them which button was clicked (similar to the dv case from the onKeypress event handler in Figure 4.1). The resulting page is shown in Figure 4.2, where the dark orange buttons were added by *news.com* and the light orange buttons were added by *ad.com*. Note that because *news.com* is Trusted, the dark orange buttons are added to all the copies of the webpage, including the Untrusted ones.

Like the leak from Section 3.2, if the user clicks the b_{secret} button on the (S, U) page, the event will be declassified to the (P, U) execution, which is guaranteed to have a matching button to capture the event and leak the keypress to the attacker. We call this a leak *between* executions because the user is interacting with code added by the host page which triggers attacker-controlled code in *another* execution. This example highlights that it is not enough to only look at the page the user is interacting with, we also need to consider the executions capturing the declassified events.

To prevent the attacker from influencing declassification, one approach would be to extend the solution from Chapter 3 to apply to events *originating from* dynamic elements in Untrusted executions (which might include attacker-controlled code), as well as events being *released to* dynamic elements in Untrusted executions. But as we described above, this would also prevent innocent declassifications, like the online shop in the previous example. Likewise, it would not be enough to simply prevent the user from interacting directly with attacker-controlled code by showing them the (S, T) copy of the page instead of the (S, U) copy, because this would still be susceptible to the leaks between executions.

To prevent these leaks without sacrificing functionality, we introduce a taint-tracking technique in Section 4.4.2 which attaches labels to page elements (and their event handlers) that reflect the trustworthiness of the source of the code. We check that the user trusts the code they are interacting with directly to decide if a declassification should be triggered (preventing leaks *within* executions), as well as the code in other executions to decide whether they should receive the event (preventing leaks *between* executions).

Security lattice:	$\mathcal{L}$	$::=$	$\mathcal{L}_c \times \mathcal{L}_i$
Security property:	p	$::=$	$c \mid i$
Confidentiality/integrity label:	l_p	$\in$	$\mathcal{L}_p$
Program counter:	pc	$\in$	$\mathcal{L}$
Event:	Ev	$::=$	$\mathsf{click} \mid \mathsf{keyPress} \mid \ldots$
Event handler:	eh	$::=$	$onEv(x)\{c\}$
Security policy:	$\mathcal{P}$		
SME Traces:	T	$::=$	$K \mid \mathcal{P} \vdash T \stackrel{\alpha_l}{\Longrightarrow} K$
SME configuration:	K	$::=$	$\Sigma; ks$
SME state:	Σ	$::=$	$\cdot \mid \Sigma, pc \mapsto \sigma^G_{pc}$
Shared state:	σ^G	$::=$	σ^g, σ^{EH}
Shared variable state:	σ^g	$::=$	$\cdot \mid \sigma^g, x \mapsto v$
Shared EH state:	σ^{EH}	$::=$	$\cdot \mid \sigma^{EH}, id \mapsto (v, M)$
EH map:	M	$::=$	$\cdot \mid M, \mathsf{Ev} \mapsto EH$
Event handler set:	EH	$::=$	$\{\ \} \mid EH \cup \{eh\}$
Configuration stack:	ks	$::=$	$\cdot \mid (\kappa, pc) :: ks$
Actions:	α	$::=$	$id.\mathsf{Ev}(v) \mid ch(v) \mid \bullet$
Event queue:	E	$::=$	$\cdot \mid E, (id.\mathsf{Ev}(v), pc)$

Figure 4.3: Syntax for processing inputs and outputs.

4.3 SME with Dynamic Features

Before we introduce our taint tracking technique for robust declassification, we describe SME semantics for reactive systems with dynamic features (without declassification). Our semantics are flexible enough to work with any security lattice with any number of confidentiality and integrity labels. Similar to Chapter 3, we organize our SME semantics into three levels: the top-most level is responsible for processing inputs and outputs, looking up event handlers, and switching between executions. The mid-level manages the execution state and event handler queue for a particular execution. The lowest level runs the current event handler. While the semantics in Chapter 3 use a simple 2-point security lattice, the semantics here are general enough for any security lattice. In this section, we describe the syntax and semantics for each level of semantics, referring to our running examples from Section 4.2.3.

4.3.1 I/O Processing and EH Lookup

Here we describe the syntax and semantics for processing inputs and outputs, looking up event handlers, and switching between executions with SME. The syntax related to our input/output semantics is shown in Figure 4.3.

In this chapter, we discuss both confidentiality and integrity. These properties are considered dual to each other, and we leverage this duality in later sections to simplify our semantics and security conditions.

We parameterize our semantics on the security property, p, where $p = c$ refers to confidentiality and $p = i$ for integrity. We have both confidentiality labels, $l_c \in \mathcal{L}_c$, which tell us how much privilege is needed to access data, and integrity labels, $l_i \in \mathcal{L}_i$, which tells us how trusted a component is. We say that information may flow from (l_c, l_i) to (l'_c, l'_i) if l'_c has privilege to see data from l_c ($l_c \sqsubseteq l'_c$) and l'_i trusts data from l_i ($l_i \sqsubseteq l'_i$). Our example from before used a simple security lattice with $\mathcal{L}_c = \{P, S\}$ and $\mathcal{L}_i = \{T, U\}$ for $P \sqsubseteq S$ and $T \sqsubseteq U$, but our rules are general enough to accommodate more complex (finite) lattices.[1] The program counter pc tells us what context we are executing an event handler in and is a pair of confidentiality and integrity labels.

Events in our reactive system are associated with elements given by unique identifiers id. Event handlers of the form $\mathsf{onEv}(x)\{c\}$ run command c with argument x when the system receives event Ev (such as a click or a keypress). The security label (l_c, l_i) of an event is determined by the security policy $\mathcal{P}$. An execution trace T is zero or more steps of the top-level I/O semantics. An SME configuration K is a snapshot of the system including the SME state Σ and the configuration stack ks.

Σ keeps track of the shared state for each execution, which persists between event handlers; each security level $pc = (l_c, l_i)$ has its own store $\sigma^G_{(l_c, l_i)}$ which is the variable storage σ^g and event handler storage σ^{EH} (i.e., the DOM) for each execution. The event handler storage maps identifiers id to attributes v and event handler maps M, which maps events Ev to their respective event handlers $eh_1, ..., eh_n$. This model allows each execution to have its own copy of the DOM, whose contents may vary in privilege and trust (see Section 4.7 for a discussion about other possible DOM models). Each execution runs its event handlers separately, beginning at the top of the configuration stack ks. Each element of the configuration stack determines what event handler to run, given by configuration κ, and in which execution, given by the security level pc. We describe the individual configurations, κ, in more detail in the next section.

As the system runs, it may react to/emit various actions, α. In the reactive setting, the system waits until it receives an input which is an event triggering (zero or more) event handlers which may produce some outputs. In our case, inputs include user interactions $id.\mathsf{Ev}(v)$ which are events Ev associated with an element id (possibly) carrying some argument (e.g., which key is pressed for a keyPress event or the location of a click). Outputs are given by values sent along a channel ch. The other actions are silent $\bullet$.

The operational semantics for processing inputs and outputs, looking up event handlers, and switching between executions is shown in Figure 4.4. Rule In receives an event Ev for page element id with parameter v from the principal with privilege and trustworthiness given by pc. The security policy tells us the label on the event is pc'. We run the event handlers associated with the event in each execution with enough

[1] Prior work describes an extension of SME which can operate on infinite lattice [3]. Instead of running the program at every security level in the lattice, they only run the program at the security levels which are relevant to the input data and outputs produced by the code. We leave exploring a similar technique for our approach to future work.

$\boxed{\mathcal{P} \vdash K \overset{(\alpha,pc)}{\Longrightarrow} K'}$

$$\frac{\mathcal{P}(id.\mathsf{Ev}(v)) = pc' \qquad E = ((id.\mathsf{Ev}(v), pc'') \mid pc'' \in \mathcal{L} \text{ s.t. } pc \sqcup pc' \sqsubseteq pc'') \qquad \Sigma, E \rightsquigarrow ks}{\mathcal{P} \vdash \Sigma; \cdot \overset{(id.\mathsf{Ev}(v),pc)}{\Longrightarrow} \Sigma; ks} \text{ IN}$$

$$\frac{\mathsf{producer}(\kappa) \qquad \Sigma, \kappa \xrightarrow{ch(v)}_{pc} \Sigma', ks' \qquad \alpha = ch(v) \text{ if } \mathcal{P}(ch) = pc \qquad \alpha = \bullet \text{ otherwise}}{\mathcal{P} \vdash \Sigma; (\kappa, pc) :: ks \overset{(\alpha,pc)}{\Longrightarrow} \Sigma'; ks' :: ks} \text{ OUT}$$

$$\frac{\mathsf{producer}(\kappa) \qquad \Sigma, \kappa \xrightarrow{\alpha}_{pc} \Sigma', ks' \qquad \alpha \neq ch(v)}{\mathcal{P} \vdash \Sigma; (\kappa, pc) :: ks \overset{(\alpha,pc)}{\Longrightarrow} \Sigma'; ks' :: ks} \text{ OUT-SILENT} \qquad \frac{\mathsf{consumer}(\kappa)}{\mathcal{P} \vdash \Sigma; (\kappa, pc) :: ks \overset{(\bullet,pc)}{\Longrightarrow} \Sigma; ks} \text{ OUT-NEXT}$$

$\boxed{\Sigma, E \rightsquigarrow ks}$

$$\frac{\begin{array}{c}\Sigma(pc) = (_, \sigma^{EH}) \qquad \sigma^{EH}(pc)(id.\mathsf{Ev}(v)) = c \\ \kappa = \cdot, c, P, \cdot \qquad \Sigma, E \rightsquigarrow ks\end{array}}{\Sigma, (id.\mathsf{Ev}(v), pc) :: E \rightsquigarrow (\kappa, pc) :: ks} \text{ LOOKUP} \qquad \frac{}{\Sigma, \cdot \rightsquigarrow \cdot} \text{ LOOKUP-EMPTY}$$

Figure 4.4: Top-level SME rules for processing inputs and outputs, and looking up event handlers

privilege to see the event and who trust the event, i.e., at all executions at or above $pc \sqcup pc'$ in the security lattice. For the simple 4-point security lattice described above, the top (most restricted) label on the lattice is (S, U), meaning that *S*ecret and *U*ntrusted events will run in the executions which have enough privilege to see the event and trust the event, which is only the (S, U) copy. Meanwhile, *S*ecret *T*rusted events will be trusted by both *T*rusted and *U*ntrusted executions, so will run in both (S, U) and (S, T) executions. Finally, every execution has enough privilege to see *P*ublic events, and every execution trusts *T*rusted events, so an event labeled (P, T) will run in every execution. The lookup semantics ($\Sigma, E \rightsquigarrow ks$) looks up the event handlers in Σ and constructs a configuration for each execution in E, resulting in ks.

The output rules run the event handlers one at a time. When an event handler is running, the configuration at the top of the stack is in producer state, $\mathsf{producer}(\kappa)$. Rule OUT handles outputs produced by the event handler. An execution performs outputs to channels only if the label on the channel matches the execution context, i.e., $\mathcal{P}(ch(v)) = pc$. Otherwise, the output is suppressed. Rule OUT-SILENT handles steps which do not produce outputs. When the event handler finishes running, the configuration at the top of the stack is in consumer state, $\mathsf{consumer}(\kappa)$, and rule OUT-NEXT pops the configuration off the stack to run the next event handler. The execution state is managed by the mid-level semantics, described next.

Example Recall our example of leaks *within* an execution from Section 4.2.3. We assume the security policy is that click events are considered secret and trusted, $\mathcal{P}(_.\mathsf{Click}(_)) = (S, T)$[2] and page load events are public and trusted, $\mathcal{P}(_.\mathsf{load}(_)) = (P, T)$. The user is interacting with the (S, U) copy of the page and the attacker who serves ads from *ad.com* is listening on (P, U) channels.

Initially, before any events have been triggered, we assume that the SME state is well-formed, meaning that source of the code (l_i) loaded to each execution (l'_i) is trusted ($l_i \sqsubseteq l'_i$). The attacker-controlled code from *ad.com* only appears in Untrusted executions, while the code from Trusted *news.com* will appear in all the executions. For our example, we also assume that *ad.com* registers an onLoad^U function to add the "Click me!" button (from Figure 4.2), and *news.com* registers onLoad^T to add the "Agree" button.

Then, the initial SME configuration is $K_0 = \Sigma_0; ks_0$ where ks_0 runs *body*.load for each execution (ks_0 will be described in more detail in the next section) in the following SME state:

$$\begin{aligned}
\Sigma_0 = \quad & (S,U) \mapsto \quad body \mapsto (_, \mathsf{load} \mapsto \{\mathsf{onLoad}^U, \mathsf{onLoad}^T\}), \\
& (P,U) \mapsto \quad body \mapsto (_, \mathsf{load} \mapsto \{\mathsf{onLoad}^U, \mathsf{onLoad}^T\}), \\
& (S,T) \mapsto \quad body \mapsto (_, \mathsf{load} \mapsto \{\mathsf{onLoad}^T\}), \\
& (P,T) \mapsto \quad body \mapsto (_, \mathsf{load} \mapsto \{\mathsf{onLoad}^T\}),
\end{aligned}$$

Next, the (S, U) execution runs the onLoad^U event handler. Since the action performed by this step of the computation is not an output to a channel ($\alpha \neq ch(v)$) rule Out-Silent makes a step from K_0 to some new configuration K_1: $K_0 \overset{(\bullet,(S,U))}{\Longrightarrow} K_1$. The new configuration K_1 now has a new button in the (S, U) copy of the store and the other copies remain unchanged:

$$\begin{aligned}
\Sigma_1 = \quad & (S,U) \mapsto \quad body \mapsto (...), b_{\mathsf{Agree}} \mapsto (\text{“}\mathit{Click\ me!}\text{”}, \cdot) \\
& (P,U) \mapsto \quad body \mapsto (...) \\
& (S,T) \mapsto \quad body \mapsto (...) \\
& (P,T) \mapsto \quad body \mapsto (...)
\end{aligned}$$

The same process will repeat to add the "Click me!" button to the (P, U) store and the "Agree" button to the other executions. Now that the event handlers have finished running, rule Out-Next pops the event handler from *ks* and the system waits for user input.

Suppose the attacker also installed an event handler in the (S, U) and (P, U) executions which directly sends them the user's account preferences. Since they are listening on a (P, U) channel, the rule out would suppress the output from the (S, U) execution which knows the real preferences (since $\mathcal{P}(ch) \neq (S, U)$). The same rule would allow the output from the (P, U) execution, which would not have access to the real preferences (it would instead output a default value dv).

[2] Not to be confused with the isTrusted property distinguishing events which come from a user from events which were generated by an event handler (see [96]).

$$\boxed{\Sigma, \kappa \xrightarrow{\alpha}_{pc} \Sigma', ks}$$

$$\frac{}{\Sigma, \sigma, \mathsf{skip}, P, \cdot \xrightarrow{\bullet}_{pc} \Sigma, ((\sigma, \mathsf{skip}, C, \cdot), pc)}\ \textsc{PtoC}$$

$$\frac{E = (id.\mathsf{Ev}(v), pc) :: E' \qquad \Sigma, E \rightsquigarrow ks}{\Sigma, \sigma, \mathsf{skip}, P, E \xrightarrow{\bullet}_{pc} \Sigma, ((\sigma, \mathsf{skip}, C, \cdot), pc) :: ks}\ \textsc{PtoLC}$$

$$\frac{\Sigma, \sigma, c \xrightarrow{\alpha}_{pc} \Sigma', \sigma', c', E'}{\Sigma, \sigma, c, P, E \xrightarrow{\alpha}_{pc} \Sigma', ((\sigma', c', P, (E, E')), pc)}\ \mathrm{P}$$

Figure 4.5: Mid-level rules for processing the event queue

4.3.2 Execution State and EH Queue

The syntax for the mid-level semantics is shown below.

$$\begin{array}{llll} \textit{Single config:} & \kappa & ::= & \sigma^v, c, s, E \\ \textit{Execution state:} & s & ::= & P \mid C \end{array}$$

A single configuration κ is a snapshot of one execution, including the local variables σ^v (which are only accessible to the event handler currently running), the current command c being executed, the execution state s of the event handler, and the event queue E. The execution state is either P for producer (meaning an event handler is running) or C for consumer (meaning the event handlers have finished and the execution is ready to process a new event). Here, the event queue, E, is a list of the events triggered by other event handlers. The events will run in the same execution, so the pc on each event in the queue will match the current execution context.

The semantics for managing the event handler queue and execution state are shown in Figure 4.5. Rule PtoC handles the case where an event handler has finished running ($c = \mathsf{skip}$) and no other event handlers have been triggered ($E = \cdot$). In this case, the execution state is changed to C for consumer state. On the other hand, if an event handler has triggered other event handlers to run ($E \neq \cdot$), rule PtoLC will additionally look up the event handlers in E and return these event handlers in ks. Finally, rule P runs an event handler using the event handler semantics, described below.

Example The top-level I/O rules use the execution state to decide whether they should continue running the event handler (rules Out and Out-Silent) or pop the event handler off ks to run the next event handler, if one exists (rule Out-Next), or wait for another input (rule In) if one does not. From the leaks *between* executions example in Section 4.2.3, before the user presses a key on their keyboard or clicks the button,

$$\boxed{\Sigma, \sigma, c \xrightarrow{\alpha}_{pc} \Sigma', \sigma', c', E}$$

$$\frac{}{\Sigma, \sigma, \mathsf{skip}; c \xrightarrow{\bullet}_{pc} \Sigma, \sigma, c, \cdot} \text{ SKIP} \qquad \frac{\Sigma, \sigma, c_1 \xrightarrow{\alpha}_{pc} \Sigma', \sigma', c_1', E}{\Sigma, \sigma, c_1; c_2 \xrightarrow{\alpha}_{pc} \Sigma', \sigma', c_1'; c_2, E} \text{ SEQ}$$

$$\frac{[\![e]\!]^{pc}_{\sigma,\Sigma} = v \qquad \Sigma(pc) = (\sigma^g, _) \qquad x \notin \sigma^g}{\Sigma, \sigma, x := e \xrightarrow{\bullet}_{pc} \Sigma, \sigma[x \mapsto v], \mathsf{skip}, \cdot} \text{ ASSIGN-L}$$

$$\frac{[\![e]\!]^{pc}_{\sigma,\Sigma} = v \qquad \Sigma(pc) = (\sigma^g, \sigma^{EH}) \qquad x \in \sigma^g \qquad \Sigma' = \Sigma[pc \mapsto (\sigma^g[x \mapsto v], \sigma^{EH})]}{\Sigma, \sigma, x := e \xrightarrow{\bullet}_{pc} \Sigma', \sigma, \mathsf{skip}, \cdot} \text{ ASSIGN-G}$$

$$\frac{[\![e]\!]^{pc}_{\sigma,\Sigma} = v \qquad \Sigma(pc) = (\sigma^g, \sigma^{EH}) \qquad \sigma^{EH}(id) = (_, M) \qquad \Sigma' = \Sigma[pc \mapsto (\sigma^g, \sigma^{EH}[id \mapsto (v, M)])]}{\Sigma, \sigma, id := e \xrightarrow{\bullet}_{pc} \Sigma', \sigma, \mathsf{skip}, \cdot} \text{ UPDATE}$$

$$\frac{[\![e]\!]^{pc}_{\sigma,\Sigma} = b \qquad c = c_1 \text{ if } b = \mathsf{true} \qquad c = c_2 \text{ if } b = \mathsf{false}}{\Sigma, \sigma, \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \xrightarrow{\bullet}_{pc} \Sigma, \sigma, c, \cdot} \text{ IF}$$

$$\frac{[\![e]\!]^{pc}_{\sigma,\Sigma} = b \qquad c' = c; \mathsf{while}\ e\ \mathsf{do}\ c \text{ if } b = \mathsf{true} \qquad c' = \mathsf{skip} \text{ if } b = \mathsf{false}}{\Sigma, \sigma, \mathsf{while}\ e\ \mathsf{do}\ c \xrightarrow{\bullet}_{pc} \Sigma, \sigma, c', \cdot} \text{ WHILE}$$

Figure 4.6: Rules for running event handers

the system is in Consumer state, waiting for user interaction ($ks = \cdot$). When the user clicks the *secret* button, the input rule In looks up the event handler for *secret*.Click() and the rule LOOKUP sets the execution state to *P*roducer. The rule P in the mid-level semantics run the event handler to completion and then PtoC switches the execution state back to Consumer state.

4.3.3 Individual Event Handlers

The syntax relevant to running individual event handlers is summarized below.

$$\begin{array}{lcl} \textit{Expression: } e & ::= & x \mid v \mid id \mid \mathsf{uop}\, e \mid e_1\, \mathsf{bop}\, e_2 \\ \textit{Command: } c & ::= & \mathsf{skip} \mid c_1; c_2 \mid x := e \mid id := e \mid \mathsf{while}\ e\ \mathsf{do}\ c \mid \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \\ & \mid & \mathsf{output}\ ch\ e \mid \mathsf{new}(id, e) \mid \mathsf{addEh}(id, eh) \mid \mathsf{trigger}\ id.\mathsf{Ev}(e) \end{array}$$

Expressions include variables, values (integers and booleans), page element identifiers, *id*, unary, and binary operators. Commands are mostly standard and include outputs to channels output $ch\ e$ and dynamic behaviors. $\mathsf{new}(id, e)$ adds a new page element with attribute e, $\mathsf{addEh}(id, eh)$ registers a new event handler eh to a page element given by id, and trigger $id.\mathsf{Ev}(e)$ triggers an event handler for event Ev associated with page element id, passing argument e.

$$\boxed{\Sigma, \sigma, c \xrightarrow{\alpha}_{pc} \Sigma', \sigma', c', E}$$

$$\frac{[\![e]\!]^{pc}_{\sigma,\Sigma} = v}{\Sigma, \sigma, \mathsf{output}\ ch\ e \xrightarrow{ch(v)}_{pc} \Sigma, \sigma, \mathsf{skip}, \cdot}\ \textsc{output} \qquad \frac{[\![e]\!]^{pc}_{\sigma,\Sigma} = v \qquad E = (id.\mathsf{Ev}(v), pc)}{\Sigma, \sigma, \mathsf{trigger}\ id.\mathsf{Ev}(e) \xrightarrow{\bullet}_{pc} \Sigma, \sigma, \mathsf{skip}, E}\ \textsc{trigger}$$

$$\frac{[\![e]\!]^{pc}_{\sigma,\Sigma} = v \qquad \Sigma(pc) = (\sigma^g, \sigma^{EH}) \qquad id \notin \sigma^{EH} \qquad \Sigma' = \Sigma[pc \mapsto (\sigma^g, \sigma^{EH}[id \mapsto (v, \cdot)])]}{\Sigma, \sigma, \mathsf{new}(id, e) \xrightarrow{\mathsf{new}(id)}_{pc} \Sigma', \sigma, \mathsf{skip}, \cdot}\ \textsc{new}$$

$$\frac{\begin{array}{c}\Sigma(pc) = (\sigma^g, \sigma^{EH}) \\ \sigma^{EH}(id) = (v, M) \qquad M' = M(\mathsf{Ev}) \cup eh \qquad \Sigma' = \Sigma[pc \mapsto (\sigma^g, \sigma^{EH}[id \mapsto (v, M')])]\end{array}}{\Sigma, \sigma, \mathsf{addEh}(id, \mathsf{Ev}, eh) \xrightarrow{\mathsf{new}(id,eh)}_{pc} \Sigma', \sigma, \mathsf{skip}, \cdot}\ \textsc{add-eh}$$

Figure 4.7: Additional rules for running event handers

We show the event handler operational semantics in Figures 4.6 and 4.7. The rules in Figure 4.6 are mostly standard. We evaluate expressions with $[\![e]\!]^{pc}_{\sigma,\Sigma}$ where pc tells us which copy of the shared storage to access in Σ and σ is the store local to the current event handler. Rule ASSIGN-L handles the case where we are updating the event handler store, while ASSIGN-G handles the case where we are updating the (persistent) shared storage. We assume that the set of shared variables are static throughout the execution so we can determine which store to modify by checking if the variable is in the shared store. Rule UPDATE updates attributes of the DOM.

Candidate outputs are produced by rule OUTPUT, and the I/O semantics decide whether the output is permitted or not. The other rules are for handling dynamic elements, including triggering event handlers (rule TRIGGER), generating new page elements (rule NEW), and registering a new event handler (rule ADD-EH). In each of these rules, we interact with the copy of the global storage which matches the current execution context. Event handlers run in the same context they were triggered in, denoted by pc. New page elements must have a unique identifier, $id \notin \sigma^{EH}$, and are initialized with the given attribute and no event handlers, $M = \cdot$. When registering a new event handler, the existing event handlers associated with the event are looked up in the event handler map, $M(\mathsf{Ev})$. The event handler map is updated to include the original event handlers plus the new one, $M[\mathsf{Ev} \mapsto M(\mathsf{Ev}) \cup eh]$.

4.4 Declassification

The semantics presented in the previous section enforce a strict noninterference, where information from secret events can never flow to more public outputs. We extend the syntax and semantics from Section 4.3 to include declassification.

$\boxed{\mathcal{P}, \mathcal{D} \vdash K \overset{(\alpha,pc)}{\Longrightarrow} K'}$

$$\frac{\mathcal{P}(id.\mathsf{Ev}(v)) = pc' \qquad E = ((id.\mathsf{Ev}(v), pc'') \mid pc'' \in \mathcal{L} \text{ s.t. } pc \sqcup pc' \sqsubseteq pc'') \\ (\mathcal{R}', E_d) = \mathsf{declassify}(\mathcal{D}, \mathcal{R}, \Sigma, (id.\mathsf{Ev}(v), pc), pc') \qquad \Sigma, E :: E_d \rightsquigarrow ks}{\mathcal{P}, \mathcal{D} \vdash \mathcal{R}; \Sigma; \cdot \overset{(id.\mathsf{Ev}(v),pc)}{\Longrightarrow} \mathcal{R}'; \Sigma; ks} \text{ IN}$$

$\boxed{\mathsf{declassify}(\mathcal{D}, \mathcal{R}, \Sigma, (id.\mathsf{Ev}(v), pc), pc') = (\mathcal{R}', E)}$

$$\frac{\mathcal{D}((id.\mathsf{Ev}(v), pc), pc', \rho) = (\rho', v_d, E_d) \qquad d' = \mathsf{update}(d, v_d)}{\mathsf{declassify}(\mathcal{D}, (\rho, d), \Sigma, (id.\mathsf{Ev}(v), pc), pc') = ((\rho', d'), E_d)} \text{ DECLASSIFY}$$

Figure 4.8: Updated input rule for declassification. We highlight the noteworthy changes to the existing input rule using red text.

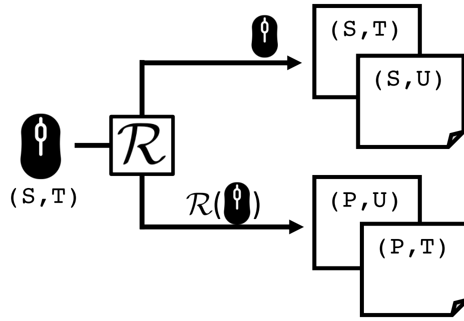


Figure 4.9: SME model for enforcing confidentiality and integrity information flow policies, including declassification. A secret and trusted event (mouse click) would be shared with (S, T) and (S, U) executions, and would only be shared with (P, U) and (P, T) executions if permitted by the declassification policy.

4.4.1 Declassification Syntax and Semantics

We use *stateful* declassification, similar to prior work [91] and Chapter 3. A stateful policy is one that may involve the system state when deciding whether to declassify. Here, we describe the syntax for declassification, which is shown below.

Declassification policy:	$\mathcal{D}$		
Declassification module:	$\mathcal{R}$	$::=$	(ρ, d)
Declassification state:	ρ	$::=$	$\cdot \mid \rho, (id_1.\mathsf{Ev}_1, n_1)$
Declassification channel:	d	$::=$	$(\iota_1, v_1), \cdots, (\iota_n, v_n)$

The declassification policy is given by $\mathcal{D}$. Given an event and the current state, as well as information from the security policy, $\mathcal{P}$, $\mathcal{D}$ updates the current state and decides whether the event should be declassified. The declassification module $\mathcal{R}$ keeps track of the current state for making decisions about declassification as well as channels for event handlers to access released values. A declassification state

ρ keeps track of relevant state conditions, for example, whether a particular button has been clicked, the number of key presses, or the aggregate location of clicks. Here, for simplicity, ρ simply counts how many times an event has been seen. The declassification channel d associates locations ι (such as a line number in the code) with the released value accessible by that location.

The relationship between the declassification module and SME executions is shown in Figure 4.9. Incoming events are shared with the executions with enough privilege who trust the input, like before. For example, if the security policy says that mouse clicks are Secret and Trusted, then the (S, T) and (S, U) executions receive the event as-is. Now, events may also be shared with executions with less privilege, as determined by the declassification policy. For the example in Figure 4.9, this means that (P, U) and (P, T) receive the event released by the declassification module (if any). Note that the declassified events may differ between executions, for instance, a policy might release the coordinates of the mouse click to some executions and replace the coordinates with a default value for other executions.

Updating the semantics from Section 4.3 is straightforward. The judgement for the I/O semantics is updated to include $\mathcal{D}$ and $\mathcal{R}$:

$$\mathcal{P}, \mathcal{D} \vdash \mathcal{R}; \Sigma; ks \stackrel{\alpha_l}{\Longrightarrow} \mathcal{R}'; \Sigma'; ks'$$

A declassification function ($\mathsf{declassify}$), shown in Figure 4.8 is added to the input rule. It uses the declassification policy $\mathcal{D}$ and the label on the event pc' (which comes from the security policy) to determine whether the new event $(id.\mathsf{Ev}(v), pc)$ should be released to run event handlers in additional execution contexts E_d, whether the system state ρ should be updated, and what values should be updated on the declassification channel d (if any). Note that declassification will only change the integrity of an event to ensure the executions which trust the event receive the declassified event; it will never make the event *more* trusted (more on endorsement in the next Section).

Example Recall the example of leaks *within* an execution from Section 4.2.3, where the security policy says that clicks are (S, T), and the declassification policy says that the user's preferences may be declassified from S to P when b_{Agree} is clicked.

When the user clicks b_{Agree} in the (S, U) execution, IN will share the event with all the executions with enough privilege and who trust the user (just (S, U)), but we also use $\mathsf{declassify}(\mathcal{D}, \mathcal{R}, \Sigma, (b_{\mathsf{Agree}}.\mathsf{Click}(\ ),$ $(S, U)), (S, T))$ to determine whether the event should be declassified to additional executions:

$$\mathcal{D}((b_{\mathsf{Agree}}.\mathsf{Click}(\ ), (S, U)), (S, T), (b_{\mathsf{Agree}}.\mathsf{click}, n)) = ((b_{\mathsf{Agree}}.\mathsf{Click}, n + 1), loc, \cdot)$$

This indicates that the state ρ has been updated to reflect that one more click has been seen (n becomes $n + 1$), the user's location should be released on the declassification channel (loc), and the click event should not be released to any additional executions.

For the example of leaks *between* executions, the security policy says that button clicks and keypresses are both (S, T), but now, the declassification policy says that button clicks may be released from S to P. When the user clicks b_{secret}, rule IN runs the event as-is in the (S, U) execution and declassifies the event as follows:

$$\mathcal{D}((b_{\mathsf{secret}}.\mathsf{click}(\ ), (S, T)), (S, T), (b_{\mathsf{secret}}.\mathsf{click}, m)) = ((b_{\mathsf{secret}}.\mathsf{click}, m + 1), \mathsf{none}, (b_{\mathsf{secret}}.\mathsf{click}(\ ), (P, U)))$$

Here, ρ is updated to reflect the click, nothing is updated on the declassification channel (none), and the click event is released to the P executions who trust the event. That is, the event is released to all executions with label l_i s.t. l_i trusts the event l'_i (determined by the security policy) and the source of the event l''_i (formally, $l'_i \sqcup l''_i \sqsubset l_i$). Here, this is just $(b_{\mathsf{secret}}.\mathsf{Click}(\ ), (P, U))$. The result is that the onClick event handler will run in both the (S, U) and (P, U) executions. Rule OUT will suppress the output from the (S, U) execution, but the same rule will permit the output from the (P, U) execution, which is guaranteed to have a matching button to capture the event.

4.4.2 Robust Declassification

In the presence of an active attacker who may have control over some of the code, we need to ensure that they do not control what/whether data is declassified [101]. As we show in the example in Section 4.2.3, SME ensures that each execution only receives data they have the privilege to access (and is only influenced by code they trust). Now, for the declassifications to be robust against attacker influence, we also need to ensure that the source of the event l_i trusts the code l'_i on the *same* execution they are interacting with ($l'_i \sqsubseteq l_i$). Additionally, we need to check that the source of the event trusts the code which added the page element in the *other* execution receiving the declassified event. From the example in the previous section: By declassifying an event from the attacker-controlled button (in the former case), or letting the attacker add elements to capture declassified events (in the latter case), more information is leaked than the declassification policy intends.

In this section, we compose taint tracking with the SME semantics presented in the previous section to also keep track of the source of the page elements in each execution. First, we modify the event handler storage σ^{EH} so that page elements and event handlers have labels $l \in \mathcal{L}_i$ indicating the trustworthiness of their source:

$$\begin{array}{llll} \textit{EH state: } \sigma^{EH} & ::= & \cdot \mid \sigma^{EH}, id \mapsto (v, M)^{l} \\ \textit{EH map: } \ M & ::= & \cdot \mid \mathsf{Ev} \mapsto \{eh_1^{l_1}, ..., eh_n^{l_n}\} \end{array}$$

The labels on the page elements are independent of the labels on the event handlers. A Trusted source may add a page element, which an Untrusted source later registers an event handler to, or vice versa. We

do assume that the source of the code l_i is trusted by the execution pc, i.e., $l_i \sqsubseteq pc \downarrow^i$, where $pc \downarrow^i$ is the integrity label in pc. This means that Untrusted sources cannot add code to Trusted executions. Therefore, the labels on page elements/event handlers l_i are related to the execution context pc in the same way.

The input rules prevent leaks *within* executions by using the labels in σ^{EH} to decide whether to proceed with a declassification. In order to declassify, the source of the event must trust the source of the page element. We use the shorthand $\mathsf{labelOf}(\sigma^{EH}(id))$ to represent the label on the element identified by id in σ^{EH}. Then, an event from a user at security level pc associated with a page element given by id in σ^{EH} is allowed to be declassified when the following holds: $\mathsf{labelOf}(\sigma^{EH}(id)) \sqsubseteq pc \downarrow^i$. If this condition holds, rule IN-RELEASE attempts to declassify the event. Otherwise, rule IN-NO-RELEASE only runs the event in the executions which have enough privilege to see the event and who trust the user.

$$\boxed{\mathcal{P}, \mathcal{D} \vdash K \stackrel{(\alpha, pc)}{\Longrightarrow} K'}$$

$$\frac{\begin{array}{c}\mathcal{P}(id.\mathsf{Ev}(v)) = pc' \qquad \Sigma(pc) = (_, \sigma^{EH}) \\ \mathsf{labelOf}(\sigma^{EH}(id)) \sqsubseteq pc \downarrow^i \qquad E = ((id.\mathsf{Ev}(v), pc'') \mid pc'' \in \mathcal{L} \text{ s.t. } pc \sqcup pc' \sqsubseteq pc'') \\ (\mathcal{R}', E_d) = \mathsf{declassify}(\mathcal{D}, \mathcal{R}, \Sigma, (id.\mathsf{Ev}(v), pc), pc') \qquad \Sigma, E \rightsquigarrow ks \qquad pc \downarrow^i, \mathsf{r} \vdash \Sigma, E_d \rightsquigarrow ks_d\end{array}}{\mathcal{P}, \mathcal{D} \vdash \mathcal{R}; \Sigma; \cdot \stackrel{(id.\mathsf{Ev}(v), pc)}{\Longrightarrow} \mathcal{R}'; \Sigma; ks :: ks_d} \textsc{In-Release}$$

$$\frac{\begin{array}{c}\mathcal{P}(id.\mathsf{Ev}(v)) = pc' \qquad \Sigma(pc) = (_, \sigma^{EH}) \\ \mathsf{labelOf}(\sigma^{EH}(id)) \not\sqsubseteq pc \downarrow^i \qquad E = ((id.\mathsf{Ev}(v), pc'') \mid pc'' \in \mathcal{L} \text{ s.t. } pc \sqcup pc' \sqsubseteq pc'') \qquad \Sigma, E \rightsquigarrow ks\end{array}}{\mathcal{P}, \mathcal{D} \vdash \mathcal{R}; \Sigma; \cdot \stackrel{(id.\mathsf{Ev}(v), pc)}{\Longrightarrow} \mathcal{R}'; \Sigma; ks} \textsc{In-NoRelease}$$

Figure 4.10: Modified input rules for robust declassification. Noteworthy changes are shown in red text.

We use the declassification function from Section 4.4.1 to prevent leaks *between* executions. The updated declassification rules are in Figures 4.10 and 4.11. In addition to looking up the declassified event(s) and the execution(s) they will run in, robust throws out executions where the source of the event does not trust the source of the page element. Rule ROBUST handles the case where the source of the code is trusted (the event is sent to the execution), and rule NOT-ROBUST handles the case where it is not (the execution does not receive the event). Then, the lookup semantics (judgement $l, \mathsf{r} \vdash \Sigma, E \rightsquigarrow ks$) ensure only trusted event handlers run. We define $(eh, l') \downarrow_l$ as eh when $l' \sqsubseteq l$ and $\cdot$ otherwise. When there is at least one event handler the user trusts ($\Sigma(pc)(id.\mathsf{Ev}(v)) \downarrow_l = c$), rule LOOKUP-R adds the trusted event handlers to ks and attaches a label $\Sigma(pc) \sqcup \Sigma(pc)(id.\mathsf{Ev}(v))$ reflecting the source of the code. When there are no trusted event handlers ($\Sigma(pc)(id.\mathsf{Ev}(v)) \downarrow_l = \cdot$), rule LOOKUP-NOTR moves to the next execution receiving the declassified event. The rules adding a new page element (NEW) or event handler (ADD-EH) are responsible for assigning the labels in the event handler store, where l_{src} is the label from rule LOOKUP-R.

$\boxed{\mathsf{declassify}(\mathcal{D}, \mathcal{R}, \Sigma, (id.\mathsf{Ev}(v), pc), pc_{\mathsf{Ev}}) = (\mathcal{R}', E)}$

$$\frac{\mathcal{D}((id.\mathsf{Ev}(v), pc), pc', \rho) = (\rho', v_d, E_d) \qquad d' = \mathsf{update}(d, v_d) \qquad E = \mathsf{robust}(\Sigma, E_d, pc \downarrow^i)}{\mathsf{declassify}(\mathcal{D}, (\rho, d), \Sigma, (id.\mathsf{Ev}(v), pc), pc') = ((\rho', d'), E)} \text{ DECLASSIFY}$$

$\boxed{\mathsf{robust}(\Sigma, E, pc_{\mathsf{Ev}}) = E'}$

$$\frac{\Sigma(pc) = (_, \sigma^{EH}) \qquad \mathsf{labelOf}(\sigma^{EH}(id)) \sqsubseteq l}{\mathsf{robust}(\Sigma, ((id.\mathsf{Ev}(v), pc) :: E), l) = (id.\mathsf{Ev}(v), pc) :: \mathsf{robust}(\Sigma, E, l)} \text{ ROBUST}$$

$$\frac{\Sigma(pc) = (_, \sigma^{EH}) \qquad \mathsf{labelOf}(\sigma^{EH}(id)) \not\sqsubseteq l}{\mathsf{robust}(\Sigma, ((id.\mathsf{Ev}(v), pc) :: E), l) = \mathsf{robust}(\Sigma, E, l)} \text{ NOTR} \qquad \frac{}{\mathsf{robust}(\Sigma, \cdot), l) = \cdot} \text{ EMPR}$$

$\boxed{pc, \mathsf{r} \vdash \Sigma, E \rightsquigarrow ks}$

$$\frac{\begin{array}{c}\Sigma(pc) = (_, \sigma^{EH}) \qquad \sigma^{EH}(id.\mathsf{Ev}(v)) \downarrow_l = c \\ pc' = \mathsf{labelOf}(\sigma^{EH}(id)) \sqcup \mathsf{labelOf}((\sigma^{EH}(id))(\mathsf{Ev})) \qquad \kappa = \cdot, c, P, \cdot \qquad l, \mathsf{r} \vdash \Sigma, E \rightsquigarrow ks\end{array}}{l, \mathsf{r} \vdash \Sigma, (id.\mathsf{Ev}(v), pc) :: E \rightsquigarrow (\kappa, pc', pc) :: ks} \text{ LOOKUP-R}$$

$$\frac{\begin{array}{c}\Sigma(pc) = (_, \sigma^{EH}) \qquad \sigma^{EH}(id.\mathsf{Ev}(v)) \downarrow_l = \cdot \\ l, \mathsf{r} \vdash \Sigma, E \rightsquigarrow ks\end{array}}{l, \mathsf{r} \vdash \Sigma, (id.\mathsf{Ev}(v), pc) :: E \rightsquigarrow ks} \text{ LOOKUP-NOTR} \qquad \frac{}{l, \mathsf{r} \vdash \Sigma, \cdot \rightsquigarrow \cdot} \text{ LOOKUP-EMPR}$$

$\boxed{l_{src}, d \vdash \Sigma, \sigma, c \xrightarrow{\alpha}_{pc} \Sigma', \sigma', c', E}$

$$\frac{[\![e]\!]^{pc}_{\sigma,\Sigma} = v \qquad \Sigma(pc) = (\sigma^g, \sigma^{EH}) \qquad id \notin \sigma^{EH} \qquad \Sigma' = \Sigma[pc \mapsto (\sigma^g, \sigma^{EH}[id \mapsto (v, \cdot)^{l_{src}}])]}{l_{src}, d \vdash \Sigma, \sigma, \mathsf{new}(id, e) \xrightarrow{\bullet}_{pc} \Sigma', \sigma, \mathsf{skip}, \cdot} \text{ NEW}$$

$$\frac{\begin{array}{c}\Sigma(pc) = (\sigma^g, \sigma^{EH}) \\ \sigma^{EH}(id) = (v, M)^{l_{id}} \qquad M' = M[\mathsf{Ev} \mapsto M(\mathsf{Ev}) \cup eh^{l_{src}}] \qquad \Sigma' = \Sigma[pc \mapsto (\sigma^g, \sigma^{EH}[id \mapsto (v, M')^{l_{id}}])]\end{array}}{l_{src}, d \vdash \Sigma, \sigma, \mathsf{addEh}(id, \mathsf{Ev}, eh) \xrightarrow{\bullet}_{pc} \Sigma', \sigma, \mathsf{skip}, \cdot} \text{ ADD-EH}$$

Figure 4.11: Additional rule changes for robust declassification. Noteworthy changes are shown in red.

Example Recall the examples from Section 4.2.3. Like before, we assume that the initial SME state is well-formed, meaning that the page elements and event handlers are trusted by the execution context they appear in. With our new labels, we can make this more precise: execution (l_c, l_i) should trust the page elements and their event handlers, from source l_i', $l_i' \sqsubseteq l_i$.

For our example of leaks *within* an execution, there are three event handlers. onLoad^U is added by the attacker via *ad.com*, who is Untrusted, and onLoad^T is added by the host via *news.com*, who is Trusted. These event handlers are associated with the *body* of the page, which we treat as Trusted. Recall that we assume that the source of the code is trusted by the execution, meaning code from *ad.com* only runs in the

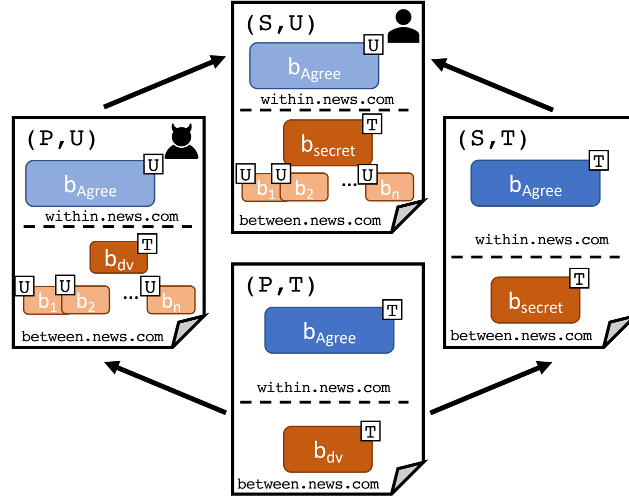


Figure 4.12: Insecure example from Section 4.2 with robustness checks. The labels tell us the trustworthiness of the source of the page elements and event handlers, depicted here as small white labels on each page element.

Untrusted executions and code from *news.com* runs in both Untrusted and Trusted executions. Then, the initial SME state with integrity labels is:

$$\begin{aligned}\Sigma_0 = \quad & (S,U) \mapsto \quad body \mapsto (_, \mathsf{load} \mapsto \{\mathsf{onLoad}^U, \mathsf{onLoad}^T\})^T \\ & (P,U) \mapsto \quad body \mapsto (_, \mathsf{load} \mapsto \{\mathsf{onLoad}^U, \mathsf{onLoad}^T\})^T \\ & (S,T) \mapsto \quad body \mapsto (_, \mathsf{load} \mapsto \{\mathsf{onLoad}^T\})^T \\ & (P,T) \mapsto \quad body \mapsto (_, \mathsf{load} \mapsto \{\mathsf{onLoad}^T\})^T\end{aligned}$$

Now, when the onLoad^U event handler runs, the execution knows the code came from an Untrusted source because of the label U. When the event handler adds the "Click me!" button, rule NEW uses the label on the page element T and event handler U to determine the trustworthiness of the new button $T \sqcup U = U$. The state after adding the "Click me!" button to the (S,U) execution is:

$$\begin{aligned}\Sigma_1 = \quad & (S,U) \mapsto \quad body \mapsto (...)^T, b_{\mathsf{Agree}} \mapsto (\text{"}Click\ me!\text{"}, \{\ \})^U \\ & (P,U) \mapsto \quad body \mapsto (...)^T \\ & (S,T) \mapsto \quad body \mapsto (...)^T \\ & (P,T) \mapsto \quad body \mapsto (...)^T\end{aligned}$$

Figure 4.12 shows the resulting page after all the buttons are loaded, including their labels. When the user clicks the "Click me!" button on the (S,U) copy of the page, the input rules will use the label on the button to determine if the declassification is allowed. The user is treated as a Trusted source of events, so because $U \not\sqsubseteq T$, rule IN-NO-RELEASE prevents the event from being declassified and the attacker does not learn the user's location.

For our example of leaks *between* executions, the host installs an onKeypress event handler to some field which adds a different button to the page depending on what the user types, and the attacker adds all possible buttons to the page. After the user presses a key, the SME store has one Trusted button per execution, and several Untrusted buttons in the Untrusted executions:

$$\begin{aligned}\Sigma_0 = \quad & (S,U) \mapsto b_{\mathsf{secret}} \mapsto (...)^T, b_1 \mapsto (...)^U, ..., b_n \mapsto (...)^U \\ & (P,U) \mapsto b_{\mathsf{dv}} \mapsto (...)^T, b_1 \mapsto (...)^U, ..., b_n \mapsto (...)^U \\ & (S,T) \mapsto b_{\mathsf{secret}} \mapsto (...)^T \\ & (P,T) \mapsto b_{\mathsf{dv}} \mapsto (...)^T\end{aligned}$$

When the user clicks the b_{secret} button on the (S,U) copy of the page, rule IN-RELEASE attempts to declassify the event to the (P,U) execution since the button is Trusted. Next, the robust rules use the labels on the button capturing the event to determine if the (P,U) execution should receive the declassified event. In this case, the button b_i capturing the event was added by the attacker. Since $U \not\sqsubseteq T$, rule NOT-ROBUST skips the (P,U) execution and the attacker does not learn which key the user pressed.

4.5 Endorsement

Declassification allows us to make exceptions to strict confidentiality policies. The dual condition for integrity is called *endorsement*. Declassification and endorsement are more generally called *downgrading*. Prior work shows that, like declassification, attackers can use endorsement to launder secrets unless the endorsement is *transparent* [31]. The focus of our work is robust declassification, but in this section we briefly describe how endorsement can be added to our model. We define transparent endorsement for our reactive system and, finally, show that we can use similar techniques from Section 4.4 to make endorsements transparent.

4.5.1 Endorsement Syntax and Semantics

Similar to declassification, we define a *stateful* endorsement. For example, a stateful endorsement policy might be that before an event can be endorsed, some software attestation process needs to happen for the event handler generating the event. For instance, every n times we run the event handler, we need to compare the hash of the event handler against what is stored in the browser to make sure we are running the correct code. The number of times we have run the event handler is maintained by the endorsement state. The syntax for both stateful declassification and endorsement is shown below.

$$\boxed{\mathcal{P},\mathcal{D} \vdash K \stackrel{(\alpha,pc)}{\Longrightarrow} K'}$$

$$\frac{\begin{array}{c}\mathcal{P}(id.\mathsf{Ev}(v)) = pc' \\ E = ((id.\mathsf{Ev}(v), pc'') \mid pc'' \in \mathcal{L} \text{ s.t. } pc \sqcup pc' \sqsubseteq pc'') \qquad \mathsf{declassify}_{\mathcal{D}}(\mathcal{R}, \Sigma, \alpha, pc') = (\mathcal{R}', E_d) \\ (\mathcal{S}', E_e) = \mathsf{endorse}(\mathcal{E}, \mathcal{S}, \Sigma, (id.\mathsf{Ev}(v), pc), pc') \qquad \Sigma, E :: E_d :: E_e \rightsquigarrow ks\end{array}}{\mathcal{P}, \mathcal{E} \vdash \mathcal{R}, \mathcal{S}; \Sigma; \cdot \stackrel{(id.\mathsf{Ev}(v),pc)}{\Longrightarrow} \mathcal{R}', \mathcal{S}'; \Sigma; ks} \text{ IN}$$

$$\boxed{\mathsf{endorse}(\mathcal{E}, \mathcal{S}, \Sigma, (id.\mathsf{Ev}(v), pc)) = (\mathcal{S}', E)}$$

$$\frac{\mathcal{E}((id.\mathsf{Ev}(v), pc), pc', \rho) = (\rho', v_e, E_e) \qquad d' = \mathsf{update}(d, v_e)}{\mathsf{endorse}(\mathcal{E}, (\rho, d), \Sigma, (id.\mathsf{Ev}(v), pc), pc') = ((\rho', d'), E_e)} \text{ ENDORSE}$$

Figure 4.13: Update input rule for endorsement. We highlight the noteworthy changes to the existing input rule using red text.

Declassification function:	$\mathcal{D}$		
Endorsement function:	$\mathcal{E}$		
Declassification module:	$\mathcal{R}$	$::=$	(ρ_d, d_d)
Endorsement module:	$\mathcal{S}$	$::=$	(ρ_e, d_e)
Downgrade state:	ρ	$::=$	$\cdot \mid \rho, (id_1.\mathsf{Ev}_1, n_1)$
Downgrade channels:	d	$::=$	$(\iota_1, v_1), \cdots, (\iota_n, v_n)$

The endorsement policy is given by $\mathcal{E}$. Given an event and the current state, as well as information from the security policy, $\mathcal{P}$, $\mathcal{E}$ updates the current state and decides whether the event should be endorsed. Note that endorsement will only change the confidentiality of an event to ensure the executions which have enough privilege to see the event receive the endorsed event; it will never make the event *more* public. The endorsement module $\mathcal{S}$ keeps track of the current state for making decisions about endorsement as well as channels for event handlers to access endorsed values. The endorsement state ρ counts how many times an event has been seen, and the endorsement channel d associates locations ι (such as a line number in the code) with the sanitized value accessible by that location.

The judgement for the I/O semantics is updated to include $\mathcal{E}$ and $\mathcal{S}$:

$$\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash \mathcal{R}, \mathcal{S}; \Sigma; ks \stackrel{\alpha_l}{\Longrightarrow} \mathcal{R}', \mathcal{S}'; \Sigma'; ks'$$

An endorsement function (endorse), shown in Figure 4.13 is added to the input rule. It uses the endorsement policy $\mathcal{E}$ to determine whether the new event should be endorsed to run event handlers in additional execution contexts E_e, whether the system state ρ should be updated, and what values should be updated on the endorsement channel d (if any).

4.5.2 Transparent Endorsement

The dual condition to robust declassification is transparent endorsement. Similar to the requirement that the source of the event trusts the code they are interacting with before declassification, we want to ensure that the source of the event has enough privilege to interact with the page element before endorsement. Without transparent endorsement, prior work demonstrates that the attacker can leverage endorsement to launder secrets or even fool a password checker [31]. For the endorsements to be transparent, we ensure that the source of the event l_c has enough privilege to interact with the code l'_c on the *same* execution ($l'_c \sqsubseteq l_c$). Additionally, we check that the source of the event has enough privilege to interact with the code in the *other* execution receiving the endorsed event. We modify the event handler storage σ^{EH} to track both confidentiality and integrity labels:

$$\begin{array}{lll} \textit{EH state: } \sigma^{EH} & ::= & \cdot \mid \sigma^{EH}, id \mapsto (v, M)^{pc} \\ \textit{EH map: } \ M & ::= & \cdot \mid \mathsf{Ev} \mapsto \{eh_1^{pc_1}, ..., eh_n^{pc_n}\} \end{array}$$

Using the same strategy as Section 4.4.2, we compose taint tracking with the SME semantics to also keep track of the secrecy level of the page elements in each execution. The updated input rules are shown in Figure 4.14. We use the integrity label on page elements $\mathsf{labelOf}(\sigma^{EH}(id)) \downarrow^i$ to decide whether the declassification would be robust *within* the execution (when the page element is trusted by the source of the event: $\mathsf{labelOf}(\sigma^{EH}(id)) \downarrow^i \sqsubseteq pc \downarrow^i$), and the confidentiality label $\mathsf{labelOf}(\sigma^{EH}(id)) \downarrow^c$ to decide whether the endorsement would be transparent *within* the execution (when the source of the event has enough privilege to see the page element: $\mathsf{labelOf}(\sigma^{EH}(id)) \downarrow^c \sqsubseteq pc \downarrow^c$). Rule In-Release handles the case where declassification is possible (but not endorsement), which is performed by declassify, like in the previous Section. Rule In-Sanitize handles the case where endorsement is possible (but not declassification), which is performed by endorse, and rule In handles the case where neither are possible. When both declassification and endorsement are possible, rule In-Downgrade uses downgrade to decide which additional executions should receive the event. Note that downgrade does not modify $\mathcal{R}$ or $\mathcal{S}$, it simply combines the results of $\mathcal{D}$ and $\mathcal{E}$.

4.6 Security

To prove that our SME semantics from Sections 4.3 and 4.4 are secure, in this section we present three security conditions. First, a knowledge-based progress-insensitive noninterference with declassification (Section 4.6.1) which ensures that the attacker's knowledge of the secret inputs is not refined as the system runs outside of what is declassified (and the fact that the system makes progress). Second, we describe a novel influence-based progress-insensitive noninterference (Section 4.6.2) which is the integrity

$$\boxed{\mathcal{P},\mathcal{D},\mathcal{E} \vdash K \overset{(\alpha,pc)}{\Longrightarrow} K'}$$

$$\frac{\begin{array}{c}\mathcal{P}(id.\mathsf{Ev}(v)) = pc' \qquad \Sigma(pc) = (_, \sigma^{EH}) \\ \mathsf{labelOf}(\sigma^{EH}(id)) \downarrow^i \not\sqsubseteq pc \downarrow^i \qquad \mathsf{labelOf}(\sigma^{EH}(id)) \downarrow^c \not\sqsubseteq pc \downarrow^c \\ E = ((id.\mathsf{Ev}(v), pc'') \mid pc'' \in \mathcal{L} \text{ s.t. } pc \sqcup pc' \sqsubseteq pc'') \qquad \Sigma, E \rightsquigarrow ks\end{array}}{\mathcal{P},\mathcal{D},\mathcal{E} \vdash \mathcal{R},\mathcal{S};\Sigma;\cdot \overset{(id.\mathsf{Ev}(v),pc)}{\Longrightarrow} \mathcal{R},\mathcal{S};\Sigma;ks} \text{ IN}$$

$$\frac{\begin{array}{c}\alpha = (id.\mathsf{Ev}(v), pc) \qquad \mathcal{P}(id.\mathsf{Ev}(v)) = pc' \qquad \Sigma(pc) = (_, \sigma^{EH}) \\ \mathsf{labelOf}(\sigma^{EH}(id)) \downarrow^i \sqsubseteq pc \downarrow^i \qquad \mathsf{labelOf}(\sigma^{EH}(id)) \downarrow^c \not\sqsubseteq pc \downarrow^c \\ E = ((id.\mathsf{Ev}(v), pc'') \mid pc'' \in \mathcal{L} \text{ s.t. } pc \sqcup pc' \sqsubseteq pc'') \qquad \Sigma, E \rightsquigarrow ks \\ \mathsf{declassify}(\mathcal{D},\mathcal{R},\Sigma,\alpha,pc') = (\mathcal{R}', E_d) \qquad pc, \mathsf{r} \vdash \Sigma, E_d \rightsquigarrow ks_d\end{array}}{\mathcal{P},\mathcal{D},\mathcal{E} \vdash \mathcal{R},\mathcal{S};\Sigma;\cdot \overset{\alpha}{\Longrightarrow} \mathcal{R}',\mathcal{S};\Sigma;ks :: ks_d} \text{ IN-RELEASE}$$

$$\frac{\begin{array}{c}\alpha = (id.\mathsf{Ev}(v), pc) \qquad \mathcal{P}(id.\mathsf{Ev}(v)) = pc' \qquad \Sigma(pc) = (_, \sigma^{EH}) \\ \mathsf{labelOf}(\sigma^{EH}(id)) \downarrow^i \not\sqsubseteq pc \downarrow^i \qquad \mathsf{labelOf}(\sigma^{EH}(id)) \downarrow^c \sqsubseteq pc \downarrow^c \\ E = ((id.\mathsf{Ev}(v), pc'') \mid pc'' \in \mathcal{L} \text{ s.t. } pc \sqcup pc' \sqsubseteq pc'') \qquad \Sigma, E \rightsquigarrow ks \\ \mathsf{endorse}(\mathcal{E},\mathcal{S},\Sigma,\alpha,pc') = (\mathcal{S}', E_e) \qquad pc, \mathsf{t} \vdash \Sigma, E_e \rightsquigarrow ks_e\end{array}}{\mathcal{P},\mathcal{D},\mathcal{E} \vdash \mathcal{R},\mathcal{S};\Sigma;\cdot \overset{\alpha}{\Longrightarrow} \mathcal{R},\mathcal{S}';\Sigma;ks :: ks_e} \text{ IN-SANITIZE}$$

$$\frac{\begin{array}{c}\alpha = (id.\mathsf{Ev}(v), pc) \qquad \mathcal{P}(id.\mathsf{Ev}(v)) = pc' \qquad \Sigma(pc) = (_, \sigma^{EH}) \\ \mathsf{labelOf}(\sigma^{EH}(id)) \downarrow^i \sqsubseteq pc \downarrow^i \qquad \mathsf{labelOf}(\sigma^{EH}(id)) \downarrow^c \sqsubseteq pc \downarrow^c \\ E = ((id.\mathsf{Ev}(v), pc'') \mid pc'' \in \mathcal{L} \text{ s.t. } pc \sqcup pc' \sqsubseteq pc'') \qquad \Sigma, E \rightsquigarrow ks \\ \mathsf{declassify}(\mathcal{D},\mathcal{R},\Sigma,\alpha,pc') = (\mathcal{R}', E_d) \qquad pc, \mathsf{r} \vdash \Sigma, E_d \rightsquigarrow ks_d \\ \mathsf{endorse}(\mathcal{E},\mathcal{S},\Sigma,\alpha,pc') = (\mathcal{S}', E_e) \qquad pc, \mathsf{t} \vdash \Sigma, E_e \rightsquigarrow ks_e \\ \mathsf{downgrade}(\mathcal{D},\mathcal{E},\mathcal{R},\mathcal{S},\Sigma,\alpha,pc') = E_{d,e} \qquad pc, \mathsf{rt} \vdash \Sigma, E_{d,e} \rightsquigarrow ks_{d,e}\end{array}}{\mathcal{P},\mathcal{D},\mathcal{E} \vdash \mathcal{R},\mathcal{S};\Sigma;\cdot \overset{\alpha}{\Longrightarrow} \mathcal{R}',\mathcal{S}';\Sigma;ks :: ks_d :: ks_e :: ks_{d,e}} \text{ IN-DOWNGRADE}$$

Figure 4.14: Modified input rules for transparent endorsement. Noteworthy changes are shown in red.

dual to the knowledge-based security condition to demonstrate that our SME semantics do not allow the attacker to influence the more trusted components of the system (except the fact that the system makes progress). Finally, we show if we treat declassification as a trusted behavior, the influence-based security condition may be extended so that robust declassification follows.

4.6.1 Knowledge-based security (confidentiality)

We measure the knowledge of an observer by keeping track of what they believe the inputs might have been after observing the system run. Knowledge-based security conditions are convenient because they allow us to be precise about what information (if any) is leaked. In this section, we will informally define several knowledge conditions (summarized in Table 4.1) and our knowledge-based progress-insensitive noninterference. Proofs may be found in Appendix B.2.

For someone with enough privilege to observe data up to label l, their knowledge is the set of all

$\boxed{\mathsf{endorse}(\mathcal{E}, \mathcal{S}, \Sigma, (id.\mathsf{Ev}(v), pc), pc_{\mathsf{Ev}}) = (\mathcal{S}', E)}$

$$\frac{\mathcal{E}((id.\mathsf{Ev}(v), pc), pc', \rho) = (\rho', v_e, E_e) \qquad d' = \mathsf{update}(d, v_e) \qquad E = \mathsf{transparent}(\Sigma, E_e, pc \downarrow^c)}{\mathsf{endorse}(\mathcal{E}, (\rho, d), \Sigma, (id.\mathsf{Ev}(v), pc), pc') = ((\rho', d'), E)} \text{ ENDORSE}$$

$\boxed{\mathsf{transparent}(\Sigma, E, pc_{\mathsf{Ev}}) = E'}$

$$\frac{\Sigma(pc) = (_, \sigma^{EH}) \qquad \mathsf{labelOf}(\sigma^{EH}(id)) \downarrow^c \sqsubseteq l}{\mathsf{transparent}(\Sigma, ((id.\mathsf{Ev}(v), pc) :: E), l) = (id.\mathsf{Ev}(v), pc) :: \mathsf{transparent}(\Sigma, E, l)} \text{ TRANSPARENT}$$

$$\frac{\Sigma(pc) = (_, \sigma^{EH}) \qquad \mathsf{labelOf}(\sigma^{EH}(id)) \downarrow^c \not\sqsubseteq l}{\mathsf{transparent}(\Sigma, ((id.\mathsf{Ev}(v), pc) :: E), l) = \mathsf{transparent}(\Sigma, E, l)} \text{ NOTT} \qquad \frac{}{\mathsf{transparent}(\Sigma, \cdot, l) = \cdot} \text{ EMPT}$$

$\boxed{pc, \mathsf{t} \vdash \Sigma, E \rightsquigarrow ks}$

$$\frac{\begin{array}{c}\Sigma(pc) = (_, \sigma^{EH}) \qquad \sigma^{EH}(id.\mathsf{Ev}(v)) \downarrow_l = c \\ pc' = \mathsf{labelOf}(\sigma^{EH}(id)) \sqcup \mathsf{labelOf}((\sigma^{EH}(id))(\mathsf{Ev})) \qquad \kappa = \cdot, c, P, \cdot \qquad l, \mathsf{r} \vdash \Sigma, E \rightsquigarrow ks\end{array}}{l, \mathsf{r} \vdash \Sigma, (id.\mathsf{Ev}(v), pc) :: E \rightsquigarrow (\kappa, pc', pc) :: ks} \text{ LOOKUP-T}$$

$$\frac{\Sigma(pc) = (_, \sigma^{EH}) \qquad \sigma^{EH}(id.\mathsf{Ev}(v)) \downarrow_l = \cdot \qquad l, \mathsf{t} \vdash \Sigma, E \rightsquigarrow ks}{l, \mathsf{t} \vdash \Sigma, (id.\mathsf{Ev}(v), pc) :: E \rightsquigarrow ks} \text{ LOOKUP-NOTT} \qquad \frac{}{l, \mathsf{t} \vdash \Sigma, \cdot \rightsquigarrow \cdot} \text{ LOOKUP-EMPT}$$

Figure 4.15: Additional rule changes for transparent endorsement. Noteworthy changes are shown in red text.

Knowledge	$\mathcal{K}(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l) = \{\tau \mid \exists T' \in \mathsf{runs}(\Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}), T \approx^c_l T', \tau = \mathsf{in}(T')\}$	All possible inputs producing the same observations
Progress Knowledge	$\mathcal{K}_p(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l) = \{\tau \mid \exists T' \in \mathsf{runs}(\Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}), T \approx^c_l T', \tau = \mathsf{in}(T'), \mathsf{prog}(T')\}$	All possible inputs producing the same observations *and* accept another input: $\mathsf{prog}(T')$ holds if T' can reach the consumer state
Release Knowledge	$\mathcal{K}_{rp}(T \stackrel{\alpha}{\Longrightarrow} K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l) = \{\tau \mid \exists T' \in \mathsf{runs}(\Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}), T \approx^c_l T', \tau = \mathsf{in}(T'), \mathsf{prog}(T'), \mathsf{releaseT}(T', \alpha))\}$	All possible inputs producing the same observations, accept another input, *and* release the same event: $\mathsf{releaseT}(T', \alpha)$ holds if T' can be extended to release the same event α

Table 4.1: Knowledge definitions. Knowledge and progress knowledge are for defining a knowledge-based progress-insensitive noninterference. Release knowledge and transparent knowledge account for what is leaked to the attacker through declassification and the addition of a page element/event handler capable of transparent endorsement (respectively). Complete definitions may be found in Appendix B.1.

possible inputs which might have produced the observations they made. Knowledge can also be thought of as a measure of *uncertainty*. As the attacker learns more, they will become more confident about the inputs received by the system and the knowledge set will become smaller (i.e., the attacker has become more certain about what the inputs might have been). We define the knowledge of an observer with privilege $l \in \mathcal{L}_c$:

$$\boxed{T \downarrow^p_l = \tau}$$

$$\frac{}{\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K \downarrow^p_l = \cdot}\ \textsc{TP-Base}
\qquad
\frac{pc \downarrow^p \sqsubseteq l \qquad \alpha \notin \{id.\mathsf{Ev}(v), ch(v)\}}{(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K \stackrel{(\alpha,pc)}{\Longrightarrow} T') \downarrow^p_l = \alpha :: T' \downarrow^p_l}\ \textsc{TP-Out1}$$

$$\frac{pc \downarrow^p \sqsubseteq l \vee \mathcal{P}(ch) \downarrow^p \sqsubseteq l}{(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K \stackrel{(ch(v),pc)}{\Longrightarrow} T') \downarrow^p_l = ch(v) :: T' \downarrow^p_l}\ \textsc{TP-Out2}
\qquad
\frac{pc \downarrow^p \not\sqsubseteq l \downarrow^p \qquad \mathcal{P}(ch) \downarrow^p \not\sqsubseteq l}{(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K \stackrel{(ch(v),pc)}{\Longrightarrow} T') \downarrow^p_l = T' \downarrow^p_l}\ \textsc{TP-Out-S1}$$

$$\frac{\alpha \notin \{id.\mathsf{Ev}(v), ch(v)\} \qquad pc \downarrow^p \not\sqsubseteq l}{(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K \stackrel{(\alpha,pc)}{\Longrightarrow} T') \downarrow^p_l = T' \downarrow^p_l}\ \textsc{TP-Out-S2}$$

$$\frac{\begin{array}{c}\mathcal{P}(id.\mathsf{Ev}(v)) = pc' \qquad \Sigma(pc) = (_, \sigma^{EH}) \qquad \mathsf{labelOf}(\sigma^{EH}(id)) \downarrow^i \not\sqsubseteq pc \downarrow^i \\ \mathsf{labelOf}(\sigma^{EH}(id)) \downarrow^c \not\sqsubseteq pc \downarrow^c \qquad \tau = id.\mathsf{Ev}(v) \text{ if } pc' \downarrow^p \sqcup pc \downarrow^p \sqsubseteq l \qquad \tau = \cdot \text{ otherwise}\end{array}}{(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash _, _; \Sigma; _ \stackrel{(id.\mathsf{Ev}(v),pc)}{\Longrightarrow} T') \downarrow^p_l = \tau :: T' \downarrow^p_l}\ \textsc{TP-In}$$

$$\frac{\begin{array}{c}\mathcal{P}(id.\mathsf{Ev}(v)) = pc' \qquad \Sigma(pc) = (_, \sigma^{EH}) \qquad \mathsf{labelOf}(\sigma^{EH}(id)) \downarrow^i \sqsubseteq pc \downarrow^i \\ \mathsf{labelOf}(\sigma^{EH}(id)) \downarrow^c \not\sqsubseteq pc \downarrow^c \qquad \tau = \mathsf{trRelease}(\mathcal{D}, \mathcal{R}, \Sigma, (id.\mathsf{Ev}(v), pc), pc')\end{array}}{(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash \mathcal{R}, _; \Sigma; _ \stackrel{(id.\mathsf{Ev}(v),pc)}{\Longrightarrow} T') \downarrow^p_l = \tau :: T' \downarrow^p_l}\ \textsc{TP-In-R}$$

$$\boxed{\mathsf{trRelease}(\mathcal{D}, \mathcal{R}, \Sigma, (id.\mathsf{Ev}(v), pc), pc') = \tau}$$

$$\frac{(\mathcal{R}', E) = \mathsf{declassify}(\mathcal{D}, \mathcal{R}, \Sigma, (id.\mathsf{Ev}(v), pc), pc') \qquad pc, \mathsf{r} \vdash \Sigma, E \rightsquigarrow ks \qquad \mathcal{R} \neq \mathcal{R}' \text{ or } ks \downarrow^p_l \neq \cdot}{\mathsf{trRelease}(\mathcal{D}, \mathcal{R}, \Sigma, (id.\mathsf{Ev}(v), pc), pc') = \mathsf{rls}(id.\mathsf{Ev}(v), \mathcal{R}', E \downarrow^p_l)}\ \textsc{TP-rls}$$

$$\frac{\begin{array}{c}(\mathcal{R}, E) = \mathsf{declassify}(\mathcal{D}, \mathcal{R}, \Sigma, (id.\mathsf{Ev}(v), pc), pc') \\ pc, \mathsf{r} \vdash \Sigma, E \rightsquigarrow ks \qquad ks \downarrow^p_l = \cdot \qquad pc \downarrow^p \sqcup pc' \downarrow^p \sqsubseteq l\end{array}}{\mathsf{trRelease}(\mathcal{D}, \mathcal{R}, \Sigma, (id.\mathsf{Ev}(v), pc), pc') = (id.\mathsf{Ev}(v), pc)}\ \textsc{TP-rIn}$$

$$\frac{\begin{array}{c}(\mathcal{R}, E) = \mathsf{declassify}(\mathcal{D}, \mathcal{R}, \Sigma, (id.\mathsf{Ev}(v), pc), pc') \\ pc, \mathsf{r} \vdash \Sigma, E \rightsquigarrow ks \qquad ks \downarrow^p_l = \cdot \qquad pc \downarrow^p \sqcup pc' \downarrow^p \not\sqsubseteq l\end{array}}{\mathsf{trRelease}(\mathcal{D}, \mathcal{R}, \Sigma, (id.\mathsf{Ev}(v), pc), pc') = \cdot}\ \textsc{TP-rEmp}$$

Figure 4.16: The observation ($p = c$) or behavior ($p = i$) of a trace at l (only declassification shown)

Definition 13 (Attacker Knowledge at l)**.** *An attacker's knowledge with privilege $l \in \mathcal{L}_c$ after observing trace T beginning from state σ_0 with security policy $\mathcal{P}$ and declassification policy $\mathcal{R}$ denoted $\mathcal{K}(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l)$ is defined as* $\{\tau \mid \exists T' \in \mathsf{runs}(\Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}), T \approx_l^c T', \tau = \mathsf{in}(T')$

Reading this backward: the knowledge of an observer with privilege l is the set of all inputs from execution traces T' ($\tau = \mathsf{in}(T')$) that are *observationally equivalent at l* to T ($T \approx_l^c T'$) and start from the same initial state with the same security and declassification policies ($T' \in \mathsf{runs}(\Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P})$). For now, we define an input as a user-generated event ($id.\mathsf{Ev}(v)$). We say that two runs are *observationally equivalent at l* $T \approx_l^c T'$ if they look the same to an observer with privilege l (i.e., they make the same outputs on any l-visible channel and the l-visible executions behave the same) $T \downarrow_l^c = T' \downarrow_l^c$. To define the *observation* of a trace, we need some additional syntax:

$$\begin{array}{llll} \textit{Actions:} & \alpha & ::= & id.\mathsf{Ev}(v) \mid ch(v) \mid \bullet \\ \textit{Sequence of actions}: & \tau & ::= & \cdot \mid \tau :: \alpha \mid \tau :: \mathsf{rls}(id.\mathsf{Ev}(v), \mathcal{R}, E) \end{array}$$

A sequence of actions is the result of an observation and include inputs, outputs, silent actions, and declassifications $\mathsf{rls}(...)$. The rules for the observation of a trace are shown in Figure 4.16. Note that $T \downarrow_l^p$ is parameterized by p, where $p = c$ is for confidentiality security and $T \downarrow_l^c$ is the observation of a trace at l, and $p = i$ will be for integrity security (to be defined in the next section) and $T \downarrow_l^i$ is the behavior of a trace at l. A parametric equivalence definition makes proving security for both confidentiality and integrity more efficient since, for the most part, the observation and behavior of a trace is the same, except where noted below.

The observation of an output is $ch(v)$ if the output is made on an observable channel $\mathcal{P}(ch) \downarrow^p \sqsubseteq l$ or by an observable execution $pc \downarrow^p \sqsubseteq l$ (rule TP-OUT2), otherwise the output is skipped (rule TP-OUT-S1). Inputs are observable if the security policy and source is observable (rule TP-IN), and declassifications are observable if they are successful (rule TP-IN-R via trRelease). Other actions are observable if they happen in an observable execution (rules TP-OUT1); otherwise, they are skipped (rule TP-OUT-S2).

A knowledge-based progress-sensitive noninterference says that an attacker should not be able to refine their knowledge of the secret inputs by watching the system run:

$$\mathcal{K}(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l) \subseteq_{\preceq} \mathcal{K}(T \Longrightarrow K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l)$$

We write $A \subseteq_{\preceq} B$ to mean that each element of A is a prefix of an element in B (since the last step of $T \Longrightarrow K$ may be an input). When the system takes a step ($T \Longrightarrow K$), the attacker's knowledge should not be refined; they should be equally uncertain about the possible secret inputs before and after the step. Because we run event handlers in a single-threaded loop, it is possible for an event handler to get "stuck"

in an infinite loop, which could leak something to the attacker if the loop condition is secret. Therefore, we will permit this leak and prove *progress-insensitive* noninterference instead. We define *progress knowledge* as the set of traces producing the same outputs *and* making enough progress to accept another input.

Then, a knowledge-based progress-insensitive security condition would be:

$$\mathcal{K}_p(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l) \subseteq_{\preceq} \mathcal{K}(T \Longrightarrow K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l)$$

When the system takes a step, the attacker's knowledge should not be refined outside of what is leaked by the system making progress. While this definition captures progress leaks, it does not hold if an event is declassified. For example, if a user's click on b_{Hat} is declassified for website analytics, the attacker's knowledge would be refined to include only the traces involving the click on b_{Hat}. This leak is permitted by the declassification policy, but the definition above would consider it insecure. Therefore, we define *release knowledge* as the set of traces producing the same outputs, making progress, *and* releasing *the same event*. Requiring the trace to produce the same declassification ensures that the attacker learns no more than what is explicitly declassified. Our definition for knowledge-based progress-insensitive noninterference (PINI) with declassification says that, outside of declassification, the attacker should not learn anything by watching the system take a step (outside of what they learn by the fact that the system has made progress) and when something is declassified, the attacker should only learn what is declassified. We say $\mathsf{releaseA}(T \Longrightarrow K)$ if $(\mathsf{last}(T) \Longrightarrow K) \downarrow_l^c = \mathsf{rls}(...)$, where $\mathsf{last}(T)$ is the last state in T. That is, $\mathsf{releaseA}(T \Longrightarrow K)$ means something was declassified in the last step.

Definition 14 (Knowledge-based PINI w/ Declassification)**.** *A system satisfies progress-insensitive noninterference (PINI), outside of what is declassified, against l-observers for $l \in \mathcal{L}_c$ iff given any initial global store Σ_0, security policy $\mathcal{P}$, and declassification policy $\mathcal{R}$, it is the case that for all traces T, actions α, and configurations K s.t. $(T \stackrel{\alpha}{\Longrightarrow} K) \in \mathsf{runs}(\Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P})$, then, the following holds*

- *If $\mathsf{releaseA}(T \stackrel{\alpha}{\Longrightarrow} K)$: $\mathcal{K}(T \stackrel{\alpha}{\Longrightarrow} K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l) \supseteq_{\preceq} \mathcal{K}_{rp}(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, \alpha, l)$*
- *Otherwise: $\mathcal{K}(T \stackrel{\alpha}{\Longrightarrow} K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l) \supseteq_{\preceq} \mathcal{K}_p(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l)$*

Example Recall our example of leaks *between* executions from Section 4.2.3 where the security policy is that keypress events should be Secret, but clicks may be declassified from Secret to Public. The host adds a different button to the page depending on what the user types, and the attacker adds all possible keypress buttons $b_1, ..., b_n$. They also register an $\mathsf{onClick}$ event handler to the buttons which output $1, ..., n$ (respectively) on a (P, U) channel.

When the user presses a key on their keyboard, the attacker is not sure which key the user pressed. Their knowledge at this point includes all possible keypresses:

$$\mathcal{K}(K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, P) = \{f.\mathsf{keyPress}(1), ..., f.\mathsf{keyPress}(n)\}$$

The keypress triggers the $\mathsf{onKeypress}$ event handler which adds a different button to the user's page, depending on which key they pressed. Suppose the attacker also registered a Click event handler to b_{secret} to directly leak the user's keypress through a (P, U) channel. If the output were allowed, the attacker would be able to eliminate all the traces where the user pressed a different key because they could not have produced the output they received:

$$\mathcal{K}(K \overset{ch(i)}{\Longrightarrow} K', \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, P) = \{\sout{f.\mathsf{keyPress}(1)}, ..., f.\mathsf{keyPress}(secret) :: b_{\mathsf{secret}}.\mathsf{Click}(_), ..., \sout{f.\mathsf{keyPress}(n)}\}$$

Since the attacker's knowledge is refined by the observation, our knowledge-based security condition would correctly identify this output as insecure:

$$\mathcal{K}(K \overset{ch(i)}{\Longrightarrow} K', ...) \not\supseteq \mathcal{K}(K, ...)$$

In reality, the SME monitor would prevent the output from the (S, U) execution to the (P, U) channel. The user's click would not be able to directly leak their keypress to the attacker, but it could be declassified to the (P, U) execution. Since the attacker added all possible buttons $b_1, ..., b_n$, they are guaranteed to trigger the leaky output and learn which key the user pressed. Because the $\mathsf{releaseA}$ condition allows the attacker's knowledge to be refined by declassifications, our security condition for confidentiality does not catch this leak. Next, we describe our security condition for integrity and how this condition can be used to describe both progress-insensitive noninterference as well as robust declassification.

4.6.2 Influence-based security (integrity)

We measure the attacker's ability to change the behavior of the system with a dual condition to attacker knowledge called *attacker influence*. In this section, we explain the intuition behind attacker influence (based on attacker power from prior work [10]). Then, we describe our influence definitions (summarized in Table 4.2), including *robust influence* which captures what is leaked by adding an element to the page that is capable of robust declassification.

At a high level, an attacker's influence is the set of all untrusted inputs which might have produced the same behaviors. The attacks included in the influence set have the same relative ability to influence the system's behavior. If the attacker has no influence over the system, then the set should include all possible attacks: all the attacks are equally powerless. As the system runs, if the attacker's influence is refined, it

Influence	$\mathcal{I}(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l) = \{\tau \mid \exists T' \in \mathsf{runs}(\Sigma_0, \mathcal{S}, \mathcal{R}, \mathcal{P}), T \approx_l^i T', \tau = \mathsf{in}(T')\}$	All possible inputs producing the same trusted actions
Progress Influence	$\mathcal{I}_p(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l) = \{\tau \mid \exists T' \in \mathsf{runs}(\Sigma_0, \mathcal{S}, \mathcal{R}, \mathcal{P}), T \approx_l^i T', \tau = \mathsf{in}(T'), \mathsf{prog}(T')\}$	All possible inputs producing the same trusted actions *and* accept another input: $\mathsf{prog}(T')$ holds if T' can reach the consumer state
Robust Influence	$\mathcal{I}_{rp}(T \overset{\alpha}{\Longrightarrow} K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l) = \{\tau \mid \exists T' \in \mathsf{runs}(\Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}), T \approx_l^i T', \tau = \mathsf{in}(T'), \mathsf{prog}(T'), \mathsf{robustT}(T', \alpha))\}$	All possible inputs producing the same trusted actions, accept another input, *and* capable of the same robust declassifications: $\mathsf{robustT}(T', \alpha)$ holds if T' can be extended to create the same trusted page event α

Table 4.2: Influence definitions. Influence and progress influence are for defining an influence-based progress-insensitive noninterference. Robust influence is for defining robust declassification. Complete definitions may be found in Appendix B.1.

indicates that some attack(s) must be more powerful than the others because the ones eliminated could not have led to the observed behavior. We define the influence of an attacker over behaviors at l (for $l \in \mathcal{L}_i$) below:

Definition 15 (Attacker Influence over l)**.** *An attacker's influence over behaviors at $l \in \mathcal{L}_i$ in T beginning from state σ_0 with security policy $\mathcal{P}$ and endorsement policy $\mathcal{S}$ denoted $\mathcal{I}(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l)$ is defined as $\{\tau \mid \exists T' \in \mathsf{runs}(\Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}), T \approx_l^i T', \tau = \mathsf{in}(T')\}$*

Reading this backward: the influence of an attacker over behaviors at l is the set of all τ which are inputs from execution traces T' ($\tau = \mathsf{in}(T')$) that are *behaviorally equivalent at l* to T ($T \approx_l^i T'$) and start from the same initial state with the same security and declassification policies ($T' \in \mathsf{runs}(\Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P})$). We say that two runs are *behaviorally equivalent at l* if they produce the same l-trusted actions (i.e., they make the same outputs on any l-trusted channel and the l-trusted executions behave the same). Recall that $T \downarrow_l^p$ in Figure 4.16 is parameterized by p and applies to both observational equivalence (in the previous section) and behavioral equivalence.

Then, an influence-based progress-sensitive noninterference says that the attacker's influence over a system should never be refined:

$$\mathcal{I}(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l) \subseteq_{\preceq} \mathcal{I}(T \Longrightarrow K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l)$$

Similar to the way that a loop condition could leak something secret to an attacker, a loop condition could allow the attacker to control whether the system makes progress. We define *progress influence* as the set of traces producing the same behaviors *and* making enough progress to accept another input. Then, an influence-based progress-insensitive security condition would be:

$$\mathcal{I}_p(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l) \subseteq_{\preceq} \mathcal{I}(T \Longrightarrow K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l)$$

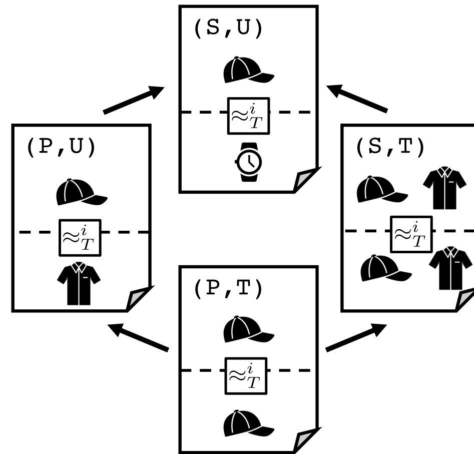


Figure 4.17: The states above and below the dotted line are behaviorally equivalent at T even though there are different products in the (P, U) and (S, U) states.

When the system takes a step, the attacker's influence should not be refined, outside of what control they have over whether the system makes progress.

4.6.3 Influence-based security with robust declassification

In addition to showing that the attacker does not have influence over trusted behaviors, we also want to show that the attacker does not influence declassification. In this section, we show that we can define robust declassification by extending our influence-based security condition.

Initially, we try to naïvely include declassification in behavioral equivalence. We model an *active attacker* by treating the addition of a page element or event handler ($\mathsf{new}(id, pc), \mathsf{new}(id, eh, pc)$) as an input. A system is robust if any of these *attacks* (i.e., an untrusted source adding new page elements or event handlers) have equivalent power. That is, when a new declassification happens, we will know the attacker's code influenced the declassification if the set of attacks *without* the new page element/event handler could not have led to the same declassification.

But it turns out that this is leads to false positives. Consider the online shop described in Section 4.2. The buttons are all loaded by the Trusted host, so they can safely influence declassification: the declassifications in this example are robust. The issue is that behavioral equivalence at T only guarantees that the Trusted executions behave the same. See the example of two equivalent traces in Figure 4.17. The (S, T) execution has the same products in both traces, as does the (P, T) shop, but even among two equivalent runs, the (S, U) and (P, U) executions may have different products. When the user clicks the hat in the (S, U) execution, the click is declassified. But it is not possible to produce the same declassification in the equivalent state because there is no hat for the user to click on. This makes it appear as though the

$$\frac{\alpha \in \{\mathsf{new}(id, l_{src}), \mathsf{new}(id, eh, l_{id}, l_{src})\} \qquad pc \downarrow^i \not\sqsubseteq l \\ \tau = \mathsf{r}(id, pc) \text{ if } l_{src} \sqsubseteq pc \downarrow^i \qquad \tau = \mathsf{r}(id, eh, pc) \text{ if } l_{src} \sqcup l_{id} \sqsubseteq pc \downarrow^i \qquad \tau = \cdot \text{ otherwise}}{(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K \overset{(\alpha, pc)}{\Longrightarrow} T') \downarrow_l^i = \tau :: T' \downarrow_l^i} \text{TP-NEWI}$$

Figure 4.18: New rule for the behavior of a trace for robust declassification

attacker had some influence over the declassification, even though the declassification is actually robust against their influence.

To make these benign influence refinements concrete, we introduce *robust influence* for when trusted page elements are created. Robust influence is the set of traces producing the same elements, making progress, *and* capable of producing the same robust declassifications in the untrusted executions. This is similar to release knowledge from Section 4.6.1. We say $\mathsf{robustA}(T)$ if $(\mathsf{last}(T) \Longrightarrow K) \downarrow^i = \mathsf{r}(...)$, where $\mathsf{last}(T)$ is the last state in T. That is, $\mathsf{robustA}(T \Longrightarrow K)$ means something capable of robust declassification was added to an Untrusted execution.

To model an *active* attacker's ability to add code to the page, we emit an action for dynamically-generated elements and event handlers. $\mathsf{new}(id, pc)$ is a new page element identified by id to the execution at security level pc, while $\mathsf{new}(id, eh, pc)$ is a new event handler eh registered to the element identified by id in the execution at security level pc. Sequences of actions also include the page elements/event handlers *which are capable of robust declassification* $\mathsf{r}(...)$.

$$\begin{array}{lll} \textit{Actions:} \quad \alpha & ::= & id.\mathsf{Ev}(v) \mid ch(v) \mid \mathsf{new}(id, pc) \mid \mathsf{new}(id, eh, pc) \mid \bullet \\ \textit{Sequence of actions}: \tau & ::= & \cdot \mid \tau :: \alpha \mid \tau :: \mathsf{rls}(id.\mathsf{Ev}(v), \mathcal{R}, E) \mid \tau :: \mathsf{r}(id, pc) \mid \tau :: \mathsf{r}(id, eh, pc) \end{array}$$

We modify the behavior of a trace as shown in Figure 4.18. When a new page element is created or event handler is registered, this is not considered an observable action unless it is capable of a robust declassification (rule TP-NEWI).

Definition 16 (Influence-based PINI w/ Robust Declassification)**.** *A system satisfies progress-insensitive non-interference (PINI) with robust declassification for behaviors at $l \in \mathcal{L}_i$ iff given any initial global store Σ_0, security policy $\mathcal{P}$, and declassification policy $\mathcal{R}$, it is the case that for all traces T, actions α, and configurations K s.t. $(T \overset{\alpha}{\Longrightarrow} K) \in \mathsf{runs}(\Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P})$, then, the following holds*

- *If $\mathsf{robustA}(T \overset{\alpha}{\Longrightarrow} K)$: $\mathcal{I}(T \overset{\alpha}{\Longrightarrow} K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l) \supseteq_{\preceq} \mathcal{I}_{rp}(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, \alpha, l)$*
- *Otherwise: $\mathcal{I}(T \overset{\alpha}{\Longrightarrow} K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l) \supseteq_{\preceq} \mathcal{I}_p(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l)$*

Example To illustrate how this new definition is sufficient for defining robust declassification, we return to the examples from Section 4.2. In the example of a leak *within* an execution, the Untrusted attacker registers the event handler onLoad^U and the Trusted host registers onLoad^T to add buttons to the page.

After the page finishes loading, we know that the Trusted *"Agree"* button, b_{Agree}, must have been dynamically loaded because all the behaviorally equivalent Trusted executions have run onLoad^T. On the other hand, we are not sure whether the Untrusted "Click me!" button, was added because the Untrusted pages are equivalent whether or not onLoad^U has run. At this point, the attacks where the *"Click me!"* button has been added are equally as powerful as the attacks without it:[3]

$$\mathcal{I}(K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, T) = \{\mathsf{new}(b)^T, \mathsf{new}(b)^U :: \mathsf{new}(b)^T, \ldots\}$$

If the system allowed the click on the Untrusted b_{Agree} to be declassified, it would mean there *must* be a *"Click me!"* button on the (S, U) copy of the page. Therefore, the only viable attack leading to this behavior are the ones including the Untrusted b_{Agree} button:

$$\mathcal{I}(T \overset{b_{\mathsf{Atk}}}{\Longrightarrow} K, \mathcal{R}, \mathcal{S}, \mathcal{P}, T) = \{\cancel{\mathsf{new}(b)^T}, \mathsf{new}(b)^U :: \mathsf{new}(b)^T :: b^U.\mathsf{Click}(\ ), \ldots\}$$

Because $\mathcal{I}(T \overset{b_{\mathsf{Atk}}}{\Longrightarrow} K, \mathcal{R}, \mathcal{S}, \mathcal{P}, T) \not\supseteq_{\preceq} \mathcal{I}_p(T, \mathcal{R}, \mathcal{S}, \mathcal{P}, T)$, the attacker must have had influence over the declassification, so it is not robust.

The example of a leak *between* executions is similar. Here, the Trusted host adds a different button to the page depending on which key the user pressed and the Untrusted attacker adds all possible buttons.

After the user presses a key on their keyboard, we know that there is one button on the (S, T) page (based on the actual *secret* value) and another button on the (U, T) page (based on the default value dv) because all of the behaviorally equivalent Trusted executions have run the Trusted event handler in response to the user's keypress. We also know that the (S, U) and (P, U) copies of the page must include b_{secret} and b_{dv} (respectively) because those buttons are capable of robust declassification since they were added by the host. On the other hand, we are not sure whether the attacker has added their buttons, because the Untrusted pages are equivalent with or without those buttons:[4]

$$\mathcal{I}(K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l) = \{\mathsf{new}(b_{\mathsf{secret}}) :: \mathsf{new}(b_{\mathsf{dv}}), \mathsf{new}(b_{\mathsf{secret}}) :: \mathsf{new}(b_{\mathsf{dv}}) :: \mathsf{new}(b_1)^U :: \ldots :: \mathsf{new}(b_n)^U, \ldots\}$$

Now, when the user's click on b_{secret} in the (S, U) page is declassified to the matching button b_i in the (P, U) page, we know there must be a b_i button on the (P, U) copy of the page to capture the event. Then, the only viable attack is the one where b_i has been added to the page:

$$\mathcal{I}(K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l) = \{\cancel{\mathsf{new}(b_{\mathsf{secret}}) :: \mathsf{new}(b_{\mathsf{dv}})}, \mathsf{new}(b_{\mathsf{secret}}) :: \mathsf{new}(b_{\mathsf{dv}}) :: \mathsf{new}(b_1)^U :: \ldots :: \mathsf{new}(b_n)^U, \ldots\}$$

[3] For brevity, we write $\mathsf{new}(b)^T$ instead of $\mathsf{new}(b_{\mathsf{Agree}}, (S, T))$ and $\mathsf{new}(b_{\mathsf{Agree}}, (P, T))$, and likewise for $\mathsf{new}(b)^U$ for the Untrusted executions.

[4] For brevity, we write $\mathsf{new}(b_{\mathsf{secret}})$ to mean that the button was added in all executions.

$\boxed{T \downarrow_l^p = \tau}$

$$\frac{\begin{array}{c}\mathcal{P}(id.\mathsf{Ev}(v)) = pc' \quad \Sigma(pc) = (_, \sigma^{EH}) \quad \mathsf{labelOf}(\sigma^{EH}(id)) \downarrow^i \not\sqsubseteq pc \downarrow^i \\ \mathsf{labelOf}(\sigma^{EH}(id)) \downarrow^c \sqsubseteq pc \downarrow^c \quad \tau = \mathsf{trSanitize}(\mathcal{E}, \mathcal{S}, \Sigma, (id.\mathsf{Ev}(v), pc), pc')\end{array}}{(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash _, \mathcal{S}; \Sigma; _ \overset{(id.\mathsf{Ev}(v),pc)}{\Longrightarrow} T') \downarrow_l^p = \tau :: T' \downarrow_l^p} \text{ TP-IN-E}$$

$$\frac{\begin{array}{c}\mathcal{P}(id.\mathsf{Ev}(v)) = pc' \quad \Sigma(pc) = (_, \sigma^{EH}) \quad \mathsf{labelOf}(\sigma^{EH}(id)) \downarrow^i \sqsubseteq pc \downarrow^i \\ \mathsf{labelOf}(\sigma^{EH}(id)) \downarrow^c \sqsubseteq pc \downarrow^c \quad \tau = \mathsf{trDownrade}(\mathcal{D}, \mathcal{E}, \mathcal{R}, \mathcal{S}, \Sigma, (id.\mathsf{Ev}(v), pc), pc')\end{array}}{(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash \mathcal{R}, \mathcal{S}; \Sigma; _ \overset{(id.\mathsf{Ev}(v),pc)}{\Longrightarrow} T') \downarrow_l^p = \tau :: T' \downarrow_l^p} \text{ TP-IN-D}$$

$$\frac{\begin{array}{c}\alpha \in \{\mathsf{new}(id, l_{src}), \mathsf{new}(id, eh, l_{id}, l_src)\} \quad pc \downarrow^c \not\sqsubseteq l \\ \tau = \mathsf{t}(id, pc) \text{ if}g\ l_{src} \sqsubseteq pc \downarrow^c \quad \tau = \mathsf{t}(id, eh, pc) \text{ if } l_{id} \sqcup l_{src} \sqsubseteq pc \downarrow^c \quad \tau = \cdot \text{ otherwise}\end{array}}{(\mathcal{P}, \mathcal{D}, \mathcal{E}, \vdash K \overset{(\alpha,pc)}{\Longrightarrow} T') \downarrow_l^c = \tau :: T' \downarrow_l^c} \text{ TP-NEW-C}$$

$$\frac{\begin{array}{c}\alpha \in \{\mathsf{new}(id, l_{src}), \mathsf{new}(id, eh, l_{id}, l_src)\} \\ pc \downarrow^i \not\sqsubseteq l \quad \tau = \mathsf{r}(id, pc) \text{ if } l_{src} \sqsubseteq pc \downarrow^i \quad \tau = \mathsf{r}(id, eh, pc) \text{ if } l_{id} \sqcup l_{src} \sqsubseteq pc \downarrow^i \quad \tau = \cdot \text{ otherwise}\end{array}}{(\mathcal{P}, \mathcal{D}, \mathcal{E}, \vdash K \overset{(\alpha,pc)}{\Longrightarrow} T') \downarrow_l^i = \tau :: T' \downarrow_l^i} \text{ TP-NEW-I}$$

Figure 4.19: Additional rules for the observation ($p = c$) or behavior ($p = i$) of a trace at l to account for endorsement, downgrading, and the creation of a new page element.

Since the attacker's influence has been refined we know this example is not robust either.

Finally, consider the secure web shop where the host adds products to the page and declassifies click counts so that a (P, U) library can do analytics for them. All the elements are added by the Trusted host, so the events from these elements can be declassified safely. From the robustA case in Definition 16, the attacker's influence can be refined by the addition of these elements to include only the traces that load the same products on the web store. Since the element is Trusted, this refinement does not impact security, but does ensure that the Trusted page elements in the Untrusted executions stay synchronized in equivalent traces. Our security condition correctly identifies this as robust.

4.6.4 Knowledge-based security with transparent endorsement

The same way we define robust declassification by including declassifications in our definition for the behavior of a trace, we can define transparent endorsement by including endorsements in our definition for the observation of a trace. We define new actions for when we observe the behavior of a trace. For endorsements, we use $\mathsf{sntz}(id.\mathsf{Ev}(v), \mathcal{S}, E)$, and when an input is both declassified and endorsed, the observation/behavior is $\mathsf{down}(id.\mathsf{Ev}(v), \mathcal{R}, \mathcal{S}, E_d, E_e, E)$. We also add actions to reflect the creation of page elements/event handlers whose events are safe to endorse $\mathsf{t}(...)$. We show the relevant rules for the observation/behavior of a trace (with endorsement/downgrading) in Figure 4.18.

$\boxed{\mathsf{trSanitize}(\mathcal{E}, \mathcal{S}, \Sigma, (id.\mathsf{Ev}(v), pc), pc') = \tau}$

$$\frac{(\mathcal{R}', E) = \mathsf{endorse}(\mathcal{E}, \mathcal{S}, \Sigma, (id.\mathsf{Ev}(v), pc), pc') \qquad pc, \mathsf{t} \vdash \Sigma, E \rightsquigarrow ks \qquad \mathcal{R} \neq \mathcal{R}' \text{ or } ks \downarrow_l^p \neq \cdot}{\mathsf{trSanitize}(\mathcal{E}, \mathcal{S}, \Sigma, (id.\mathsf{Ev}(v), pc), pc') = \mathsf{sntz}(id.\mathsf{Ev}(v), \mathcal{S}', E \downarrow_l^p)} \text{ TP-SNTZ}$$

$$\frac{(\mathcal{R}, E) = \mathsf{endorse}(\mathcal{E}, \mathcal{S}, \Sigma, (id.\mathsf{Ev}(v), pc), pc') \qquad pc, \mathsf{t} \vdash \Sigma, E \rightsquigarrow ks \qquad ks \downarrow_l^p = \cdot \qquad pc \downarrow^p \sqcup pc' \downarrow^p \sqsubseteq l}{\mathsf{trSanitize}(\mathcal{E}, \mathcal{S}, \Sigma, (id.\mathsf{Ev}(v), pc), pc') = (id.\mathsf{Ev}(v), pc)} \text{ TP-SIN}$$

$$\frac{\begin{array}{c}(\mathcal{R}, E) = \mathsf{endorse}(\mathcal{E}, \mathcal{S}, \Sigma, (id.\mathsf{Ev}(v), pc), pc') \\ pc, \mathsf{t} \vdash \Sigma, E \rightsquigarrow ks \qquad ks \downarrow_l^p = \cdot \qquad pc \downarrow^p \sqcup pc' \downarrow^p \not\sqsubseteq l\end{array}}{\mathsf{trSanitize}(\mathcal{E}, \mathcal{S}, \Sigma, (id.\mathsf{Ev}(v), pc), pc') = \cdot} \text{ TP-SEMP}$$

$\boxed{\mathsf{trDowngrade}(\mathcal{D}, \mathcal{E}, \mathcal{R}, \mathcal{S}, \Sigma, (id.\mathsf{Ev}(v), pc), pc') = \tau}$

$$\frac{\begin{array}{c}\tau_d = \mathsf{trRelease}(\mathcal{D}, \mathcal{R}, \Sigma, (id.\mathsf{Ev}(v), pc), pc') \\ \tau_e = \mathsf{trSanitize}(\mathcal{E}, \mathcal{S}, \Sigma, (id.\mathsf{Ev}(v), pc), pc') \qquad E = \mathsf{downgrade}(\mathcal{D}, \mathcal{E}, \mathcal{R}, \mathcal{S}, \Sigma, (id.\mathsf{Ev}(v), pc), pc') \\ \mathsf{rt} \vdash \Sigma, E \rightsquigarrow ks \qquad \tau_d = \mathsf{rls}(...) \wedge \tau_e = \mathsf{sntz}(...) \text{ or } ks \neq \cdot\end{array}}{\mathsf{trDowngrade}(\mathcal{D}, \mathcal{E}, \mathcal{R}, \mathcal{S}, \Sigma, (id.\mathsf{Ev}(v), pc), pc') = \mathsf{down}(id.\mathsf{Ev}(v), \tau_d, \tau_e, E \downarrow_l^p)} \text{ TP-DOWN}$$

$$\frac{\begin{array}{c}\tau_d = \mathsf{trRelease}(\mathcal{D}, \mathcal{R}, \Sigma, (id.\mathsf{Ev}(v), pc), pc') \qquad \tau_e = \mathsf{trSanitize}(\mathcal{E}, \mathcal{S}, \Sigma, (id.\mathsf{Ev}(v), pc), pc') \\ E = \mathsf{downgrade}(\mathcal{D}, \mathcal{E}, \mathcal{R}, \mathcal{S}, \Sigma, (id.\mathsf{Ev}(v), pc), pc') \qquad \mathsf{rt} \vdash \Sigma, E \rightsquigarrow ks \\ ks = \cdot \qquad \tau = \tau_d \text{ if } \tau_d = \mathsf{rls}(...) \wedge \tau_e \neq \mathsf{sntz}(...) \qquad \tau = \tau_e \text{ if } \tau_e = \mathsf{sntz}(...) \wedge \tau_d \neq \mathsf{rls}(...)\end{array}}{\mathsf{trDowngrade}(\mathcal{D}, \mathcal{E}, \mathcal{R}, \mathcal{S}, \Sigma, (id.\mathsf{Ev}(v), pc), pc') = \tau} \text{ TP-DIN}$$

$$\frac{\begin{array}{c}\tau_d = \mathsf{trRelease}(\mathcal{D}, \mathcal{R}, \Sigma, (id.\mathsf{Ev}(v), pc), pc') \qquad \tau_e = \mathsf{trSanitize}(\mathcal{E}, \mathcal{S}, \Sigma, (id.\mathsf{Ev}(v), pc), pc') \\ E = \mathsf{downgrade}(\mathcal{D}, \mathcal{E}, \mathcal{R}, \mathcal{S}, \Sigma, (id.\mathsf{Ev}(v), pc), pc') \qquad \mathsf{rt} \vdash \Sigma, E \rightsquigarrow ks \qquad ks = \cdot \\ \tau_d \neq \mathsf{rls}(...) \qquad \tau_e \neq \mathsf{sntz}(...) \qquad \tau = (id.\mathsf{Ev}(v), pc) \text{ if } pc \downarrow^p \sqcup pc' \downarrow^p \sqsubseteq l \qquad \tau = \cdot \text{ otherwise}\end{array}}{\mathsf{trDowngrade}(\mathcal{D}, \mathcal{E}, \mathcal{R}, \mathcal{S}, \Sigma, (id.\mathsf{Ev}(v), pc), pc') = \tau} \text{ TP-DEMP}$$

Figure 4.20: Helper functions for trace observation and behavior for endorsement and downgrading.

$$\begin{array}{llll}
\textit{Actions:} & \alpha & ::= & id.\mathsf{Ev}(v) \mid ch(v) \mid \mathsf{new}(id, pc) \mid \mathsf{new}(id, eh, pc) \mid \bullet \\
\textit{Sequence of actions}: & \tau & ::= & \cdot \mid \tau :: \alpha \mid \tau :: \mathsf{rls}(id.\mathsf{Ev}(v), \mathcal{R}, E) \mid \tau :: \mathsf{sntz}(id.\mathsf{Ev}(v), \mathcal{S}, E) \\
 & & & \mid \tau :: \mathsf{down}(id.\mathsf{Ev}(v), \mathcal{R}, \mathcal{S}, E_d, E_e, E) \mid \tau :: \mathsf{r}(id, pc) \mid \tau :: \mathsf{r}(id, eh, pc) \\
 & & & \mid \tau :: \mathsf{t}(id, pc) \mid \tau :: \mathsf{t}(id, eh, pc)
\end{array}$$

Rule TP-IN-E handles endorsement using the trSanitize helper function and is similar to the one for declassification (rule TP-IN-R). Rule TP-IN-D handles the case where the input is eligible for both declassification and endorsement. The helper function trDowngrade returns down(...) if the input resulted in both a declassification and endorsement, which is true if trRelease = rls(...) and trSanitize = sntz(...), or if the lookup function results in at least one event handler. If the input resulted in only a declassification or endorsement, the trDowngrade returns rls(...) or sntz(...), respectively.

Transparent Knowledge	$\mathcal{K}_{tp}(T \overset{\alpha}{\Longrightarrow} K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l) = \{\tau \mid \exists T' \in \mathsf{runs}(\Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}), T \approx_l^c T', \tau = \mathsf{in}(T'), \mathsf{prog}(T'), \mathsf{transparentT}(T', \alpha)\}$	All possible inputs producing the same public actions, accept another input, *and* capable of the same transparent endorsements: $\mathsf{transparentT}(T', \alpha)$ holds if T' can be extended to create the same public page event α
Sanitization Influence	$\mathcal{I}_{ep}(T \overset{\alpha}{\Longrightarrow} K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l) = \{\tau \mid \exists T' \in \mathsf{runs}(\Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}), T \approx_l^c T', \tau = \mathsf{in}(T'), \mathsf{prog}(T'), \mathsf{sanitizeT}(T', \alpha)\}$	All possible inputs producing the same trusted actions, accept another input, *and* endorse the same event: $\mathsf{sanitizeT}(T', \alpha)$ holds if T' can be extended to endorse the same event α

Table 4.3: Additional knowledge and influence conditions for defining transparent endorsement. Complete definitions may be found in Appendix B.1.

In Table 4.3, we add a new knowledge condition to account for the information gained by the attacker when a new page element or event handler is added which is capable of transparent endorsement (dual condition to robust influence in Table 4.2) and an integrity condition to account for the additional influence from endorsement (dual condition to release knowledge in Table 4.1).

Finally, we extend our security definitions to include endorsement. Definition 17 is a new influence-based progress-insensitive noninterference (PINI) with endorsement and robust declassification. It is the same as Definition 16, except that there is a new condition for endorsed events. The last step was an endorsement at l, $\mathsf{sntzA}(T \Longrightarrow K, l)$, if the behavior at l the last step of the trace was an endorsement $(T \Longrightarrow K) \downarrow_l^i \in \{\mathsf{sntz}(...), \mathsf{down}(...)\}$. In this case, the additional influence the attacker has should be restricted to the influence gained by the endorsement. This is similar to the releaseA condition in Definition 14. Otherwise, the definition does not change.

Definition 17 (Influence-based PINI w/ Endorsement, Robustness)**.** *A system satisfies progress-insensitive noninterference with robust declassification for behaviors at $l \in \mathcal{L}_i$ iff given any initial global store Σ_0, security policy $\mathcal{P}$, and declassification policy $\mathcal{R}$, it is the case that for all traces T, actions α, and configurations K s.t. $(T \overset{\alpha}{\Longrightarrow} K) \in \mathsf{runs}(\Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P})$, then, the following holds*

- *If* $\mathsf{sntzA}(\mathsf{last}(T) \overset{\alpha_l}{\Longrightarrow} K, l)$: $\mathcal{I}(T \overset{\alpha_l}{\Longrightarrow} K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l) \supseteq_{\preceq} \mathcal{I}_{ep}(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, \alpha_l, l)$
- *If* $\mathsf{robustA}(T \overset{\alpha}{\Longrightarrow} K, l)$: $\mathcal{I}(T \overset{\alpha}{\Longrightarrow} K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l) \supseteq_{\preceq} \mathcal{I}_{rp}(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, \alpha, l)$
- *Otherwise:* $\mathcal{I}(T \overset{\alpha}{\Longrightarrow} K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l) \supseteq_{\preceq} \mathcal{I}_p(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l)$

Definition 18 is a new knowledge-based progress-insensitive noninterference (PINI) with declassification and transparent endorsement. It is the same as Definition 14, except that there is a new condition for the information leaked due to the existence of page elements/event handlers capable of transparent endorsement, similar to the robustA condition from Definition 16. We also add downgrade actions down(...) to releaseA so that downgrades are also considered declassifications.

Definition 18 (Knowledge-based PINI w/ Declassification, Transparency). *A system satisfies progress-insensitive noninterference, outside of what is declassified, against l-observers for $l \in \mathcal{L}_c$ iff given any initial global store Σ_0, security policy $\mathcal{P}$, and declassification policy $\mathcal{R}$, it is the case that for all traces T, actions α, and configurations K s.t. $(T \stackrel{\alpha}{\Longrightarrow} K) \in \mathsf{runs}(\Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P})$, then, the following holds*

- *If* $\mathsf{releaseA}(T \stackrel{\alpha}{\Longrightarrow} K)$: $\mathcal{K}(T \stackrel{\alpha}{\Longrightarrow} K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l) \supseteq_{\preceq} \mathcal{K}_{rp}(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, \alpha, l)$
- *If* $\mathsf{trnsprntA}(\mathsf{last}(T) \stackrel{\alpha_l}{\Longrightarrow} K, l)$: $\mathcal{K}(T \stackrel{\alpha_l}{\Longrightarrow} K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l) \supseteq_{\preceq} \mathcal{K}_{tp}(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, \alpha_l, l)$
- *Otherwise:* $\mathcal{K}(T \stackrel{\alpha}{\Longrightarrow} K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l) \supseteq_{\preceq} \mathcal{K}_p(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l)$

4.6.5 Metatheory

We prove that our enforcement mechanism is sound. Formally:

Theorem 19 (Soundness). *$\forall \mathcal{P}, \mathcal{D}, \mathcal{E}, \Sigma_0$, the SME state Σ_0 satisfies knowledge-based progress-insensitive noninterference with declassification and transparent endorsement at l_c and influence-based progress-insensitive noninterference with endorsement and robust declassification at l_i w.r.t. the security policy $\mathcal{P}$, declassification policy $\mathcal{D}$, endorsement policy $\mathcal{E}$*

Proofs may be found in Appendix B.2. Robust declassification follows from influence-based progress-insensitive noninterference. If we treat declassifications as trusted and prove that untrusted sources cannot influence trusted behaviors, then it must be the case that the declassifications are robust.

Corollary 20 (Robust Declassification). *$\forall \mathcal{P}, \mathcal{D}, \mathcal{E}, \Sigma_0$ s.t. Σ_0 satisfies influence-based progress-insensitive noninterference with endorsement and robust declassification at l_i w.r.t. the security policy $\mathcal{P}$, declassification policy $\mathcal{D}$, and endorsement policy $\mathcal{E}$, then an attacker at $l_i' \in \mathcal{L}_i$ with $l_i \sqsubset l_i'$ has no influence over whether the user's events at l_i are declassified (except for what is endorsed).*

Similarly, transparent endorsement follows from Definition 18.

4.7 Discussion

(Transparent) endorsement and qualified robustness The focus of this work is robust declassification, but like our "influence-based" security condition is the integrity dual of "knowledge-based" security conditions for confidentiality, (transparent) endorsement is the integrity dual of (robust) declassification. Here, we define a stateful endorsement dual to our stateful robust declassification but have not explored how it might be used. In future work, we will investigate how transparent endorsement might be useful.

It is important to note that because an event associated with an attacker-controlled page element might be endorsed, we are actually proving a *qualified robustness* condition [67] (and *qualified transparency*) which says that the attacker does not have influence over declassifications, outside of what has been endorsed (and we do not endorse what the attacker does not have privilege to see, outside of what has been declassified). This does not otherwise change our security conditions because sanitized influence (and release knowledge) already capture this.

4.7.1 Alternative DOM models

In our model, each execution has its own copy of the DOM, similar to Chapter 3 and prior work [26, 30]. Another option would be to have a single DOM [35, 91]. In these models, the security policy would determine which API calls would succeed and which would be replaced with a default value. It would be challenging to have the same dynamic feature flexibility in the first model, and in the next chapter, we show that a single tainted DOM is susceptible to implicit leaks, so here we opted for multiple DOMs.

4.8 Summary

In this chapter, we developed a monitor which combines SME and taint tracking that allows us to prevent untrusted parties from influencing declassification. This monitor is more fine-grained than the one described in Chapter 3 and permits the benign declassifications involving trusted dynamic features—without sacrificing security. We prove progress-insensitive noninterference for both confidentiality and integrity using knowledge-based and influence-based security conditions, respectively. Finally, we show that robust declassification follows from our novel influence-based security condition when we treat declassifications as trusted behaviors in our noninterference definition.

Chapter 5

Compositional IFC for Reactive Systems

In this chapter we present a compositional framework for using different IFC techniques for different components in our reactive model from previous chapters. [1]

5.1 Overview

Previous chapters use SME to enforce information flow policies, but many runtime mechanisms have been developed for enforcing information flow control (IFC) policies [35, 22, 88, 24, 41, 52, 20, 21]. Broadly, these approaches can be classified into *multi-execution approaches* [38, 15, 17], and *taint tracking approaches* [13, 14, 95, 84].

Multi-execution-based approaches, like secure multi-execution (SME) [38], and faceted execution (MF) [15], execute code multiple times at different security levels. These ensure that the code executing at a particular level only outputs data at the same security level and replace sensitive data from higher security levels with "default" values. *Taint tracking approaches* (TT) annotate data with *labels* to indicate its security level and can suppress outgoing sensitive data to publicly observable channels to prevent leaks. These approaches differ in performance, how much they alter the semantics of safe programs (transparency), and the relative strength of their security guarantees.

Shared storage in reactive systems allow the same data to be accessed by different event handlers (e.g., cookies) and organizes the event handlers themselves (e.g., the DOM). These applications often include code from heterogeneous and untrusted sources and could potentially leak the users' sensitive data to an adversary. Most IFC approaches use the same enforcement mechanism for all components in an application. Given the heterogeneity of applications, a compositional enforcement mechanism where different

[1] This chapter is based on published work [59].

components execute under different IFC enforcement mechanisms could offer an attractive solution to the tradeoffs of each approach.

In this chapter, we motivate the usefulness of composition (Section 5.2), build a *framework* for composing different IFC enforcement mechanisms, and compare the security guarantees of different compositions. One of the challenges is to build a unified framework so that different styles of enforcement (taint tracking-based and multi-execution-based) can interact smoothly and two distinct elements of reactive systems (event handlers and shared storage) can interface nicely. To do so, our formalism identifies the common elements between all techniques, as well as the interfaces between the event handler execution component and the shared storage components and converts values between mechanisms securely.

As we show in Section 5.3, the compositional semantics are cleanly separated into a top-level component to trigger event handlers and a low-level component that executes individual event handlers and interacts with shared storage. In Section 5.4, we review knowledge-based security properties from previous chapters. We also define a weaker security condition which extends prior work on weak and explicit secrecy [95, 84] that permits the attacker to additionally learn what is implicitly leaked by taint tracking. We propose a set of security requirements that describe what is required by each system component and could be used to securely extend our framework with additional enforcement mechanisms in the future.

Contributions We develop a framework to enable the flexible composition of dynamic IFC enforcement mechanisms for reactive programs with provable security guarantees and model a simple web environment. We use a knowledge-based security condition to compare the relative security of different compositions. We extend prior work on weak secrecy to reason about implicit flows of information due to control flow decisions within as well as between event handlers and show that the overall security of a composed system may depend more on the security of the data structures shared between event handlers than the security of the event handler execution. Detailed definitions, lemmas, and proofs can be found in Appendix C.

5.2 Motivating Example

We demonstrate the usefulness of composing enforcement mechanisms via a web example in which event handlers run under different enforcement mechanisms.

Consider a website with a sign-up form including username and password fields and a submit button. There is also a third-party password strength-checking script which registers an event handler to the password field for the `onInput` event. The event handler is triggered whenever the user changes the

```
1 onClick( ) {
2   if (strength > 5) {
3      p = pwdNode.value;
4      u = unameNode.value;
5      output (H, u+p); }};
```

Listing 5.1: Event handler to send username and password to host server

```
1 onInput(e) {
2   p = e.target.value; /* get password value */
3   if (p.match(/[0−9]/)) { strength+=1; }
4   if (p.match(/[A−Z]/)) { strength+=4; }
5   ...
6   output (L, p); }; /* Explicit leak */
```

Listing 5.2: Third-party event handler to check password strength; "strength" is a global variable

password. It checks the password strength based on some algorithm (e.g., count the character classes of the password) and writes a numeric representation of the strength to a global variable *strength* (illustrated in Listing 5.2). The main page registers (among others) an event handler for the onClick event associated with the submit button (as shown in Listing 5.1). This event handler reads the global variable *strength*, and either allows the form submission (if the strength reaches a certain threshold) or displays a pop-up suggesting adding character classes, such as numbers and symbols.

The third-party script should compute the strength of the password locally without sending it on the network. A malicious script might try to send the password to their servers (line 6). The output command models sending a message to the third-party site. Let us see how taint tracking and multi-execution would enforce IFC in this scenario, and why composing them might be desirable.

Taint-tracking enforcement Suppose we execute the event handlers with a taint tracking enforcement mechanism. NSU would terminate the execution of the entire page if any script attempts to assign to a public variable in a secret branch. This effectively opens all pages to denial-of-service attacks, so we do not use NSU here (more discussion can be found in Section 5.5). Let's consider naive taint tracking [41] without NSU, instead. If the third-party checker tries to directly leak the password on line 6 in Listing 5.2, the output will be suppressed because the output requires the value's label to be lower than or equal to that of the channel, which does not hold.

A well-known limitation of naive taint tracking without NSU is that it allows the script to leak information via implicit flows [13]. Listing 5.3 is adapted from a classic example of implicit leaks. Here, the variable *detected_a* is only tainted if the first character is 'a'. In this case, the assignment on line 8 is not executed as the branch is not taken. As a result, *present_a* remains true (and labeled L). On the other hand, if the first character is not 'a', the assignment on line 6 will not be taken. Then, the condition

```
1 onInputLeak(e) {
2   p1 = e.target.value.charAt[0];
3   present_a = true; /* labeled L */
4   detected_a = false; /* labeled L */
5   if (p1 = 'a') {
6     detected_a = true;} /* tainted H if p1 is 'a' */
7   if (!detected_a) {
8     present_a = false;} /* still labeled L if p1 is 'a' */
9   output (L, present_a) };
```

Listing 5.3: Malicious third-party event handler

on line 8 will branch on an L value and therefore, *present_a* remains L and the value updated to false. Finally, the output on line 9 will successfully notify the attacker whether the first character is 'a'. We can expand the program to test the password character-by-character for every ASCII symbol and thus leak the entire password [9]. Thus, taint tracking has weaker security guarantees, which we later formalize using *weak secrecy* [95, 84], that allows attackers to learn which H branches are taken and which L variables are upgraded to H. Because we allow branch conditions to be leaked, anyway, we simplify our semantics by not upgrading the *pc* when branching on secrets.

Multi-execution enforcement To prevent the above-mentioned leaks, we can instead execute the event handlers using a multi-execution mechanism like SME [38]. The event handlers would then execute twice: once for the secret and once for the public level, where the secret execution would allow only H outputs while the public execution would allow only L outputs. The secret execution would see the actual value of the password, but the public execution would get a default value instead. If the script sends the password on an L channel (line 6 in Listing 5.2), the public execution would send the default value instead of the actual password, while the secret execution would skip the output altogether. This also prevents the implicit leaks shown in Listing 5.3. Although SME securely computes accurate information, it runs the event handlers multiple times and stores multiple copies of data, which is resource intensive.

Composing taint-tracking and multi-execution In this example, a desirable approach would be to execute the third-party script and store the global variable *strength* using a multi-execution approach so that it can correctly compute the strength of the password without compromising its secrecy. Meanwhile the event handlers on the main page could execute with a taint tracking mechanism as they do not purposely exploit implicit leaks and will be more performant than a multi-execution approach. In this example, for the main page event handlers to access precise information, the event handler will run in the H context to access the H copy of *strength*. Composition allows us to balance good security for the untrusted third-party scripts with good performance for the more trustworthy first-party scripts.

Execution contexts:

Compositional label set:	$\mathcal{L}$	$::=$	$\{\cdot, L, H\}$
Compositional program counter:	pc	$\in$	$\mathcal{L}$
Security label:	l	$\in$	$\{L, H\}$
Security policy:	$\mathcal{P}$		
Declassification policy:	$\mathcal{D}$		
Declassification module:	$\mathcal{R}$		
Declassification channel:	d		
Global storage enforcement ID set:	G	$::=$	$\mathcal{G}_{EH}, \mathcal{G}_g$
Global storage enforcement ID:	$\mathcal{G}$	$\in$	$\{\mathsf{SMS}, \mathsf{FS}, \mathsf{TS}\}$
EH enforcement ID:	$\mathcal{V}$	$\in$	$\{\mathsf{SME}, \mathsf{MF}, \mathsf{TT}\}$

Program syntax:

Value:	v	$::=$	$n \mid b \mid \mathsf{dv}$
Expression:	e	$::=$	$x \mid v \mid \mathsf{uop}\, e \mid e_1\, \mathsf{bop}\, e_2 \mid \mathsf{ehAPIe}(...)$
Command:	c	$::=$	$\mathsf{skip} \mid c_1; c_2 \mid \mathsf{while}\ e\ \mathsf{do}\ c \mid x := e \mid id := e \mid \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid x := \mathsf{declassify}(\iota, e)$
		$\mid$	$\mathsf{output}\ ch\ e \mid \mathsf{ehAPIc}(...)$

Runtime configurations:

Global state:	$\sigma^{(\mathcal{G}_{EH}, \mathcal{G}_g)}$	$::=$	$\sigma_{EH}^{\mathcal{G}_{EH}}, \sigma_g^{\mathcal{G}_g}$
Local state:	$\sigma^{\mathcal{V}}$		
Execution state:	s	$::=$	$P \mid C$
Events:	E	$::=$	$\cdot \mid E, (id.\mathsf{Ev}(v), l)$
Configuration:	$\kappa^{\mathcal{V}}$	$::=$	$\sigma^{\mathcal{V}}, c, s, E$
Configuration stack:	ks	$::=$	$\cdot \mid (\mathcal{V}; \kappa^{\mathcal{V}}; pc) :: ks$
Compositional config.:	K^G	$::=$	$\mathcal{R}, d; \sigma^G; ks$
Actions:	α	$::=$	$\mathsf{in} \mid ch(v) \mid \bullet$

Figure 5.1: Syntax for the compositional framework

Another interesting composition question arising from this example is whether it is necessary to store shared variables (e.g., *strength*) twice as done with SME and MF or is it sufficient to merely taint the variables and execute the script with SME? In this example, when onInput runs, the L copy runs first and sets the imprecise value for *strength* based on the default value for the password. The H copy runs next and sets the precise value for *strength* based on the real password with label H, as it is written from the H execution context. Is this secure? We take the first steps to explore different ways of storing data and executing scripts (Section 5.3), as well as what type of security each composition achieves (Section 5.4).

5.3 Compositional Enforcement Framework

One of our observations is that the semantics of reactive programs necessitate a high-level event handling loop that processes inputs and outputs, leading to the high-level semantics of dynamic IFC enforcement for these programs behaving similarly, regardless of the mechanism (e.g., SME or taint tracking). We design a framework that is flexible enough to incorporate all of the dynamic enforcement techniques described in Chapter 2.3. We review the components from the reactive system (Chapter 2.1), each of which

$$\boxed{G, \mathcal{P} \vdash K \stackrel{\alpha}{\Longrightarrow} K'}$$

$$\frac{\begin{array}{c}\mathcal{P}(id.\mathsf{Ev}(v)) = H \\ \mathcal{D}(\rho, id.\mathsf{Ev}(v)) = (r, \mathsf{emp}, \rho') \qquad d' = \mathsf{update}(d, r) \qquad G, \mathcal{P}, \sigma \vdash \cdot; \mathsf{lookupEHAll}(id.ev(v)) \rightsquigarrow_H ks\end{array}}{G, \mathcal{P} \vdash (\rho, \mathcal{D}), d; \sigma; \cdot \stackrel{id.\mathsf{Ev}(v)}{\Longrightarrow} (\rho', \mathcal{D}), d'; \sigma; ks} \textsc{In-High}$$

$$\frac{\mathcal{P}(id.\mathsf{Ev}(v)) = H_\Delta \qquad G, \mathcal{P}, \sigma \vdash \cdot; \mathsf{lookupEHAll}(id.ev(v)) \rightsquigarrow_H ks}{G, \mathcal{P} \vdash \mathcal{R}, d; \sigma; \cdot \stackrel{id.\mathsf{Ev}(v)}{\Longrightarrow} \mathcal{R}, d; \sigma; ks} \textsc{In-High}\Delta$$

$$\frac{\begin{array}{c}\mathcal{P}(id.\mathsf{Ev}(v)) = H \qquad \mathcal{D}(\rho, id.\mathsf{Ev}(v)) = (r, v', \rho') \qquad v' \neq v \qquad d' = \mathsf{update}(d, r) \\ G, \mathcal{P}, \sigma \vdash \cdot; \mathsf{lookupEHAt}(id.ev(v')) \rightsquigarrow_L ks \qquad G, \mathcal{P}, \sigma \vdash ks; \mathsf{lookupEHAt}(id.ev(v)) \rightsquigarrow_H ks'\end{array}}{G, \mathcal{P} \vdash (\rho, \mathcal{D}), d; \sigma; \cdot \stackrel{id.\mathsf{Ev}(v)}{\Longrightarrow} (\rho', \mathcal{D}), d'; \sigma; ks'} \textsc{In-ReleaseDiff}$$

$$\frac{\begin{array}{c}\mathcal{P}(id.\mathsf{Ev}(v)) = H \\ \mathcal{D}(\rho, id.\mathsf{Ev}(v)) = (r, v, \rho') \qquad d' = \mathsf{update}(d, r) \qquad G, \mathcal{P}, \sigma \vdash \cdot; \mathsf{lookupEHAll}(id.ev(v)) \rightsquigarrow . \; ks\end{array}}{G, \mathcal{P} \vdash (\rho, \mathcal{D}), d; \sigma; \cdot \stackrel{id.\mathsf{Ev}(v)}{\Longrightarrow} (\rho', \mathcal{D}), d'; \sigma; ks} \textsc{In-ReleaseSame}$$

$$\frac{\mathcal{P}(id.\mathsf{Ev}(v)) = L \qquad G, \mathcal{P}, \sigma \vdash \cdot; \mathsf{lookupEHAll}(id.ev(v)) \rightsquigarrow . \; ks}{G, \mathcal{P} \vdash \mathcal{R}, d; \sigma; \cdot \stackrel{id.\mathsf{Ev}(v)}{\Longrightarrow} \mathcal{R}, d; \sigma; ks} \textsc{In-Low}$$

Figure 5.2: Semantics for processing inputs (user events).

$$\boxed{G, \mathcal{P} \vdash K \stackrel{\alpha}{\Longrightarrow} K'}$$

$$\frac{\begin{array}{c}\mathsf{producer}(\kappa) \\ G, \mathcal{P}, \mathcal{V}, d \vdash \sigma, \kappa \xrightarrow{ch(v)}_{pc} \sigma', ks' \qquad \mathsf{outCondition}_{\mathcal{V}}(\mathcal{P}, ch(v), pc) \qquad \alpha_l = \mathsf{output}(\mathcal{P}, ch(v), pc)\end{array}}{G, \mathcal{P} \vdash \mathcal{R}, d; \sigma; ((\mathcal{V}; \kappa; pc) :: ks) \stackrel{\alpha_l}{\Longrightarrow} \mathcal{R}, d; \sigma', ks' :: ks} \textsc{Out}$$

$$\frac{\mathsf{producer}(\kappa) \qquad G, \mathcal{P}, \mathcal{V}, d \vdash \sigma, \kappa \xrightarrow{ch(v)}_{pc} \sigma', ks' \qquad \neg\mathsf{outCondition}_{\mathcal{V}}(\mathcal{P}, ch(v), pc)}{G, \mathcal{P} \vdash \mathcal{R}, d; \sigma; ((\mathcal{V}; \kappa; pc) :: ks) \stackrel{(\bullet, pc)}{\Longrightarrow} \mathcal{R}, d; \sigma', ks' :: ks} \textsc{Out-Skip}$$

$$\frac{\mathsf{producer}(\kappa) \qquad G, \mathcal{P}, \mathcal{V}, d \vdash \sigma, \kappa \xrightarrow{\alpha}_{pc} \sigma', ks' \qquad \alpha \neq ch(v)}{G, \mathcal{P} \vdash \mathcal{R}, d; \sigma; ((\mathcal{V}; \kappa; pc) :: ks) \stackrel{(\alpha, pc)}{\Longrightarrow} \mathcal{R}, d; \sigma', ks' :: ks} \textsc{Out-Other}$$

$$\frac{\mathsf{consumer}(\kappa)}{G, \mathcal{P} \vdash \mathcal{R}, d; \sigma; ((\mathcal{V}; \kappa; pc) :: ks) \stackrel{(\bullet, pc)}{\Longrightarrow} \mathcal{R}, d; \sigma, ks} \textsc{Out-Next}$$

Figure 5.3: Semantics for performing outputs (communications on channels).

has its own semantics (Section 5.3.2). The topmost level of semantics is responsible for processing inputs and outputs and looking up event handlers. The next level manages the event handler queue, and another level describes how individual event handlers are run, according to the selected enforcement mechanism. Finally, the lowest level semantics are described in Section 5.3.3 determines how event handlers interact with shared storage (such as the DOM).

5.3.1 Syntax

The syntax for our compositional enforcement framework is shown in Figure 5.1. We organize our security labels, l, in a three-point security lattice which is the standard two-point security lattice with an additional label '$\cdot$'. At a high-level, $\cdot$ means "no (pc) context" and is neither public nor private, so we put it at the bottom of the security lattice. This is used by MF to differentiate a standard execution from one which has split into an L and H copy. The program context label indicates the context under which the event handlers execute, denoted as pc.

Like previous chapters, the policy context $\mathcal{P}$ keeps track of the labels assigned to input events and output channels except here it also decides which event handlers run with which enforcement mechanism. For example, $\mathcal{P}$ might mark the onInput event for the password field as secret (H), output channels that belong to an attacker as public (L), and the enforcement of the onClick event handler to be TT and the third-party onInputCk event handler to be SME. We use the same stateful declassification as Chapters 3 and 4. We discuss considerations for making such decisions in Section 5.5. The enforcement for an event handler is denoted $\mathcal{V}$. We include SME (secure multi-execution), MF (faceted execution), and TT (a simple version of taint tracking without NSU semantics). The enforcement for the global store is denoted $G = (\mathcal{G}_{EH}, \mathcal{G}_g)$ where $\mathcal{G}_{EH}$ tells us how the event handler storage is enforced, and $\mathcal{G}_g$ tells us how the shared variables are stored. This permits more flexibility for the event handlers to be stored differently from the variables. Our shared storage techniques reflect our event handler enforcement: SMS (secure multi-storage maintains a separate copy of the store for each security level), FS (faceted storage maintains multiple copies only when necessary), and TS (tainted storage tracks labels for each item in the store).

Values include integers (n), booleans (b), and a pre-determined default value dv, which is used to replace the public copy of private data in multi-execution [44]. Each value type can have a distinct default value; for simplicity we use a single default value. Commands and expressions are mostly standard in our framework. The event handler APIs ehAPIe (e.g., look up a DOM node's attribute) and ehAPIc (e.g., create a new child node in the DOM) interact with the event handler store, and $id := e$ updates the attributes in the event handler store.

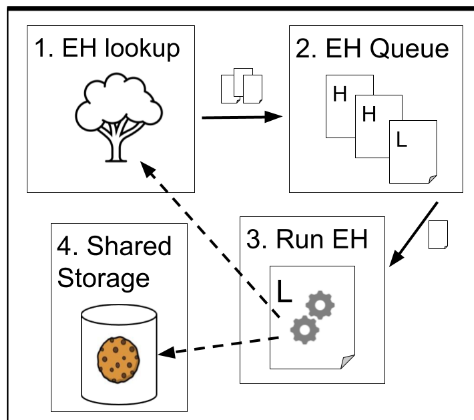


Figure 5.4: Reactive system described in Chapter 2.1.

A single configuration κ contains a local store for storing local script variables, $\sigma^{\mathcal{V}}$ (whose structure is determined by the enforcement mechanism $\mathcal{V}$), the command (event handler) being executed, the state of the execution (either Producer (P) or Consumer (C)), and a list of events triggered by that event handler, E. The compositional configuration K^G is a snapshot of the current system state. It maintains the global store σ^G and the configuration stack, ks. The global store includes variables shared between scripts and event handler storage. The structure of the global store depends on the enforcement mechanism. Each element of the configuration stack includes one of the event handlers pending execution in κ, as well as the enforcement mechanism it should run under, $\mathcal{V}$, and the context in which it should run, pc. The enforcement ($\mathcal{V}$) used for each event handler in the stack is determined by $\mathcal{P}$ and may be different for different events. Actions emitted by the execution, α, include user-generated input events, outputs on channels and silent actions, denoted $\bullet$.

5.3.2 Framework Semantics

The illustration from Chapter 2.1 is shown here in Figure 5.4 for easy reference. We organize our semantics into several layers to match the components of a reactive system including: (1) input/output processing and event handler lookup, (2) processing the event handler queue, (3) running individual event handlers, and (4) interactions with shared storage.

Input/Output, EH Lookup The top-most level for our compositional framework processes user input events and outputs to channels. These rules govern how inputs trigger event handlers and how outputs are processed and use the judgement $G, \mathcal{P} \vdash K \xRightarrow{\alpha} K'$, meaning the compositional configuration K can step to K' given input α or producing output α under the compositional enforcement G and label context $\mathcal{P}$. Input rules are shown in Figure 5.2 and output rules in Figure 5.3.

Regardless of how the event handlers or global variables are stored, or how the policy determines to enforce IFC on individual event handlers, the logic for looking up event handlers is the same. In each case, the label context, $\mathcal{P}$, tells us whether the event is secret (H) or public (L) and the declassification policy $\mathcal{D}$ tells us whether we should declassify. The EH lookup semantics, given by the judgement $G, \mathcal{P}, \sigma \vdash ks; \mathsf{lookupEH}(...) \rightsquigarrow_{pc} ks'$ return the stack of event handlers to run.

The label of an input event $id.\mathsf{Ev}(v)$ is given by the policy $\mathcal{P}$. For secret events which are not declassified (rules In-High and In-HighΔ), all event handlers visible to H are run in the H context by using $\mathsf{lookupEHAll}$ with $pc = H$ to build ks. When the input is a public event (rule In-Low), all event handlers are run in whatever context they are visible in by using the $\cdot$ pc for the lookup. When the event is declassified, the process of looking up event handlers depends on whether the released event receives the same argument as the original event. If the argument is the same (e.g., for policy "keypresses may be declassified, including *which* key was pressed"), event handler lookup is the same as for public events (rule In-ReleaseSame). If the argument is different (e.g., for policy "keypresses may be declassified, but *which* key was pressed should remain secret"), the event handlers with label H are looked up with the original argument, and event handlers with label L are looked up with the released argument.[2]

Similar to the input rules, the output rules shown in Figure 5.3 are the same regardless of the enforcement mechanism or event handler storage. The mid-level semantics are of the form: $G, \mathcal{P}, \mathcal{V} \vdash \sigma_1^G, \kappa \xrightarrow{\alpha}_{pc} \sigma_2^G, ks$ and run a single event handler κ with the given enforcement mechanism $\mathcal{V}$ and produce some output α. $\mathsf{producer}(\kappa)$ and $\mathsf{consumer}(\kappa)$ tell us whether the execution state of κ is producer or consumer (respectively). When an event handler is currently running, the system is in producer state (Out, Out-Skip, and Out-Other) and when the event handler has finished, the system is in consumer state (Out-Next) and the current event handler can be popped off ks. $\mathsf{out}_{\mathcal{V}}(...)$ determines if an output should be allowed (Out) or suppressed (Out-Skip) which is determined by whether the value being output is visible to the channel receiving the output and varies depending on the enforcement mechanism ($\mathcal{V}$). If the system is emitting a silent action (anything which is not an output to a channel), rule Out-Other applies.

EH Queue The mid-level semantics control the execution state (P for Producer, when an event handler is running, C for consumer, when it has finished) and adds event handlers for locally-triggered events (i.e., not triggered by a user) to the resulting configuration stack. After an event handler finishes running, these semantics check for any locally-triggered events. If there are some, their corresponding event handlers are added to ks. Finally, the current event handler enters consumer state to tell Out-Next to run the next event handler. The rules are shown in Figure 5.5 and are similar to the ones from Chapters 3 and 4.

[2]Note that event handlers with label $\cdot$ are returned for $\mathsf{lookupEHAt}$ for both $pc = L$ and $pc = H$.

$$\boxed{G, \mathcal{P}, \mathcal{V}, d \Vdash \sigma^G, \kappa^{\mathcal{V}} \xrightarrow{\alpha}_{pc} \sigma_2^G, ks}$$

$$\frac{E \neq \cdot \qquad G, \mathcal{P}, \mathcal{V}, \sigma^G \vdash (\mathcal{V}; (\sigma, \mathsf{skip}, C, \cdot); pc); \mathsf{lookupEHs}(E) \rightsquigarrow_{pc} ks}{G, \mathcal{P}, \mathcal{V}, d \vdash \sigma^G, \sigma, \mathsf{skip}, P, E \xrightarrow{\bullet}_{pc} \sigma^G, ks} \text{ LC}$$

$$\frac{}{G, \mathcal{P}, \mathcal{V}, d \vdash \sigma^G, \sigma, \mathsf{skip}, P, \cdot \xrightarrow{\bullet}_{pc} \sigma^G, (\mathcal{V}; (\sigma, \mathsf{skip}, C, \cdot); pc)} \text{ ProC}$$

$$\frac{G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1, c_1^{\mathsf{std}} \xrightarrow{\alpha}_{pc} \sigma_2^G, \sigma_2, c_2^{\mathsf{std}}, E_2}{G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_1^G, \sigma_1, c_1^{\mathsf{std}}, P, E_1 \xrightarrow{\alpha}_{pc} \sigma_2^G, (\mathcal{V}; (\sigma_2, c_2^{\mathsf{std}}, P, (E_1, E_2)); pc)} \text{ P}$$

Figure 5.5: Semantics for managing the event handler queue.

Running EHs The lower-level semantic rules for evaluating individual event handlers are triggered by the mid-level semantics in the "producer" state. These rules are mostly standard and enforcement-independent, except for interactions with the store. The rules in Figure 5.6 highlight the way our framework handles these differences. ASSIGN-G performs an assignment to a global variable while ASSIGN-D performs an assignment to an attribute in the event handler storage. Expressions are evaluated using the judgment $G, \mathcal{V}, \sigma^G, \sigma^{\mathcal{V}} \vdash e \Downarrow_{pc} v$. This also ensures v is in the format expected by the enforcement when different mechanisms are composed. For instance, to convert a tainted value (v, H) to a value used by SME, we check that the label on the value is visible to the execution. The L execution would receive the default value dv instead of something tainted (v, H), while the H execution would receive the real value. This is reminiscent of the way SME replaces secret inputs with dv for the L execution. More discussion on conversion can be found in Section 5.3.3.

The assignment is performed using enforcement-specific helper functions. $\mathsf{assign}_G(...)$ assigns global variables or event handler attributes, depending on whether a variable or node *id* is passed as an argument. The *pc* ensures that the assignments are performed securely (i.e., in the correct copy of the store, facet, or with the correct label, depending on the type of enforcement).

5.3.3 Shared storage

Event handlers may interact with each other through shared storage. To introduce the storage techniques, we describe the syntax for both variable and event handler storage (using the DOM as a case study) and describe their semantics at a high-level, then we explain how shared storage with one type of enforcement may be composed with an event handler running with a different type of enforcement. Finally, we illustrate these interactions by returning to our example from Section 5.2.

$$\boxed{G, \mathcal{V} \Vdash \sigma_1^G, \sigma_1^{\mathcal{V}}, c_1 \xrightarrow{\alpha}_{pc} \sigma_2^G, \sigma_2^{\mathcal{V}}, c_2, E}$$

$$\frac{G, \mathcal{V}, \sigma_1^G, \sigma_1^{\mathcal{V}} \vdash e \Downarrow_{pc} v \qquad x \in \sigma_1^G \qquad \mathsf{assign}_G(\sigma_1^G, pc, x, v) = \sigma_2^G}{G, \mathcal{V} \Vdash \sigma_1^G, \sigma_1^{\mathcal{V}}, x := e \xrightarrow{\bullet}_{pc} \sigma_2^G, \sigma_1^{\mathcal{V}}, \mathsf{skip}, \cdot} \text{ ASSIGN-G}$$

$$\frac{G, \mathcal{V}, \sigma_1^G, \sigma_1^{\mathcal{V}} \vdash e \Downarrow_{pc} v \qquad x \notin \sigma_1^G \qquad \mathsf{assign}_{\mathcal{V}}(\sigma_1^{\mathcal{V}}, pc, x, v) = \sigma_2^{\mathcal{V}}}{G, \mathcal{V} \Vdash \sigma_1^G, \sigma_1^{\mathcal{V}}, x := e \xrightarrow{\bullet}_{pc} \sigma_1^G, \sigma_2^{\mathcal{V}}, \mathsf{skip}, \cdot} \text{ ASSIGN-L}$$

$$\frac{G, \mathcal{V}, \sigma^G, \sigma^{\mathcal{V}} \vdash e \Downarrow_{pc} v}{G, \mathcal{V}, d \Vdash \sigma^G, \sigma^{\mathcal{V}}, \mathsf{output}\ ch\ e \xrightarrow{ch(v)}_{pc} \sigma^G, \sigma^{\mathcal{V}}, \mathsf{skip}, \cdot} \text{ OUTPUT}$$

Figure 5.6: Selected command semantics

Shared Storage

Shared storage: $\sigma^{(\mathcal{G}_{EH}, \mathcal{G}_g)}$	$::=$	$\sigma_g^{\mathcal{G}_g}, \sigma_{EH}^{\mathcal{G}_{EH}}$

SME/SMS **Variable Storage**

Single store:	σ_{pc}	$::=$	$\cdot \mid x \mapsto v$
SME/SMS *Storage:*	$\sigma^{\mathsf{SME}}, \sigma_g^{\mathsf{SMS}}$	$::=$	σ_H, σ_L

MF/FS **Variable Storage**

Faceted value:	$v^{\mathsf{MF}}, v^{\mathsf{FS}}$	$::=$	$v \mid \langle v_H \mid v_L \rangle \mid \langle \cdot \mid v \rangle \mid \langle v \mid \cdot \rangle$
MF *Storage:*	σ^{MF}	$::=$	$\cdot \mid \sigma^{\mathsf{MF}}, x \mapsto v^{\mathsf{MF}}$
FS *Storage:*	σ_g^{FS}	$::=$	$\cdot \mid \sigma_g^{\mathsf{FS}}, x \mapsto v^{\mathsf{FS}}$

TT/TS **Variable Storage**

Labeled value:	$v^{\mathsf{TT}}, v^{\mathsf{TS}}$	$::=$	(v, l)
TT *Storage:*	σ^{TT}	$::=$	$\cdot \mid \sigma^{\mathsf{TT}}, x \mapsto v^{\mathsf{TT}}$
TS *Storage:*	σ_g^{TS}	$::=$	$\cdot \mid \sigma_g^{\mathsf{TS}}, x \mapsto v^{\mathsf{TS}}$

Figure 5.7: Storage syntax

Variable storage syntax We refer to shared storage techniques using similar terms as the enforcement mechanisms for code execution: secure multi-storage, SMS, stores each item multiple times (once per security level), faceted storage, FS, stores multiple copies only when necessary, and tainted storage, TS, tracks labels for every item in the store. Storage syntax is shown in Figure 5.7. For SME/SMS, variables are stored twice: once at each security level. Observers at H will interact with the H copy of the store (σ_H) and observers at L with interact with the L copy of the store (σ_L). For MF/FS, variables are also stored twice, but only when the value depends on a secret. A faceted value such as $\langle v_H | v_L \rangle$ depends on a secret. H observers will interact with the H facet (v_H) and L observers interact with the L facet (v_L). Empty facets (such as the L facet of $\langle v | \cdot \rangle$) are treated as a default value and arise when a variable has been initialized in

EH Storage:

EH map: $M \quad ::= \quad \cdot \mid M, \mathsf{Ev} \mapsto \{(eh_1, l_1), ..., (eh_n, l_n)\}$

Unstructured SMS **DOM:**

Single store: $\sigma_{pc} \quad ::= \quad \cdot \mid id \mapsto (v, M)$

DOM: $\sigma_{EH}^{\mathsf{SMS}} \quad ::= \quad \sigma_H, \sigma_L$

Unstructured FS **DOM:**

DOM: $\sigma_{EH}^{\mathsf{FS}} \quad ::= \quad \cdot \mid \sigma^{\mathsf{FS}}, id \mapsto (v^{\mathsf{FS}}, M)$

Unstructured TS **DOM:**

DOM: $\sigma_{EH}^{\mathsf{TS}} \quad ::= \quad \cdot \mid \sigma^{\mathsf{TS}}, id \mapsto (v^{\mathsf{TS}}, M, l)$

DOM addresses:

Location: $loc \quad \in \quad \mathsf{Address}$

Address: $a \quad ::= \quad loc \mid \mathsf{NULL}$

Root address: $a^{\mathsf{rt}} \quad ::= \quad loc$

Address list: $A \quad ::= \quad \cdot \mid A, a$

Tree-structured SMS **DOM:**

Node: $\phi^{\mathsf{SMS}} \quad ::= \quad (id, v, M, a_p, A)$

Single store: $\sigma_{pc} \quad ::= \quad a^{\mathsf{rt}} \mapsto \phi^{\mathsf{SMS}} \mid \sigma_{pc}, loc \mapsto \phi^{\mathsf{SMS}}$

DOM: $\sigma_{EH}^{\mathsf{SMS}} \quad ::= \quad \sigma_H, \sigma_L$

Tree-structured FS **DOM:**

Faceted address: $a^{\mathsf{FS}} \quad ::= \quad a \mid \langle a_H | a_L \rangle \mid \langle \cdot | a \rangle \mid \langle a | \cdot \rangle$

Faceted address list: $A^{\mathsf{FS}} \quad ::= \quad \cdot \mid a^{\mathsf{FS}} :: A^{\mathsf{FS}}$

Node: $\phi^{\mathsf{FS}} \quad ::= \quad (id, v^{\mathsf{FS}}, M, a_p^{\mathsf{FS}}, A^{\mathsf{FS}})$

DOM: $\sigma_{EH}^{\mathsf{FS}} \quad ::= \quad a^{\mathsf{rt}} \mapsto \phi^{\mathsf{FS}} \mid \sigma^{\mathsf{FS}}, loc \mapsto \phi^{\mathsf{FS}}$

Figure 5.8: Event handler storage syntax for the DOM

one context (L or H) but not the other.[3] Finally, for TT/TS, values have an accompanying label to reflect whether they have been influenced by a secret (label H) or not (label L).

EH storage syntax Event handler storage associates events with the appropriate event handlers. The DOM is one type of event handler storage, which links event handlers to elements on a webpage. We explain how to model event handler storage in our framework by considering both an unstructured DOM, where nodes are organized as an unordered list (similar to the simple DOM model from Chapters 3 and 4), which is useful for reactive systems like OS processes, as well as a more traditional tree-structure [76], which is useful for modeling the DOM. For brevity, we refer both the unstructured and tree-structured event handler storage as the "DOM." The syntax for both structures are shown in Figure 5.8.

In the unstructured DOM, elements are identified by a unique identifier (*id*) and contain both an attribute (whose structure is determined by the type of enforcement, to be described next) and an event handler map (M), which maps events (Ev) to a list of event handlers (*eh*) and the context they were

[3] Note that empty facets are important not only for security, but also for precision: if a variable is initialized in the L context but not the H context, it would not leak anything to store v instead of $\langle \cdot | v \rangle$, but it would be less precise.

registered in (l). M is the same for all enforcement mechanisms, except that event handlers in FS may have any label in $\mathcal{L}$ ("$\cdot$" means the event handler can be triggered by either L- or H-labeled events) but SMS and TS event handlers may only be labeled L or H.

Similar to variable storage, the unstructured SMS DOM has two copies. H observers interact with the H copy of the DOM and likewise for L observers. Attributes are standard values (v), including integers and booleans. Initially, the H and L copies of the DOM will be identical. As events are triggered, new elements may be added to the DOM, event handlers registered, or attributes updated in one or both copies. The unstructured FS DOM is a single structure whose attributes are duplicated when they have been influenced by secrets. Here, attributes are standard when the value does not depend on a secret (v) or faceted values when the value appears different to H observers than L observers ($\langle v_H | v_L \rangle$). Initially, all the attributes are standard values in the FS DOM. A DOM element which has been added in only the H context will have an attribute with an empty L facet (i.e., $\langle v | \cdot \rangle$) and likewise for the H facet of an element added in only the L context. The TS DOM will associate labels with both attributes ((v, l)) and DOM elements ((v^{TS}, M, l)). The label on the element reflects the context the element was created in, while the label on the attribute reflects whether the attribute has been influenced by a secret ($l = H$) or not ($l = L$).

In the tree-structured [76] DOM, each element on the page has a matching DOM node (ϕ) which is stored by reference (*loc*). Nodes have a unique identifier (*id*), an attribute, and an event handler map, like in the unstructured DOM. They also contain a pointer to their parent (a_p), and a list of pointers to their children (A) (if any). The root of the DOM is at a^{rt}. The node at this address cannot be replaced with another node, but its attribute may be updated and children can be added to it. Since we later prove that compositions involving the unstructured TS DOM only satisfy weak secrecy, we only formalize the more complex tree-structured DOM for SMS and FS.

The tree-structured SMS DOM has two copies and behaves similarly to the unstructured SMS DOM. The tree-structured FS DOM supports faceted attributes, as well as a faceted parent pointer (a^{FS}) and list of faceted pointers to children (A^{FS}). Because nodes are uniquely identified by their ID, a node may have a faceted parent pointer, for instance, if a node is created as a child of ϕ_H in the H context and then a node with the same ID is created as a child of ϕ_L in the L context. A node might have a faceted pointer in its list of children if a child is added in the H context, but not the L context. In this case, if the child is at address a, the node would have $\langle a | \cdot \rangle$ in its list of children.

Storage composition Since different event handlers running with different enforcement mechanisms may interact through shared storage, values may need to be "converted" from the format for one enforcement mechanism (i.e., a standard, faceted, or labeled value) to another. When converting data, we follow

		Destination and *pc*								
		SME, SMS			MF, FS			TT, TS		
		$\cdot$	L	H	$\cdot$	L	H	$\cdot$	L	H
Source	v^{std}	v^{std}	v^{std}	v^{std}	v^{std}	$\langle\cdot\|v^{\text{std}}\rangle$	$\langle v^{\text{std}}\|\cdot\rangle$	(v^{std}, L)	(v^{std}, L)	(v^{std}, H)
	$\langle v_H\|v_L\rangle$	$\langle v_H\|v_L\rangle$	v_L	v_H	$\langle v_H\|v_L\rangle$	v_L	v_H	$\langle v_H\|v_L\rangle$	(v_L, L)	(v_H, H)
	$\langle v\|\cdot\rangle$	$\langle v\|\cdot\rangle$	dv	v	$\langle v\|\cdot\rangle$	dv	v	$\langle v\|\cdot\rangle$	(dv, L)	(v, H)
	$\langle\cdot\|v\rangle$	$\langle\cdot\|v\rangle$	v	dv	$\langle\cdot\|v\rangle$	v	dv	$\langle\cdot\|v\rangle$	(v, L)	(dv, H)
	(v, L)	–	v	v	v	v	v	–	(v, L)	(v, H)
	(v, H)	–	dv	v	$\langle v\|\text{dv}\rangle$	dv	v	–	(v, H)	(v, H)

Table 5.1: Conversion between standard, tainted, and faceted values.

three high-level guidelines to ensure the composition is secure:

1. The *pc* context determines which copy to access in multi-storage. If a value is coming from SMS or FS, there may be two copies to pick from. When the context (i.e., the *pc*) is H, we access the H copy, and likewise for L. If the value does not exist in that copy of the store (in the case of SMS) or is an empty facet (in the case of FS), we use a default value.

2. The *pc* context and destination determine whether to replace a labeled value with a default value. If the value is coming from TS, we need to decide if we take the actual value or use a default value. If the context is H, we take the real value without leaking any information. If the context is L and the destination is a multi-storage (SMS, FS) or multi-execution (SME, MF) technique, we replace tainted values (with label H) with a default value since the L copy of the store/execution should never be influenced by a secret. On the other hand, if the destination is TS or TT, we use the original, tainted value, and propagate the taint through the resulting label.

3. The destination and *pc* context determines the ultimate format. Multi-storage and multi-execution techniques use the context to determine which copy of the store/which facet to update. For taint tracking techniques, the context is also used to determine the final label on the data (e.g., public data is labeled H if it is computed in the H context). Consider a public event handler running with SME. It would run first in the L context and then in the H context. The L execution would interact with the L copy of store secured with SMS, or with the L facets for a store secured with FS. The H execution would interact with the H copy (respectively, H facets). On the other hand, if the store is secured with TS, any changes made by the L execution would be labeled L and ultimately be overwritten by the H execution (which would have label H). A table summarizing how data is converted for every combination of enforcement is shown in Table 5.1.

Examples We describe how the example from Section 5.2 works in our framework, using the configuration in Figure 5.9. For illustrative purposes, we describe both SMS and TS shared storage with an

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textsf{TS}\ \textbf{Shared storage}} \\
\sigma_g & = & \textit{strength} \mapsto (40, H), \\
 & & \quad \textit{username} \mapsto (\text{“bob”}, L) \\
\sigma_{EH} & = & (id_p \mapsto ((\text{“aKUd?mdu5GHa\&l7gHJ5”}, H), \textsf{input} \mapsto \{(\textsf{onInput}(x)\{c_{in}\}, L)\}, L)), \\
 & & \quad (id_b \mapsto (_, \textsf{click} \mapsto \{(\textsf{onClick}(\ )\{c_{clk}\}, L)\}, H)) \\
\multicolumn{3}{l}{\textsf{SMS}\ \textbf{Shared storage}} \\
\sigma_{g,L} & = & \textit{strength} \mapsto \textsf{dv}, \\
 & & \quad \textit{username} \mapsto \text{“bob”} \\
\sigma_{EH,L} & = & (id_p \mapsto \mapsto (\textsf{dv}, \textsf{input} \mapsto \{(\textsf{onInput}(x)\{c_{in}\}, L)\})), \\
 & & \quad (id_b \mapsto (_, \cdot)) \\
\sigma_{g,H} & = & \textit{strength} \mapsto 40, \\
 & & \quad \textit{username} \mapsto \text{“bob”} \\
\sigma_{EH,H} & = & (id_p \mapsto (\text{“aKUd?mdu5GHa\&l7gHJ5”}, \textsf{input} \mapsto \{(\textsf{onInput}(x)\{c_{in}\}, H)\}), \\
 & & \quad (id_b \mapsto (_, \textsf{click} \mapsto \{(\textsf{onClick}(\ )\{c_{clk}\}, H)\})) \\
 & & \\
\multicolumn{3}{l}{\textbf{Configuration stack}} \\
ks & = & (\textsf{SME}, ((p1 \mapsto \textsf{dv}[0], \textit{present}_a \mapsto \textsf{false}, \textit{detected}_a \mapsto \textsf{false}), [id_p/x]c_{in}, P, \cdot), L) \\
 & & \quad :: (\textsf{SME}, ((p1 \mapsto \text{“a”}, \textit{present}_a \mapsto \textsf{true}, \textit{detected}_a \mapsto \textsf{true}), [id_p/x]c_{in}, P, \cdot), H) \\
 & & \quad :: (\textsf{TT}, ((p \mapsto (\text{“aKUd?mdu5GHa\&l7gHJ5”}, H), u \mapsto (\text{“bob”}, L)), c_{clk}, P, \cdot), H)
\end{array}
$$

Figure 5.9: Example configuration

unstructured DOM.

For TS storage, everything maps to a value and a label, including both variables and attributes and elements in the DOM. SMS involves an H and L copy of both the shared variables and DOM. The onInput event handler is public, so it exists in both the H and L copies of the SMS event handler storage and is labeled L in the TS storage. The contents of the field id_p are secret, so for SMS, the contents are replaced with a default value in the L copy of the DOM, and for TS the contents are labeled H. The onClick event is secret, so it is only registered in the H copy of the SMS DOM and is labeled H in the TS DOM. The policy is that onInput event handlers should be run under SME. We trust the first-party event handler onClick to not misbehave, so the policy is to run this event handler with TT.

The *ks* in Figure 5.9 is the result of looking up event handlers for the input event on the password field and the public click event on the “Submit” button. Note that *ks* will be the same whether we use SMS or TS for shared storage (more details on this to follow). For illustrative purposes, *ks* is the *result* of running all three event handlers. In reality, the local stores would initially be empty and the input event handlers would run to completion before the click event was triggered.

Rule In-L is used to process the public Input event. It will run all the registered event handlers in whatever context they are visible. Since the event handler is registered in both the L and H copies of the SMS DOM, and with label L in the TS DOM, it is visible to both the L and H context. Since we are running this event handler with SME, the *ks* has two onInput event handlers: one running in the L

context and one in the H context (note that SMS and TS produce the same *ks*).[4] The onInput event handler attempts to output to an L channel. In the H execution, this output is suppressed (Out-Skip) because the output condition for SME requires that the label on the channel matches the label of the *pc*. On the other hand, the same output in the L execution would succeed (Out). Recall that event handlers running in the L context interact with the L copy of the SMS DOM and receive default values instead of tainted values from the TS DOM. Therefore, this output does not leak anything to the attacker since the L copy of the execution receives a default value for the password from the DOM in both cases.

For the Click event, In-H runs all event handlers visible to H (i.e., only those labeled H). This is the third element in *ks* (note that, like above, SMS and TS produce the same *ks*). This event handler will run in the H context, so it will interact with the H copy of the SMS σ_g and runs the risk of upgrading public variables in the TS σ_g. In this case, Listing 5.1 only reads from the shared storage, so nothing is leaked through TS. Recall from above that everything from the H copy of the SMS storage will be labeled H, and everything that comes from the TS storage will keep its label. An output for TT succeeds if the *pc* (H) joined with the label on the value being output ($p + u$, so, $H \sqcup H = H$ in the case of SMS or $H \sqcup L = H$ in the case of TS) is at or below the label on the channel (H). Therefore, this output succeeds.

This example shows that our framework can seamlessly compose enforcement mechanisms and securely convert data between different enforcement mechanisms, like SMS and TT.

5.4 Security and Weak Secrecy

Next we present two security definitions of different strengths, compare these two definitions, and prove that the techniques from Chapter 2 may be composed to enforce varying levels of security.

5.4.1 Attacker Observation

Our definition for trace equivalence is similar to previous chapters, except here we also consider a more complex tree-structure for our DOM. To quantify how much an attacker learns by interacting with our framework, we first define what the attacker can observe from an execution trace. A trace T is a sequence of execution steps, inductively defined as $T = G, \mathcal{P} \vdash T' \xrightarrow{\alpha_l} K$ where an empty trace is the initial state $G, \mathcal{P} \vdash K_0$. An attacker's observation of T, denoted $T \downarrow_L$, is the sequence of L-observable inputs and outputs in T. Two execution traces are L-equivalent if their L observations are the same: $T \approx_L T'$ iff $T \downarrow_L = T' \downarrow_L$. The L observation of an execution trace is defined in Figure 5.10.

[4] If the same event handler is run under TT, we only want to run the event handler once to avoid duplicated outputs to H channels. There is a precision tradeoff between running the event handler only in the H context (suppress all the L outputs) or L context (may alter the H outputs, depending on the global storage technique). We choose to run the event handler in the L context only and leave further exploration of the effects on precision to future work.

$$\frac{}{(G, \mathcal{P} \vdash K) \downarrow_L = \cdot}\ \textsc{Trace-Base} \qquad \frac{\mathcal{P}(id.\mathsf{Ev}(v)) = L}{(G, \mathcal{P} \vdash K \overset{id.\mathsf{Ev}(v)}{\Longrightarrow} T') \downarrow_L = id.\mathsf{Ev}(v) :: T' \downarrow_L}\ \textsc{Trace-LowIn}$$

$$\frac{\mathcal{P}(\alpha) = L \quad \text{or} \quad l \sqsubseteq L}{(G, \mathcal{P} \vdash K \overset{(\alpha,l)}{\Longrightarrow} T') \downarrow_L = \alpha :: T' \downarrow_L}\ \textsc{Trace-Low}$$

$$\frac{\mathsf{rel}(K) = \mathcal{R} \qquad \mathcal{P}(id.ev(v)) = H \qquad \mathcal{R}(\mathcal{P}, id.ev(v)) = \alpha \neq \bullet}{(G, \mathcal{P} \vdash K \overset{id.ev(v)}{\Longrightarrow} T') \downarrow_L = \mathsf{rls}(\alpha) :: T' \downarrow_L}\ \textsc{Trace-HighRelease}$$

$$\frac{\mathcal{P}(id.ev(v)) = H_\Delta}{(G, \mathcal{P} \vdash K \overset{id.ev(v)}{\Longrightarrow} T') \downarrow_L = T' \downarrow_L}\ \textsc{Trace-High}\Delta$$

$$\frac{\mathsf{rel}(K) = \mathcal{R} \qquad \mathcal{P}(id.ev(v)) = H \wedge \mathcal{R}(\mathcal{P}, id.ev(v)) = \bullet}{(G, \mathcal{P} \vdash K \overset{id.ev(v)}{\Longrightarrow} T') \downarrow_L = T' \downarrow_L}\ \textsc{Trace-HighNoRelease}$$

$$\frac{\mathcal{P}(\alpha) = H \quad \text{or} \quad \alpha = \bullet}{(G, \mathcal{P} \vdash K \overset{(\alpha,H)}{\Longrightarrow} T') \downarrow_L = T' \downarrow_L}\ \textsc{Trace-High}$$

Figure 5.10: Rules for projecting execution traces to L

Low inputs (Trace-LowIn) and other low actions (Trace-Low) are observable. Rule Trace-Low defines a "low action" as one produced in the low context ($l \sqsubseteq L$) or an L-labeled action ($\mathcal{P}(\alpha) = L$). L-labeled inputs and outputs to L-labeled channels are all L-labeled actions. High inputs are not observable unless they are released. Inputs from dynamic elements are never declassified (Trace-HighΔ) and we use the shorthand $\mathcal{R}(\mathcal{P}, id.\mathsf{Ev}(v))$ to decide whether the H inputs were released. If $\mathcal{R}(\mathcal{P}, id.\mathsf{Ev}(v)) = \bullet$ (Trace-HighNoRelease), it means the input was not released by the declassification policy, while $\mathcal{R}(\mathcal{P}, id.\mathsf{Ev}(v)) = (\rho', r, \alpha)$ (Trace-HighRelease) means the event was declassified (ρ' is the new declassification state, r is the valued released on the declassification channel, and α is the declassified event, or $\bullet$ if only the state and/or channel are updated). Finally, secret actions ($\mathcal{P}(\alpha) = H$) performed in the H context are not observable as shown in Trace-H.

Two configurations K_1 and K_2 are L-equivalent if their global stores σ_1^G and σ_2^G and their configuration stacks ks_1 and ks_2 are L-equivalent. Configuration stacks are L-equivalent if all the L configurations have L-equivalent local stores and they agree on commands. Most of these definitions are straightforward. The most interesting definition is L-equivalence of the tree-structured DOM, which is defined inductively over the structure of the tree beginning with the root nodes.

5.4.2 Progress-Insensitive Security

We first define attacker's knowledge assuming that the attacker can view all the publicly-observable inputs and outputs, as well as the initial state of the system (this includes the initial global variables and DOM upon page load which contains no secrets). The attacker's knowledge given a trace T is *what they believe the secret inputs might have been*, which is the set of inputs from L-equivalent execution traces starting from the same initial state:

Definition 21 (Attacker Knowledge)**.** *An attacker's knowledge after observing T beginning from state σ_0 with security policy $\mathcal{P}$ and declassification module $\mathcal{R}$ denoted $\mathcal{K}(T, \sigma_0^G, \mathcal{P}, \mathcal{R})$ is defined as $\{\tau_i \mid \exists T' \in \mathsf{runs}(\sigma_0^G, \mathcal{P}, \mathcal{R}), T \approx_L T', \tau_i = \mathsf{in}(T')\}$*

We define $\mathsf{runs}(...)$ as the set of possible execution traces resulting from the shared state σ_0^G under the policy $\mathcal{P}, \mathcal{R}$. The set of inputs from a trace T is denoted $\mathsf{in}(T)$, while τ is a sequence of actions.

Intuitively, the system is secure if the attacker does not refine their knowledge. However, this definition is too strong for our system because it is progress sensitive. A possibly diverging loop that depends on a secret will allow the attacker to refine their knowledge based on whether the system makes progress to accept another low input. Instead, we define a weaker, progress-*in*sensitive security property, by introducing the following progress-insensitive attacker's knowledge below:

Definition 22 (Progress Knowledge)**.** *An attacker's knowledge after observing trace T beginning from state σ_0 with security policy $\mathcal{P}$ and declassification module $\mathcal{R}$, when they know the system will continue to make progress, denoted $\mathcal{K}_p(T, \sigma_0^G, \mathcal{P}, \mathcal{R})$ is defined as $\{\tau_i \mid \exists T' \in \mathsf{runs}(\sigma_0^G, \mathcal{P}, \mathcal{R}), T \approx_L^{\mathcal{P}} T', \tau_i = \mathsf{in}(T'), \mathsf{prog}(T')\}$*

The attacker is allowed to observe the progress behavior of traces, so we add $\mathsf{prog}(T')$ (defined the same as in Chapter 3.4.2) as a condition on T' to consider only the traces which produce the same L-observations *and* make progress.

We also allow the attacker's knowledge to be refined by declassification. To be precise about what is leaked by declassification, we define progress-insensitive knowledge with release:

Definition 23 (Release Knowledge)**.** *An attacker's knowledge after observing a trace T beginning from state σ_0 with security policy $\mathcal{P}$ and declassification module $\mathcal{R}$, which just produced the declassification α, denoted $\mathcal{K}_{rp}(T, \sigma_0^G, \mathcal{P}, \mathcal{R}, \alpha)$ is defined as $\{\tau_i \mid \exists T' \in \mathsf{runs}(\sigma_0^G, \mathcal{P}, \mathcal{R}), T \approx_L T', \tau_i = \mathsf{in}(T'), \mathsf{prog}(T'), \alpha' = (\mathsf{last}(T) \stackrel{\alpha}{\Longrightarrow} K)) \downarrow_L, \mathsf{releaseT}(T', \alpha'))\}$*

We add $\mathsf{releaseT}$ as a condition on T' to only consider the traces which produce the same L-observations, make progress, *and* produce the same declassification.

Using these knowledge definitions, we define what it means for a program to be secure: when the system takes a step, the attacker's *confidence* about the secret inputs should not increase; they should not be able to distinguish between any more traces than before, other than through whether the system makes progress and what is declassified. We use $\preceq$ subscript for the subset relation ($\supseteq_\preceq$) to say that the input sequences after the step may be longer and $\mathsf{rlsA}(...)$ holds if the step was a declassification.

Definition 24 (Progress-Insensitive Security). *The compositional framework is progress-insensitive secure iff given any initial global store σ_0^G and release policy $\mathcal{R}, \mathcal{P}$, it is the case that for all traces T, actions α, and configurations K s.t. $(G, \mathcal{P} \vdash T \stackrel{\alpha}{\Longrightarrow} K) \in \mathsf{runs}(\sigma_0^G, \mathcal{P}, \mathcal{R})$, then, the following holds*

- *If* $\mathsf{rlsA}(G, \mathcal{P} \vdash \mathsf{last}(T) \stackrel{\alpha}{\Longrightarrow} K)$: $\mathcal{K}(G, \mathcal{P} \vdash T \stackrel{\alpha}{\Longrightarrow} K, \sigma_0^G, \mathcal{P}, \mathcal{R}) \supseteq_\preceq \mathcal{K}_{rp}(T, \sigma_0^G, \mathcal{P}, \mathcal{R}, \alpha)$
- *Otherwise:* $\mathcal{K}(G, \mathcal{P} \vdash T \stackrel{\alpha}{\Longrightarrow} K, \sigma_0^G, \mathcal{P}, \mathcal{R}) \supseteq_\preceq \mathcal{K}_p(T, \sigma_0^G, \mathcal{P}, \mathcal{R})$

We can prove that any combination of enforcement mechanisms SME, SMS, MF, and FS satisfy this progress-insensitive security condition:

Theorem 25 (Soundness). *If event handlers are enforced with $\mathcal{V} \in \{\mathsf{SME}, \mathsf{MF}\}$ and the global storage is enforced with $G \in \{\mathsf{SMS}, \mathsf{FS}\}$, then the composition of these event handlers and global stores in our framework satisfies progress-insensitive security.*

We prove our framework secure with these enforcement mechanisms by defining a series of "requirements" for the framework (called *Trace* and *Expression* requirements), variable (called *Variable* requirements), and event handler stores (called *Event Handler* requirements). These requirements are described in Table 5.2. Overall, these requirements follow a similar structure to other knowledge-based security proofs from prior work. The most noteworthy difference is the notion of "strong equivalence" for values. Traditionally, noninterference only requires that values are equivalent (i.e., they are the same public values, *or* both values are secret) but here we require that values are both equivalent and publicly observable (i.e., they are equivalent only if they are the same public values; they cannot be tainted). This distinction is important for highlighting the difference between progress-insensitive security and weak secrecy.

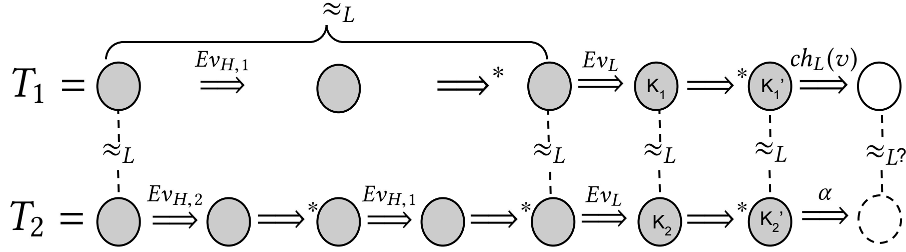
5.4.3 Weak Secrecy

As discussed in Section 5.2, NSU semantics are too rigid for our setting. Unfortunately, without NSU semantics, taint tracking techniques are susceptible to implicit leaks. Namely, branching on a secret in the L context may result in different public behavior for different secrets. We can also see implicit leaks through global store: suppose a secret event handler *upgrades* a public value stored in the global variable

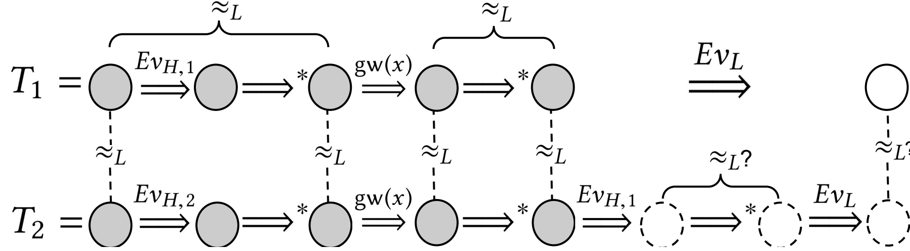
Trace Requirements		
(W)T1	$\approx_L$ traces, $\approx_L$ states	Equivalent traces starting in equivalent states lead to equivalent states
(W)T2	Empty traces, $\approx_L$ states	Traces producing no public events produce equivalent states
T3	Secret *pc*'s, empty traces	Steps under a secret *pc* produce no public events
(W)T4	Strong one-step	If a trace takes a step, then an equivalent trace can take an equivalent step
(W)T5	Weak one-step	Equivalent traces taking steps producing equivalent public observations lead to equivalent states
Expression Requirements		
E1	L-expressions are $\approx_L$	Evaluating an expression under equivalent stores with public *pc*'s results in *strong* equivalent values
WE1	L-expressions are $\approx_L$	Evaluating an expression under equivalent stores with public *pc*'s results in equivalent values
Variable Requirements		
V1	L-lookups are $\approx_L$	Lookups of the same variable under public *pc*'s in equivalent stores result in *strong* equivalent values
WV1	L-lookups are $\approx_L$	Lookups of the same variable under public *pc*'s in equivalent stores result in equivalent values
(W)V2	H-assignments are $\approx_L$	Assignments to stores under a secret *pc* result in an equivalent store
(W)V3	L-assignments are $\approx_L$	Assignments to equivalent stores under public *pc*'s result in equivalent stores
Event Handler Storage Requirements		
EH1	L-lookups are $\approx_L$	Lookups in equivalent DOMs under public *pc*'s result in *strong* equivalent values
WEH1	L-lookups are $\approx_L$	Lookups in equivalent DOMs under public *pc*'s result in equivalent values
(W)EH2	H EH lookups empty	Event handler lookups under a secret *pc* produce no public event handlers
EH3	H-updates are $\approx_L$	Updates under a secret *pc* results in an equivalent store
(W)EH4	L-updates are $\approx_L$	Updates under public *pc*'s in equivalent stores result in equivalent stores

Table 5.2: Requirements for Progress-Insensitive Security and Weak Secrecy. The requirements for both are similar, except that Weak Secrecy does not use requirements T3 or EH3 and the Progress-Insensitive Security requirements E1, V1, and EH1 use strong equivalence while the Weak Secrecy requirements WE1, WV1, and WEH1 use standard equivalence.

x. If the attacker successfully output x in the past, but cannot output x now, they can conclude that a secret event handler which writes to x must have run recently. For example, the leaky third-party script shown in Listing 5.3 violates Definition 24 when the script is enforced with TT and the global storage with TS. Consider the scenario where the user inputs a password “abcd". Before the output $(\texttt{true}, L)$, the attacker knows the input was *some* password, but they are not sure which one, so their knowledge set is all possible passwords. After the output, the attacker learns that the input password must start with an ‘a’, thus refining the set of possible inputs to only the passwords beginning with ‘a’, which violates the



(a) Progress-insensitive Security: Each ●in the H context before Ev_L is L-equivalent, even though T_2 sees different H events than T_1. From $T_1 \approx_L T_2$, T_1 and T_2 see the same public input: Ev_L. We show that each step in the L context (K_1 to K_1' and K_2 to K_2') produces $\approx_L$ states and from this, we prove that T_2 can take step ● $\overset{\alpha}{\Longrightarrow}$ ◌ producing the same output $\alpha = ch(v)$ and equivalent states ○$\approx_L$ ◌.



(b) Weak Security: The proof is similar to above except that T_1 and T_2 are also synchronized on $\mathsf{gw}(_)$ and $\mathsf{br}(_)$ actions. Because of this, when T_1 takes a step to accept a low event ● $\overset{\mathsf{Ev}_L}{\Longrightarrow}$ ○, we need to know that running the event handler for $\mathsf{Ev}_{H,1}$ in T_2 (○$\Longrightarrow^*$ ○) will not produce any $\mathsf{gw}(_)$ actions. This is guaranteed by the $\mathsf{wkTrace}$ condition in $\mathcal{K}_{wp}()$.

Figure 5.11: Comparison of Progress-insensitive security (top) and Weak Security (bottom) proofs. Given $T_1 \approx_L T_2$, where T_1 takes a step to ○, we want to show that T_2 can take equivalent steps ◌, and that trace equivalence maintains state equivalence ●.

security condition. Branching on a secret implicitly leaked information to the attacker.

Instead, we prove a weaker security condition called *weak secrecy* [95, 84] which allows *implicit* leaks through control flow but still ensures that *explicit* leaks via outputs are still prevented.

Additional attacker observations We modify our semantics with additional outputs to capture both types of implicit leaks described above: $\mathsf{br}(_)$ ("branch") when branching on a tainted value in the L context, and $\mathsf{gw}(_)$ ("global write") when a L-labeled value is upgraded in the H context.

Knowledge-based weak secrecy definition Since we allow information to leak through control flow decisions, we define another form of knowledge to capture this:

Definition 26 (Weak Knowledge)**.** *An attacker's knowledge after observing T beginning from state σ_0 with security policy $\mathcal{P}$, declassification module $\mathcal{R}$, which just implicitly leaked α_I, denoted $\mathcal{K}_{wp}(T, \Sigma_0, \mathcal{P}, \mathcal{R}, \alpha_I)$ is defined as* $\{\tau_i \mid \exists T' \in \mathsf{runs}(\Sigma_0, \mathcal{P}, \mathcal{R}), T \approx_L T', \tau_i = \mathsf{in}(T'), \mathsf{prog}(T'), \mathsf{wkTrace}(T', \alpha') \text{ where } \alpha' = (\mathsf{last}(T) \overset{\alpha_I}{\Longrightarrow} K)) \downarrow_L\}$

$\mathsf{last}(T)$ returns the last configuration in a trace. Here, $\approx_L$ ensures the implicit leaks *up to this point* were the same and $\mathsf{wkTrace}$ ensures *the next* implicit leak is the same. If T is about to output $\mathsf{br}(b)$ or $\mathsf{gw}(x)$, then T' can be extended to produce the same output. We also need to make sure that when T receives a public

input, T' does not leak anything until the next public input. Because inputs come nondeterministically, and we only want to consider traces which produce the same implicit leaks, we do not want T' to leak anything extra in a secret event handler before the next public input. This ensures that if T and T' were $\approx_L$ *up to this point*, they will continue to be equivalent *after the next step*. Maintaining equivalence like this is important for proving security.

Consider, again, our leaky third-party script in Listing 5.3 where the user inputs the password "abcd". In our weak secrecy semantics, the event handler would generate $\mathsf{br}(\mathsf{true})$ when branching on the secret. The wkTrace predicate in the weak secrecy definition allows the attacker to refine their knowledge to include the fact that the branch condition must evaluate to true by throwing out all the traces which do not generate this branch condition. Only passwords starting with 'a' cause the branch condition to be true, so at this step, the attacker is allowed to learn that the password must begin with 'a' (i.e., the knowledge set is refined from all possible passwords to all possible passwords starting with 'a'). Therefore, the output does not further refine the attacker's knowledge, so this program satisfies weak secrecy.

Definition 27 (Progress-insensitive Weak Secrecy)**.** *The compositional framework satisfies progress-insensitive weak secrecy in our framework iff given any initial global store, σ_0^G, and release policy $\mathcal{P}, \mathcal{R}$, it is the case that for all traces T, actions α, and configurations K s.t. $(T \stackrel{\alpha}{\Longrightarrow} K) \in \mathsf{runs}(\sigma_0^G, \mathcal{P}, \mathcal{R})$, the following holds*

- *If* $\mathsf{rlsA}(G, \mathcal{P} \vdash \mathsf{last}(T) \stackrel{\alpha}{\Longrightarrow} K)$*:* $\mathcal{K}(G, \mathcal{P} \vdash T \stackrel{\alpha}{\Longrightarrow} K, \sigma_0^G, \mathcal{P}, \mathcal{R}) \supseteq_{\preceq} \mathcal{K}_{rp}(T, \sigma_0^G, \mathcal{P}, \mathcal{R}, \alpha)$
- *If* $\mathsf{wkAction}(\mathsf{last}(T) \stackrel{\alpha}{\Longrightarrow} K)$*:* $\mathcal{K}(T \stackrel{\alpha}{\Longrightarrow} K, \sigma_0^G, \mathcal{P}, \mathcal{R}) \supseteq_{\preceq} \mathcal{K}_{wp}(T, \sigma_0^G, \mathcal{P}, \mathcal{R}, \alpha)$
- *Otherwise:* $\mathcal{K}(T \stackrel{\alpha}{\Longrightarrow} K, \sigma_0^G, \mathcal{P}, \mathcal{R}) \supseteq_{\preceq} \mathcal{K}_p(T, \sigma_0^G, \mathcal{P}, \mathcal{R})$.

Meta-theory We prove that any combination of enforcement mechanisms that we instantiated our framework with, including TT and TS, satisfy Definition 27:

Theorem 28 (Soundness-Weak Secrecy)**.** *If event handlers are enforced with $\mathcal{V} \in \{\mathsf{SME}, \mathsf{MF}, \mathsf{TT}\}$ and the global storage is enforced with $G \in \{\mathsf{SMS}, \mathsf{FS}, \mathsf{TS}\}$, then the composition of these event handlers and global stores in our framework satisfies progress-insensitive weak secrecy.*

We prove weak secrecy using a similar technique to progress-insensitive security. The requirements are nearly the same and are shown in Table 5.2 with a **(W)**. Requirements **T3** and **EH3** cannot be proven in the presence of implicit leaks (upgrades to global variables in the H context is publicly observable). However, they are not needed to prove weak secrecy. The requirements mentioning "strong equivalence" are weakened to "equivalence" since leaking branch conditions is permitted.

Further comparisons of the proof techniques behind these two security definitions are shown in Figure 5.11. The events that two equivalent traces T_1 and T_2 must agree on for the weak secrecy definition are a superset of those required by the regular security definition, so the set of traces in the equivalent class (knowledge set) of the former is a subset of the latter. Consequently, attackers know more in the system that allows implicit leaks. We prove that our weak secrecy security condition is weaker than our standard security condition, in general:

Theorem 29 (PI Security implies PI Weak Secrecy)**.** *If the composition of event handlers and global storage enforcement are progress-insensitive secure, then they also satisfy progress-insensitive weak secrecy.*

5.4.4 Securing TT

We can prove that in the presence of a secure global storage, using taint tracking for the event handler *is* secure, even without NSU semantics.

Theorem 30 (Soundness (TT))**.** *If event handlers are enforced with* $\mathcal{V} \in \{\mathsf{TT}, \mathsf{SME}, \mathsf{MF}\}$ *and the global storage is enforced with* $G \in \{\mathsf{SMS}, \mathsf{FS}\}$*, then the composition of these event handlers and global stores in our framework satisfies progress-insensitive security.*

The proof deviates from the requirements shown in Table 5.2. We cannot prove the variable requirements for TT because looking up a tainted value violates requirement **V1**. However, these requirements are stronger than necessary. The proof is intuitive: from the requirements, a secure global store will not allow a public event handler to access secrets, nor will it let secret event handlers modify public values. Recall that the local variable storage is cleared between event handlers, so there is no way for public event handlers to branch on secret values because the local storage will only contain public values. This means that **WV1** is sufficient to prove the stronger security condition and taint tracking techniques can be used securely, without NSU semantics, as long as the global structures satisfy strong security guarantees.

Going back to our example, the event handler in Listing 5.1 is secure, even though it is enforced with TT because it does not have implicit leaks. On the other hand, code with implicit leaks (Listing 5.2 and 5.3) can be secured by connecting the taint tracking script enforcement with a secure storage like SMS or FS, as shown by Theorem 30. This is noteworthy because it suggests that the selection of script enforcement is not as relevant to security as the selection of the global storage enforcement. Furthermore, the effects of TT are not manifested in this setting (since tainted variables never appear in the L context), meaning that as long as the shared structures are secure, the event handlers may require *no additional enforcement*.

5.5 Discussion

So far, we developed a compositional framework to combine multi-execution techniques (strong security guarantees) with taint tracking (weaker security guarantees). One question that remains is whether we can use a compositional definition and proof infrastructure of the form, "If A is secure and B is secure, then their composition is secure". This is challenging in our setting because the security of event handlers often depends on the security of the global store. Instead, we define compositional security based on the interfaces between event handlers and global storage in a rely-guarantee style using "requirements" on execution traces, variable storage, and event handler storage.

5.6 Summary

We develop a framework to enable the flexible composition of dynamic IFC enforcement mechanisms for reactive programs with provable security guarantees. We use a knowledge-based security condition to compare the relative security of different compositions. We extend weak secrecy to reason about implicit flows of information due to control flow decisions within as well as between event handlers. Finally, we show that some compositions (namely, ones using multi-storage techniques for shared storage) allows taint tracking enforcement for event handlers to achieve *stronger* security guarantees compared to using taint tracking for all of the code.

Chapter 6

Discussion and Future Work

In this Chapter we discuss alternative approaches for protecting sensitive data on the web (Section 6.1) as well as potential directions for future work, including alternative web models or reactive systems (Sections 6.2 and 6.3) and possible extensions for our compositional framework (Section 6.4).

6.1 Alternatives to IFC

In this thesis, we focus on IFC techniques but there are other approaches for protecting sensitive data on the web. One option is to isolate untrusted JavaScript [90] so that it cannot access sensitive user data on the webpage. These approaches would guarantee that untrusted, possibly attacker-controlled, code cannot access private information, but they also tend to be more coarse-grained than IFC techniques, and may require code or browser modifications (similar to our IFC monitors). The tradeoffs between isolation and IFC techniques have been studied [86], highlighting that that performance and usability are both potential barriers to IFC adoption. Our compositional framework is one approach to help alleviate the performance concerns, but more research is needed on usable IFC. Future efforts in this area should look at how users or developers can write complex policies [100] and how to allow users to express their preferences about the tradeoffs between IFC techniques (as we describe in Chapter 5.1).

6.2 More Realistic Web Models

Alternative DOM models, event bubbling, pre-emptive event scheduling DOM event handling logic is quite complex and can be used to leak information [76, 25, 5, 73]. Interactions between SME and DOM event scheduling logic is an interesting problem that has not been investigated. Some of those problems can be mitigated in our system because script-generated events are handled by the execution at the same

security level. However, as we show in Chapter 4, sometimes surprising interactions *between* executions arise due to declassification. Interactions between event bubbling/pre-emptive event scheduling and declassification may pose new challenges. It would also be interesting to incorporate a more realistic tree-structured DOM into Chapter 4 to explore whether the structure of the DOM yields any interesting new ways for the attacker to influence declassification, and whether the same technique can mitigate the new risks.

DOM element removal In Chapters 3 and 4, we show that attackers can leverage dynamic features like DOM element generation and event simulation to trigger declassifications. DOM element removal is another dynamic feature that could lead to leaks. For example, consider the following program where the security policy says that keypress events should be kept secret and button clicks may be declassified:

$$\mathsf{onKeyPress}(k)\ \{\ \mathsf{if}\ k == 42\ \mathsf{then}\ \mathsf{remove}(b)\ \}$$

This event handler removes the button identified by b if the secret keypress is 42, otherwise (for other keypresses) the button stays. If the attacker sees a declassified event associated with b they will know that the user has not generated a secret keypress with 42: declassifying an event from a button that *might* have been removed if a different branch were taken leaks something about secret keypress events to the attacker. Unfortunately, leaks are difficult to detect using purely dynamic monitors because dynamic techniques only know what happens in the branch that gets executed, while this leak is related to the branch of code that *was not* executed. A hybrid monitor that looks at both branches and does not allow declassification for events associated with page elements which might have been removed would be a possible solution.

Other web features and implementation Our work focuses on JavaScript event handlers and interactions with the DOM, but we could expand our model to include other relevant browser features. For example, the network and AJAX requests included in the Featherweight Firefox model [27] would require more sophisticated event handler scheduling, but could also lead to a more interesting attacker model for Chapter 4. Another direction for future research would be to develop a prototype browser based on Firefox (similar to FlowFox [35]) or Chromium (similar to prior work [22]). One goal of our compositional framework is that it would allow users to balance the tradeoffs of different enforcement mechanisms. We also assume that someone (e.g., the user, or browser manufacturer) writes security policies to determine which information flows are allowed. A prototype browser would also help us evaluate the usability of the mechanisms for selecting IFC enforcement and writing policies.

6.3 Applications to Other Reactive Settings

Our framework can be applied to different reactive settings, such as web apps with a full DOM, OS processes [54, 102], mobile phone applications [52, 41, 68], and serverless computing [6]. Other reactive systems typically have less sophisticated storage than the DOM and more complex scheduling compared to JavaScript's single-threaded execution. We would need to modify the semantics to accommodate different schedulers and ensure they do not become a source of information leakage.

In Chapter 5, we find that the security of the overall application may rely more on the security of the shared resources than the individual event handlers. Meanwhile, when enforcing information flow policies in concurrent systems [85, 89, 8, 55], the security of shared resources is also important to the security of the overall application. It would be interesting to explore other connections between our compositional framework and work on information flow in concurrent systems, which may also involve timing-sensitive [77, 87] or probabilistic [93, 80] notions of noninterference.

6.4 Extending our Compositional Framework

Additional IFC monitors We consider only a few dynamic enforcement mechanisms, but our framework could be easily extended to accommodate others. To add another event handler enforcement mechanism, the local storage would need to be defined as well as conditions for when to produce or suppress output. Rules for interacting with the local storage and any other special rules (for instance, for switching executions in SME or branching on faceted values in MF) would also need to be added. For global variable storage, only the storage syntax and rules for accessing the store would be necessary. The event handler storage is by far the most involved, likely requiring rules for defining the storage itself as well as rules for interacting with each of the event handler enforcement mechanisms.

Traditional taint tracking upgrades the *pc* when branching on a tainted value, but our semantics in Chapter 5 do not. We made this choice for two reasons: First, this choice is consistent with prior work on weak secrecy [84, 95]. Second, upgrading the *pc* on tainted branches adds complexity to the semantics, but still leaks information. Other approaches like *no sensitive upgrade* (NSU) semantics (and its more flexible variants like permissive upgrades [14]), or terminating the execution of individual event handlers [73], can be adapted to the reactive setting. Adapting these mechanisms for our framework is straightforward: low-level rules for commands need to be defined. Variants of NSU techniques may achieve a stronger security guarantee but run the risk of altering the behavior of non-leaky programs if they prevent upgrades to variables which never affect outputs to public channels. The focus of our work is on the effect of composition on security and we leave the investigation of additional mechanisms to future work.

Attacker influence and other security properties Our compositional framework in Chapter 5 uses the techniques from Chapter 3 for protecting declassification from attacker influence. It would be interesting to incorporate our notion of integrity from Chapter 4 into our framework, both to allow for more fine-grained monitoring, but also to investigate whether the attacks we identify for SME (specifically the leaks *between* executions from Chapter 4.2.3) also exist in faceted execution.

Appendix A

Supporting Materials for Chapter 3

A.1 Additional Definitions

A.1.1 Equivalence definitions

We say two configurations Σ_1 and Σ_2 are equivalent w.r.t. label L, denoted $\Sigma_1 \approx_L \Sigma_2$, if their declassification states and configurations for the low-execution are the same.

Definition 31 (Configuration equivalence)**.** *Given two configurations Σ_1 and Σ_2, where $\Sigma_1 = \mathcal{R}_1, d_1; \kappa_{L1}; \kappa_{H1}$ and $\Sigma_2 = \mathcal{R}_2, d_2; \kappa_{L2}; \kappa_{H2}$, $\Sigma_1 \approx_L \Sigma_2$ iff $\mathcal{R}_1 = \mathcal{R}_2, d_1 = d_2$, and $\kappa_{L1} = \kappa_{L2}$.*

$$\frac{}{\mathsf{proj}_{\mathcal{R},\mathcal{P}}(\cdot) = \cdot} \qquad \frac{\mathcal{P} \vdash a : L}{\mathsf{proj}_{\mathcal{R},\mathcal{P}}(a) = a} \qquad \frac{\mathcal{P} \vdash a : H_\Delta}{\mathsf{proj}_{\mathcal{R},\mathcal{P}}(a) = \cdot}$$

$$\frac{\begin{array}{c}\mathcal{P} \vdash a : H \qquad \mathcal{R} = (\rho, \mathcal{D}) \qquad \mathcal{D}(\rho, a) = (\rho', r, a') \\ \rho' \neq \rho \text{ or } r \neq \mathsf{none} \text{ or } a' \neq \bullet\end{array}}{\mathsf{proj}_{\mathcal{R},\mathcal{P}}(a) = (\rho', r, a')} \qquad \frac{\mathcal{P} \vdash a : H \qquad \mathcal{R} = (\rho, \mathcal{D}) \qquad \mathcal{D}(\rho, a) = (\rho, \mathsf{none}, \bullet)}{\mathsf{proj}_{\mathcal{R},\mathcal{P}}(a) = \cdot}$$

Figure A.1: Projection of actions

$$\frac{}{(\Sigma) \Downarrow_L^{\mathcal{P}} = \cdot} \qquad \frac{\Sigma = \mathcal{R}, d; \kappa_L; \kappa_H \qquad T' \in \mathsf{runs}(\Sigma', \mathcal{R}', \mathcal{P}) \qquad \Sigma \not\approx_L \Sigma' \qquad \alpha \in \mathsf{in}(T)}{(\mathcal{P} \vdash \Sigma \stackrel{\alpha}{\Longrightarrow} T') \Downarrow_L^{\mathcal{P}} = \mathsf{proj}_{\mathcal{R},\mathcal{P}}(\alpha) :: T' \Downarrow_L^{\mathcal{P}}}$$

$$\frac{T' \in \mathsf{runs}(\Sigma', \mathcal{R}', \mathcal{P}) \qquad \Sigma \not\approx_L \Sigma' \qquad \alpha \notin \mathsf{in}(T)}{(\mathcal{P} \vdash \Sigma \stackrel{\alpha}{\Longrightarrow} T') \Downarrow_L^{\mathcal{P}} = \alpha :: T' \Downarrow_L^{\mathcal{P}}} \qquad \frac{T' \in \mathsf{runs}(\Sigma', \mathcal{R}', \mathcal{P}) \qquad \Sigma \approx_L \Sigma'}{(\mathcal{P} \vdash \Sigma \stackrel{\alpha}{\Longrightarrow} T') \Downarrow_L^{\mathcal{P}} = T' \Downarrow_L^{\mathcal{P}}}$$

Figure A.2: Projection of traces

Next we formally define the equivalence between two traces w.r.t. L, denoted $T_1 \approx_L^{\mathcal{P}} T_2$. We first define a projection relation on actions based on the declassification module, denoted $\mathsf{proj}_{\mathcal{R},\mathcal{P}}(a)$ in Figure A.1. We define the projection of execution traces w.r.t. $\mathcal{P}$, denoted $T \Downarrow_L^{\mathcal{P}}$, in Figure A.2. We say two traces are equivalent w.r.t. the label L and label context $\mathcal{P}$ if their projections are the same.

Definition 32 (Trace equivalence). *Given two execution traces T_1 and T_2, $T_1 \approx_L^{\mathcal{P}} T_2$ iff $T_1 \Downarrow_L^{\mathcal{P}} = T_2 \Downarrow_L^{\mathcal{P}}$.*

A.1.2 Knowledge definitions

We use the same knowledge definitions as in Chapter 3.4.2.

Definition 2 (Attacker Knowledge). *Formally, $\mathcal{K}(T, \sigma_0, \mathcal{P}, \mathcal{R})$ is defined as $\{\tau_i \mid \exists T' \in \mathsf{iruns}(\sigma_0, \mathcal{P}, \mathcal{R}), T \approx_L^{\mathcal{P}} T', \tau_i = \mathsf{in}(T')\}$*

Definition 4 (Progress Knowledge). *Formally, $\mathcal{K}_p(T, \sigma_0, \mathcal{P}, \mathcal{R})$ is defined as $\{\tau_i \mid \exists T' \in \mathsf{iruns}(\sigma_0, \mathcal{P}, \mathcal{R}), T \approx_L^{\mathcal{P}} T', \tau_i = \mathsf{in}(T'), \mathsf{prog}(T', \mathcal{P})\}$*

Where $\mathsf{prog}(T, \mathcal{P})$ holds iff $T = \mathcal{P} \vdash \Sigma_0 \Longrightarrow^* \Sigma$ and $\exists T'$ s.t. $T' = \Sigma \overset{\tau}{\Longrightarrow}{}^* \Sigma_C$, and $\mathsf{consumer}(\Sigma_C)$

A.1.3 Robust declassification

Robustness requires that active attackers cannot learn more than passive attackers. We define $\sigma_1 <_A \sigma_2$ to mean that σ_2 contains more active components (i.e., event handlers and page elements) than σ_1.

$\boxed{\sigma_1 <_A \sigma_2}$

$$\frac{}{\sigma <_A \sigma}\ \textsc{refl} \qquad \frac{\sigma_1 <_A \sigma_2 \quad x \notin \mathsf{dom}(\sigma_2)}{\sigma_1 <_A \sigma_2[x \mapsto v]}\ \textsc{var} \qquad \frac{\sigma_1 <_A \sigma_2}{\sigma_1 <_A \sigma_2, id \mapsto (v, M)}\ \textsc{obj}$$

$$\frac{\sigma_1 <_A \sigma_2 \qquad M_1 <_A M_2}{\sigma_1, id \mapsto (v, M_1) <_A \sigma_2, id \mapsto (v, M_2)}\ \textsc{eh}$$

We say two input traces τ_1 and τ_2 are L equivalent w.r.t. the label context $\mathcal{P}$ and release policy $\mathcal{R}$, written $\tau_1 \approx_L^{\mathcal{P},\mathcal{R}} \tau_2$, when they have the same low inputs and equivalent high inputs under declassification. We define $\mathcal{R}^*_{\mathcal{P}}(\tau)$ in Figure A.3, which is the declassified input trace of τ.

Definition 33 (Input equivalence w.r.t. release policy). *We say that τ_1 and τ_2 are equivalent w.r.t. the label context $\mathcal{P}$ and release module $\mathcal{R}$, written $\tau_1 \approx_L^{\mathcal{P},\mathcal{R}} \tau_2$, iff $\mathcal{R}^*_{\mathcal{P}}(\tau_1) = \mathcal{R}^*_{\mathcal{P}}(\tau_2)$.*

We write $(\sigma, \mathcal{P}, \mathcal{R}, \tau_i) \Uparrow$ to mean that the SME execution trace starting from σ given input τ_i will not reach a consumer state. We say $\kappa(\tau) \Uparrow$ iff there does not exist a κ' s.t. $\kappa \overset{\tau}{\longrightarrow}{}^* \kappa'$ and $\mathsf{consumer}(\kappa')$. We say $\Sigma(\tau) \Uparrow$ iff there does not exist a Σ' s.t. $\mathcal{P} \vdash \Sigma \overset{\tau}{\Longrightarrow}{}^* \Sigma'$ and $\mathsf{consumer}(\Sigma')$.

$$\frac{}{\mathcal{R}^*_{\mathcal{P}}(\cdot) = \cdot} \qquad \frac{\mathcal{P} \vdash a : L}{\mathcal{R}^*_{\mathcal{P}}(a\ \tau) = a :: \mathcal{R}^*_{\mathcal{P}}(\tau)} \qquad \frac{\mathcal{P} \vdash a : H_\Delta}{\mathcal{R}^*_{\mathcal{P}}(a\ \tau) = \mathcal{R}^*_{\mathcal{P}}(\tau)}$$

$$\frac{\mathcal{P} \vdash a : H \quad \mathcal{R} = (\rho, \mathcal{D}) \quad \mathcal{D}(\rho, a) = (\rho', r, a') \quad \rho' \neq \rho \text{ or } r \neq \mathsf{none} \text{ or } a' \neq \bullet \quad \mathcal{R}' = (\rho', \mathcal{D})}{\mathcal{R}^*_{\mathcal{P}}(a\ \tau) = (r, a') :: \mathcal{R}'^*_{\mathcal{P}}(\tau)}$$

$$\frac{\mathcal{P} \vdash a : H \quad \mathcal{R} = (\rho, \mathcal{D}) \quad \mathcal{D}(\rho, a) = (\rho, \mathsf{none}, \bullet)}{\mathcal{R}^*_{\mathcal{P}}(a\ \tau) = \mathcal{R}^*_{\mathcal{P}}(\tau)}$$

Figure A.3: Declassified input trace

We formally define an interleaving of two traces as follows:

$$\frac{}{\tau_1 \bowtie \cdot = \tau_1} \qquad \frac{\tau_1 = \tau_1' :: \tau_1'' \quad \tau_2 = \alpha :: \tau_2'}{\tau_1 \bowtie \tau_2 = \tau_1' :: \alpha :: (\tau_1'' \bowtie \tau_2')}$$

We define the following relation for A and B, sets of traces:

$$A \subseteq_{\ll} B \text{ iff } \forall \tau \in A, \exists \tau', \tau_\Delta \text{ with } \tau' \in B, \text{ and } \tau \bowtie \tau_\Delta = \tau'$$

We define the domain of a set of inputs:

$$\frac{\tau = \tau' :: \alpha}{\mathsf{dom}(\tau) = \mathsf{dom}(\alpha) \cup \mathsf{dom}(\tau')} \qquad \frac{\alpha = id.ev(v)}{\mathsf{dom}(\alpha) = \{id\}} \qquad \frac{\alpha = ch(v)}{\mathsf{dom}(\alpha) = \{\ \}}$$

Definition 34 (Compatibility)**.** *We say that a state σ is compatible with a release policy R and label context $\mathcal{P}$, when for all τ $\mathcal{P} \vdash d_0, R; \kappa \xrightarrow{\tau}{}^* d', R'; \kappa'$ iff $\kappa \xrightarrow{\tau}{}^* \kappa'$ where d_0 is the initial release channel, $\kappa = (\sigma, \mathsf{skip}, C, \cdot)$.*

We say that a configuration is in consumer state, written $\mathsf{consumer}(\Sigma)$ or $\mathsf{consumer}(\kappa)$ if all its execution states are in consumer state.

$$\begin{array}{lll} \mathsf{producer}(\kappa) & \text{iff} & \exists \sigma, c, E, \kappa = (\sigma, c, P, E) \\ \mathsf{consumer}(\kappa) & \text{iff} & \exists \sigma, \kappa = (\sigma, \mathsf{skip}, C, \cdot) \\ \mathsf{consumer}(\Sigma) & \text{iff} & \exists d, \mathcal{R}, \Sigma = d, \mathcal{R}; \kappa_L; \kappa_H \text{ and } \mathsf{consumer}(\kappa_L) \text{ and } \mathsf{consumer}(\kappa_H) \end{array}$$

We say that for a trace $T = \Sigma_1 \Longrightarrow ... \Longrightarrow \Sigma_N$, an increasing sequence of indices $[i_1, i_2, ..., i_K]$ s.t. $K \leq N$ is a separation.

For a separation $[i_1, i_2, ..., i_K]$:

(1) $\Sigma_1 \Longrightarrow^* \Sigma_{i_1}$ is a sequence;

(2) For any $t \leq K-1$, $\Sigma_{i_t+1} \Longrightarrow^* \Sigma_{i_{t+1}}$ is a sequence.

We define $\mathsf{fst}(T)$ and $\mathsf{last}(T)$ to be the first and last configuration of T respectively.

We say that a trace T is a complete run if $\mathsf{consumer}(\mathsf{last}(T))$.

We call a trace T a complete segment if $\mathsf{consumer}(\mathsf{last}(T))$, and no configuration in T other than the first the last is in consumer state.

$\Sigma \sim_H^C d, \mathcal{R}; \kappa_H$ iff $\Sigma = d, \mathcal{R}; \kappa_L; \kappa_H$ and $\mathsf{consumer}(\Sigma)$

$\Sigma \sim_L^C d, \kappa_L$ iff $\Sigma = d, \mathcal{R}; \kappa_L; \kappa_H$ and $\mathsf{consumer}(\kappa_L)$

$\Sigma \sim_H^P d, \mathcal{R}; \kappa_H$ iff $\Sigma = d, \mathcal{R}; \kappa_L; \kappa_H$ and $\mathsf{consumer}(\kappa_L)$, and $\neg\mathsf{consumer}(\kappa_H)$

$\Sigma \sim_H^{PP} d, \mathcal{R}; \kappa_H$ iff $\Sigma = d, \mathcal{R}; \kappa_L; \kappa_H$ and $\neg\mathsf{consumer}(\kappa_L)$

$\Sigma \sim_L^P d, \kappa_L$ iff $\Sigma = d, \mathcal{R}; \kappa_L; \kappa_H$ and $\neg\mathsf{consumer}(\kappa_H)$

$\Sigma \sim_L^{PP} d, \mathcal{R}; \kappa_L$ iff $\Sigma = d, \mathcal{R}; \kappa_L; \kappa_H$, $\mathsf{consumer}(\kappa_L)$ and $\neg\mathsf{consumer}(\kappa_H)$

$$\frac{\Sigma \sim_H^C d, \mathcal{R}; \kappa_H}{\mathcal{P} \vdash \Sigma \sim_H^T d, \mathcal{R}; \kappa_H}\ \textsc{Sim-HS} \qquad \frac{\Sigma \sim_H^P d, \mathcal{R}; \kappa \quad \mathcal{P} \vdash \alpha_2 : H \quad \alpha_1 = \alpha_2 \quad \mathcal{P} \vdash T \sim_H^T t}{\mathcal{P} \vdash \Sigma \stackrel{\alpha_1}{\Longrightarrow} T \sim_H^T d, \mathcal{R}; \kappa \stackrel{\alpha_2}{\longrightarrow} t}\ \textsc{Sim-HH}$$

$$\frac{\Sigma \sim_H^P d, \mathcal{R}; \kappa \quad \mathcal{P} \vdash \alpha_2 : L \text{ or } \alpha_2 = \bullet \quad \mathcal{P} \nvdash \alpha_1 : H \quad \mathcal{P} \vdash T \sim_H^T t}{\mathcal{P} \vdash \Sigma \stackrel{\alpha_1}{\Longrightarrow} T \sim_H^T d, \mathcal{R}; \kappa \stackrel{\alpha_2}{\longrightarrow} t}\ \textsc{Sim-HL}$$

$$\frac{\Sigma \sim_H^{PP} d, \mathcal{R}; \kappa_H}{\mathcal{P} \vdash \Sigma \sim_H^{TP} d, \mathcal{R}; \kappa_H}\ \textsc{Sim-LS} \qquad \frac{\Sigma \sim_H^{PP} d, \mathcal{R}; \kappa_H \quad \mathcal{P} \vdash \alpha : L \text{ or } \alpha = \bullet \quad \mathcal{P} \vdash T \sim_H^{TP} d, \mathcal{R}; \kappa_H}{\mathcal{P} \vdash \Sigma \stackrel{\alpha}{\Longrightarrow} T \sim_H^{TP} d, \mathcal{R}; \kappa_H}\ \textsc{Sim-LL}$$

Definition 35 (No leak outside declassification). *We say that a state σ is has no leak outside declassification, if for all label context $\mathcal{P}$, $\mathcal{R}$, $\mathcal{R}'$, τ_{i1} and τ_{i2}, s.t. $\mathcal{R}^*_{\mathcal{P}}(\tau_{i_1}) = \mathcal{R}'^*_{\mathcal{P}}(\tau_{i_2})$, for all τ_1 and τ_2 $\mathcal{P} \vdash t_1 = d_0, R; \kappa \longrightarrow^* d', R'; \kappa'$ and $\mathcal{P} \vdash t_2 = d_0, R'; \kappa \longrightarrow^* d', R'; \kappa'$, and $\mathsf{in}(t_1) = \tau_{i1}$, $\mathsf{in}(t_2) = \tau_{i2}$, it is the case that $\mathsf{out}(t_1)|_L^{\mathcal{P}} = \mathsf{out}(t_2)|_L^{\mathcal{P}}$.*

We define $pr(\mathcal{P}, \mathcal{R}, n, \tau) = (\tau', A)$, to help compute a new projection function. It returns a projected trace and an array that records released values.

$$\frac{}{pr(\mathcal{P},\mathcal{R},n,\cdot) = (\cdot,\emptyset)} \qquad \frac{\mathcal{P} \vdash a : L \qquad pr(\mathcal{P},\mathcal{R},n,\tau) = (\tau',A)}{pr(\mathcal{P},\mathcal{R},n,a\ \tau) = (a :: \tau',A)} \qquad \frac{\mathcal{P} \vdash a : H_\Delta}{pr(\mathcal{P},\mathcal{R},n,a\ \tau) = pr(\mathcal{P},\mathcal{R},n,\tau)}$$

$$\frac{\begin{array}{c}\mathcal{P} \vdash a : H \qquad \mathcal{R} = (\rho,\mathcal{D}) \\ \mathcal{D}(\rho,a) = (\rho',r,a') \qquad \rho' \neq \rho \text{ or } r \neq \textsf{none} \text{ or } a' \neq \bullet \qquad \mathcal{R}' = (\rho',\mathcal{D}) \qquad pr(\mathcal{P},\mathcal{R},n+1,a\ \tau) = (\tau',A)\end{array}}{pr(\mathcal{P},\mathcal{R},n,a\ \tau) = (a' :: \tau',A[n] := r)}$$

$$\frac{\mathcal{P} \vdash a : H \qquad \mathcal{R} = (\rho,\mathcal{D}) \qquad \mathcal{D}(\rho,a) = (\rho,\textsf{none},\bullet)}{pr(\mathcal{P},\mathcal{R},n,a\ \tau) = pr(\mathcal{P},\mathcal{R},n,\tau)}$$

$PR(\mathcal{P},\mathcal{R},\tau) = (\tau',\mathcal{R}')$ where $pr(\mathcal{P},\mathcal{R},1,\tau) = (\tau',A)$, $\mathcal{R}' = ((0,A),\mathcal{D}')$, and $\mathcal{D}'((n,A),a) = ((n+1,A),A[n],a)$.

We assume that If $\mathcal{P} \vdash a : H$, and $\mathcal{D}(\rho,a) = a'$, then $\mathcal{P} \vdash a' : H$. We make sure that the projected input $\bullet$ is considered H.

$$\frac{\Sigma \sim_L^{PP} d,\mathcal{R};\kappa_L}{\mathcal{P} \vdash \Sigma \sim_L^{TP} d,\mathcal{R};\kappa_L} \qquad \frac{\Sigma \sim_L^{PP} d,\mathcal{R};\kappa_L \qquad \mathcal{P} \vdash \alpha : H \text{ or } \alpha = \bullet \qquad \mathcal{P} \vdash T \sim_L^{TP} d,\mathcal{R};\kappa_L}{\mathcal{P} \vdash \Sigma \stackrel{\alpha}{\Longrightarrow} T \sim_L^{TP} d,\mathcal{R};\kappa_L}$$

$$\frac{\mathcal{R}^*_{\mathcal{P}}(\tau) = \mathcal{R}'^*_{\mathcal{P}}(\tau') \qquad \textsf{consumer}(\kappa_H) \qquad \textsf{consumer}(\kappa_L)}{\mathcal{P} \vdash d,\mathcal{R};\kappa_L;\kappa_H \sim_L^C d,\mathcal{R}';\kappa_L@(\tau,\tau')}$$

$$\frac{\Sigma \sim_L^C d,\kappa_L}{\mathcal{P} \vdash \Sigma \sim_L^T d,\mathcal{R};\kappa_L}\ \textsc{Sim-LC} \qquad \frac{\Sigma \sim_L^P d,\kappa \qquad \mathcal{P} \vdash \alpha_2 : L \qquad \alpha_1 = \alpha_2 \qquad \mathcal{P} \vdash T \sim_L^T t}{\mathcal{P} \vdash \Sigma \stackrel{\alpha_1}{\Longrightarrow} T \sim_L^T d,\mathcal{R};\kappa \stackrel{\alpha_2}{\longrightarrow} t}\ \textsc{Sim-LL}$$

$$\frac{\Sigma \sim_L^P d,\kappa \qquad \mathcal{P} \vdash \alpha_2 : H \text{ or } \alpha_2 = \bullet \qquad \mathcal{P} \nvdash \alpha_1 : L \qquad \mathcal{P} \vdash T \sim_L^T t}{\mathcal{P} \vdash \Sigma \stackrel{\alpha_1}{\Longrightarrow} T \sim_L^T d,\mathcal{R};\kappa \stackrel{\alpha_2}{\longrightarrow} t}\ \textsc{Sim-HL}$$

A.2 Soundness Proofs

Theorem 6 (Soundness) $\forall \mathcal{P},\mathcal{R},\sigma_0,T,\Sigma,\alpha\ s.t.\ (T \stackrel{\alpha}{\Longrightarrow} \Sigma) \in \textsf{iruns}(\sigma_0,\mathcal{P},\mathcal{R})$,
$\mathcal{K}(T \stackrel{\alpha}{\Longrightarrow} \Sigma,\mathcal{P},\mathcal{R}) \supseteq_\preceq \mathcal{K}_p(T,\mathcal{P},\mathcal{R})$

Proof.

Need to show $\forall \tau \in \mathcal{K}_p(T, \mathcal{P}, \mathcal{R})$, $\exists \tau' \in \mathcal{K}(T \overset{\alpha}{\Longrightarrow} \Sigma, \mathcal{P}, \mathcal{R})$ s.t. $\tau \preceq \tau'$

Let $\tau \in \mathcal{K}_p(\mathcal{P} \vdash T, \mathcal{P}, \mathcal{R})$.

Then from the definition of $\mathcal{K}_p()$, there is a trace for which τ is the input $T_1 = \mathcal{P} \vdash \Sigma_0 \Longrightarrow^* \Sigma_1$

and $\tau = \mathsf{in}(T_1)$

From definition of $\mathcal{K}_p()$, we know that $T_1 \approx_L^{\mathcal{P}} T$

We also know from $\mathcal{K}_p()$ that there is a trace $T = \mathcal{P} \vdash \Sigma_0 \Longrightarrow^* \Sigma_2$ and $\Sigma_2 \overset{\alpha}{\Longrightarrow} \Sigma$

Case I: $\Sigma \approx_L \Sigma_2$

By assumption and Lemma 37, $(\mathcal{P} \vdash \Sigma_0 \Longrightarrow^* \Sigma_2) \approx_L^{\mathcal{P}} (\mathcal{P} \vdash \Sigma_0 \Longrightarrow^* \Sigma_2 \Longrightarrow \Sigma)$

Then from $T_1 \approx_L^{\mathcal{P}} T$, we know that $T_1 \approx_L^{\mathcal{P}} (\mathcal{P} \vdash T \overset{\alpha}{\Longrightarrow} \Sigma)$

Let $\tau' = \mathsf{in}(T_1)$.

Thus, from $T_1 \approx_L^{\mathcal{P}} (\mathcal{P} \vdash T \overset{\alpha}{\Longrightarrow} \Sigma)$, we know that $\tau' \in \mathcal{K}(T \overset{\alpha}{\Longrightarrow} \Sigma, \mathcal{P}, \mathcal{R})$ and $\tau \preceq \tau'$.

Case II: $\Sigma \not\approx_L \Sigma_2$

From $T_1 \approx_L^{\mathcal{P}} T$ and Lemma 123, $\Sigma_1 \approx_L \Sigma_2$

Then from Lemma 7, $\exists \Sigma_1', \tau''$ s.t.

$\mathcal{P} \vdash \Sigma_1 \overset{\tau''}{\Longrightarrow^*} \Sigma_1'$ with $(\Sigma_1 \overset{\tau''}{\Longrightarrow^*} \Sigma_1') \approx_L^{\mathcal{P}} (\Sigma_2 \overset{\alpha}{\Longrightarrow} \Sigma)$.

Then from $T_1 \approx_L^{\mathcal{P}} T$, we know that $(T_1 \overset{\tau''}{\Longrightarrow^*} \Sigma_1') \approx_L^{\mathcal{P}} (T \overset{\alpha}{\Longrightarrow} \Sigma)$, so

Thus, from $(T_1 \overset{\tau''}{\Longrightarrow^*} \Sigma_1') \approx_L^{\mathcal{P}} (T \overset{\alpha}{\Longrightarrow} \Sigma)$ we know that $\mathsf{in}(T_1) :: \mathsf{in}(\Sigma_1 \overset{\tau''}{\Longrightarrow^*} \Sigma_1') \in \mathcal{K}(T \overset{\alpha}{\Longrightarrow} \Sigma, \mathcal{P}, \mathcal{R})$

Let $\tau' = \mathsf{in}(T_1) :: \mathsf{in}(\Sigma_1 \overset{\tau''}{\Longrightarrow^*} \Sigma_1')$.

Thus, we know that $\tau' \in \mathcal{K}(T \overset{\alpha}{\Longrightarrow} \Sigma, \mathcal{P}, \mathcal{R})$ and $\tau \preceq \tau'$

□

Lemma 8 (Eq trace Eq state) *If $T_1 = \mathcal{P} \vdash \Sigma_1 \Longrightarrow^* \Sigma_1'$ and $T_2 = \mathcal{P} \vdash \Sigma_2 \Longrightarrow^* \Sigma_2'$ with $\Sigma_1 \approx_L \Sigma_2$ and $T_1 \approx_L^{\mathcal{P}} T_2$, then $\Sigma_1' \approx_L \Sigma_2'$.*

Proof.

$\exists \tau_1, \tau_2$ s.t. $T_1 = \mathcal{P} \vdash \Sigma_1 \overset{\tau_1}{\Longrightarrow^*} \Sigma_1'$ and $T_2 = \mathcal{P} \vdash \Sigma_2 \overset{\tau_2}{\Longrightarrow^*} \Sigma_2'$

By induction over the length of τ_1.

Base Case: $\mathsf{len}(\tau_1) = 0$

Follows from $T_1 \approx_L^{\mathcal{P}} T_2$, $T_1 \Downarrow_L = \cdot$, and the definition of $\approx_L^{\mathcal{P}}$ for execution traces

Inductive Case $\mathsf{len}(\tau_1) = k + 1$ **where** $k \geq 0$

Subcase I: $\Sigma_1'' \approx_L \Sigma_1'$

Follows from Lemma 37 and the IH

Subcase II: $\Sigma_1'' \not\approx_L \Sigma_1'$

The proof for this case follows from the IH and Lemma 36

□

Lemma 7 (Strong One-step) *If* $T_1 = \mathcal{P} \vdash \Sigma_1 \overset{\alpha}{\Longrightarrow} \Sigma_1'$ *with* $\Sigma_1 \not\approx_L \Sigma_1'$, $\Sigma_1 \approx_L \Sigma_2$, *and* $\mathsf{prog}(\Sigma_2, \mathcal{P})$ *then* $\exists \Sigma_2', T_2$ *s.t.* $T_2 = \mathcal{P} \vdash \Sigma_2 \Longrightarrow^* \Sigma_2'$ *with* $T_1 \approx_L^{\mathcal{P}} T_2$ *and* $\Sigma_1' \approx_L \Sigma_2'$

Proof.

We examine each case of $\mathcal{E} :: \mathcal{P} \vdash \Sigma_1 \overset{\alpha}{\Longrightarrow} \Sigma_1'$. The proof is straightforward. We show a few representative cases.

Denote $\Sigma_1 = \mathcal{R}_1, d_1; \kappa_{L1}; \kappa_{H1}$ with $\kappa_{L1} = \sigma_{L1}, c_{L1}, s_{L1}, E_{L1}$ and likewise for $\kappa_{H1}, \Sigma_2, \Sigma_1', \Sigma_2'$

Case I: $\mathcal{E}$ ends in SMEI-NR1

From $\Sigma_1 \approx_L \Sigma_2$, we know that $s_{L2} = C$ and $c_{L2} = \mathsf{skip}$

By assumption, $\mathcal{P}(id.ev(v)) = H$

Subcase i: $s_{H2} \neq C$

From $\mathsf{prog}(\Sigma_2, \mathcal{P})$, we know that $\exists T'$ s.t. $T' = \mathcal{P} \vdash \Sigma_2 \overset{\tau}{\Longrightarrow}^* \Sigma_C$ with $\mathsf{consumer}(\Sigma_C)$

Then repeatedly applying SMEO-LH and SMEO-HH will produce T' s.t. $T' \Downarrow_L^{\mathcal{P}} = \cdot$

From this, $\Sigma_C \approx_L \Sigma_2$ and $s_{H2} = C$, so the proof proceeds the same as Subcase ii, below

Subcase ii: $s_{H2} = C$

From $\Sigma_1 \approx_L \Sigma_2$, we know that $\mathcal{R}_1 = (\rho_1, \mathcal{D}_1) = \mathcal{R}_2 = (\rho_2, \mathcal{D}_2)$, which tells us that $\mathcal{D}_1(\rho_1, id.ev(v)) = (r, \mathsf{emp}, \rho') = \mathcal{D}_2(\rho_2, id.ev(v))$

Then, SMEI-NR1 may be applied to the second trace with input $id.ev(v)$

From $\Sigma_1 \approx_L \Sigma_2$, we know that $d_1 = d_2$

Then from $\mathcal{D}_1(\rho_1, id.ev(v)) = (r, \mathsf{emp}, \rho') = \mathcal{D}_2(\rho_2, id.ev(v))$, we know that $d_1' = \mathsf{update}(d_1, r) = \mathsf{update}(d_2, r) = d_2'$

Then from $\Sigma_1 \approx_L \Sigma_2$, $\mathcal{D}_1(\rho_1, id.ev(v)) = (r, \mathsf{emp}, \rho') = \mathcal{D}_2(\rho_2, id.ev(v))$, and $d_1' = \mathsf{update}(d_1, r) = \mathsf{update}(d_2, r) = d_2'$, we know that $\Sigma_1' \approx_L \Sigma_2'$ and $T_1 \approx_L^{\mathcal{P}} T_2$

Case II: $\mathcal{E}$ ends in SMEI-R

From $\Sigma_1 \approx_L \Sigma_2$, $s_{L2} = C$ and $c_{L2} = \mathsf{skip}$

By assumption, $\mathcal{P}(id.ev(v)) = H$

Subcase i: $s_{H2} \neq C$

The proof proceeds the same as Subcase I.i

Subcase ii: $s_{H2} = C$

From $\Sigma_1 \approx_L \Sigma_2$, we know that $\mathcal{R}_1 = (\rho_1, \mathcal{D}_1) = \mathcal{R}_2 = (\rho_2, \mathcal{D}_2)$, which tells us that

$\mathcal{D}_1(\rho_1, id.ev(v)) = (r, e_L, \rho') = \mathcal{D}_2(\rho_2, id.ev(v))$

Then, SmeI-R may be applied to the second trace with input $\alpha = id.ev(v)$

From $\Sigma_1 \approx_L \Sigma_2$, we know that $d_1 = d_2$

Then from $\mathcal{D}_1(\rho_1, id.ev(v)) = (r, e_L, \rho') = \mathcal{D}_2(\rho_2, id.ev(v))$, we know that

$d_1' = \mathsf{update}(d_1, r) = \mathsf{update}(d_2, r) = d_2'$

From $\Sigma_1 \approx_L \Sigma_2$, we also know that $\sigma_{L1} = \sigma_{L2}$, which also tell us that $\sigma_{L1}(e_L) = c_L = \sigma_{L2}(e_L)$

From $\Sigma_1 \approx_L \Sigma_2$, $\mathcal{D}_1(\rho_1, id.ev(v)) = (r, e_L, \rho') = \mathcal{D}_2(\rho_2, id.ev(v))$,

$d_1' = \mathsf{update}(d_1, r) = \mathsf{update}(d_2, r) = d_2'$, and $\sigma_{L1}(e_L) = c_L = \sigma_{L2}(e_L)$, we know that $\Sigma_1' \approx_L \Sigma_2'$

and $T_1 \approx_L^{\mathcal{P}} T_2$

Case III: $\mathcal{E}$ ends in SmeO-LL

From $\Sigma_1 \approx_L \Sigma_2$, we know that $s_{L1} = s_{L2}$ and $\neg\mathsf{consumer}(\kappa_{L1})$

From this, $\neg\mathsf{consumer}(\kappa_{L2})$

By inspecting the SME rules, one of the following must be true:

$\mathsf{consumer}(\kappa_{L2}) \wedge \mathsf{consumer}(\kappa_{H2})$; or $\neg\mathsf{consumer}(\kappa_{L2}) \wedge \mathsf{producer}(\kappa_{H2})$; or

$\mathsf{consumer}(\kappa_{L2}) \wedge \neg\mathsf{consumer}(\kappa_{H2})$

Then, from $\neg\mathsf{consumer}(\kappa_{L2})$, we know that $\mathsf{producer}(\kappa_{H2})$

Then, SmeO-LL may be applied to the second trace

From $\Sigma_1 \approx_L \Sigma_2$, we know that $d_1 = d_2$ and $\kappa_{L1} = \kappa_{L2}$

From $d_1 = d_2$, $\kappa_{L1} = \kappa_{L2}$, and because the operational semantics are deterministic, $d_2, \kappa_{L2} \xrightarrow{\alpha} \kappa_{L2}'$ and $\kappa_{L2}' = \kappa_{L1}'$

From $\Sigma_1 \approx_L \Sigma_2$ and $\kappa_{L2}' = \kappa_{L1}'$, we know that $\Sigma_1' \approx_L \Sigma_2'$ and $T_1 \approx_L^{\mathcal{P}} T_2$

Case IV: $\mathcal{E}$ ends in SmeO-HH

From $\Sigma_1 \approx_L \Sigma_2$, we know that $s_{L1} = s_{L2}$ and $\mathsf{consumer}(\kappa_{L1})$

By assumption, $\Sigma_1' \approx_L \Sigma_1$

From $s_{L1} = s_{L2}$ and $\mathsf{consumer}(\kappa_{L1})$, we know that $\mathsf{consumer}(\kappa_{L2})$

From $\Sigma_1' \approx_L \Sigma_1$ and definition of $\approx_L^{\mathcal{P}}$ for execution traces, $T_1 \approx_L^{\mathcal{P}} \cdot$

Subcase i: $s_{H2} \neq C$

By assumption, $\neg\mathsf{consumer}(\kappa_{H2})$

Applying the operational semantics rules gives, $d_2, \kappa_{H2} \xrightarrow{\alpha'} \kappa_{H2}'$

Subsubcase a: $\mathcal{P}(\alpha') = H$

Then, SmeO-HH may be applied to the second trace, which tells us $\Sigma_2' \approx_L \Sigma_2$

From $\Sigma_2' \approx_L \Sigma_2$ and definition of $\approx_L^{\mathcal{P}}$ for execution traces, $T_2 \approx_L^{\mathcal{P}} \cdot$

From $\Sigma_1 \approx_L \Sigma_2$, $\Sigma_1' \approx_L \Sigma_1$, and $\Sigma_2' \approx_L \Sigma_2$, we know that $\Sigma_1' \approx_L \Sigma_2'$

From $T_1 \approx_L^{\mathcal{P}} \cdot$ and $T_2 \approx_L^{\mathcal{P}} \cdot$, we know that $T_1 \approx_L^{\mathcal{P}} T_2$

Subsubcase b: $\mathcal{P}(\alpha') = L$ or $\alpha' = \bullet$

The proof proceeds the same as subsubcase a, above, except that SmeO-HL is applied.

Subcase ii: $s_{H2} = C$

Let $\Sigma_2' = \Sigma_2$, which tells us $T_2 \approx_L^{\mathcal{P}} \cdot$

From $\Sigma_1 \approx_L \Sigma_2$, $\Sigma_1' \approx_L \Sigma_1$, and $\Sigma_2' = \Sigma_2$, we know that $\Sigma_1' \approx_L \Sigma_2'$

From $T_1 \approx_L^{\mathcal{P}} \cdot$ and $T_2 \approx_L^{\mathcal{P}} \cdot$, then we know that $T_1 \approx_L^{\mathcal{P}} T_2$

□

Lemma 36 (Weak One-step)**.** *If $T_1 = \mathcal{P} \vdash \Sigma_1 \overset{\alpha_1}{\Longrightarrow} \Sigma_1'$ and $T_2 = \mathcal{P} \vdash \Sigma_2 \overset{\alpha_2}{\Longrightarrow} \Sigma_2'$ with $T_1 \approx_L^{\mathcal{P}} T_2$, $\Sigma_1 \approx_L \Sigma_2$, $\Sigma_1 \not\approx_L \Sigma_1'$, and $\Sigma_2 \not\approx_L \Sigma_2'$ then $\Sigma_1' \approx_L \Sigma_2'$.*

Proof.

We examine each case of $\mathcal{E} :: \mathcal{P} \vdash \Sigma_1 \overset{\alpha}{\Longrightarrow} \Sigma_1'$ The proof is straightforward. We show a few representative cases.

Denote $\Sigma_1 = \mathcal{R}_1, d_1; \kappa_{L1}; \kappa_{H1}$ with $\kappa_{L1} = \sigma_{L1}, c_{L1}, s_{L1}, E_{L1}$ and likewise for $\kappa_{H1}, \Sigma_2, \Sigma_1', \Sigma_2'$

Case I: $\mathcal{E}$ ends in SmeI-NR1

From $\Sigma_1 \not\approx_L \Sigma_1'$, we know that $R_1 \neq R_1'$ or $d_1 \neq d_2'$, which tells us that $T_1 \Downarrow_L^{\mathcal{P}} = (r, \mathsf{emp}, \rho')$ with $r \neq \mathsf{none}$ or $\rho' \neq \rho$

Then, from $T_1 \approx_L^{\mathcal{P}} T_2$ we know that $T_2 \Downarrow_L^{\mathcal{P}} = (r, \mathsf{emp}, \rho')$ and $\mathcal{P}(\alpha_2) = H$

From $\Sigma_1 \approx_L \Sigma_2$, we know that $s_{L2} = C$, $c_{L2} = \mathsf{skip}$, $\mathcal{R}_1 = \mathcal{R}_2$ and $d_1 = d_2$

From all this, we know that the second trace must end in SmeI-NR1 with $\alpha_2 = id'.ev'(v')$

From $\mathcal{R}_1 = \mathcal{R}_2$, $T_1 \Downarrow_L^{\mathcal{P}} = (r, \mathsf{emp}, \rho')$, and $T_2 \Downarrow_L^{\mathcal{P}} = (r, \mathsf{emp}, \rho')$, we know that $\mathcal{R}_1' = \mathcal{R}_2'$

By similar reasoning, $d_1' = \mathsf{update}(d_1, r) = \mathsf{update}(d_2, r) = d_2'$

Thus, from $\Sigma_1 \approx_L \Sigma_2$, $\mathcal{R}_1' = \mathcal{R}_2'$, and $d_1' = d_2'$, we know that $\Sigma_1' \approx_L \Sigma_2'$

Case II: $\mathcal{E}$ ends in SmeI-L

From $\Sigma_1 \approx_L \Sigma_2$, we know that $s_{L2} = C$ and $c_{L2} = \mathsf{skip}$

And from $T_1 \approx_L^{\mathcal{P}} T_2$, we know $T_1 \Downarrow_L^{\mathcal{P}} = id.ev(v) = T_2 \Downarrow_L^{\mathcal{P}}$

By assumption, $\mathcal{P}(id.ev(v)) = L$

From all this, the second trace must end in SmeI-L with $\alpha_2 = id.ev(v)$

From $\Sigma_1 \approx_L \Sigma_2$, we know $\sigma_{L1} = \sigma_{L2}$ which tells us that $\sigma_{L1}(id.ev(v)) = c_L = \sigma_{L2}(id.ev(v))$

Thus, from $\Sigma_1 \approx_L \Sigma_2$ and $\sigma_{L1}(id.ev(v)) = c_L = \sigma_{L2}(id.ev(v))$, we know that $\Sigma'_1 \approx_L \Sigma'_2$

Case III: $\mathcal{E}$ ends in SmeO-LL

By assumption, $\neg\mathsf{consumer}(\kappa_{L1})$ and $\mathcal{P}(\alpha_1) = L$

From $\Sigma_1 \approx_L \Sigma_2$, $s_{L1} = s_{L2}$ and $\alpha_1 = \alpha_2$, $d_1 = d_2$ and $\kappa_{L1} = \kappa_{L2}$

From $\neg\mathsf{consumer}(\kappa_{L1})$ and $s_{L1} = s_{L2}$, it must also be the case that $\neg\mathsf{consumer}(\kappa_{L2})$

By inspecting the SME rules, one of the following must be true:

$\mathsf{consumer}(\kappa_{L2}) \wedge \mathsf{consumer}(\kappa_{H2})$; or $\neg\mathsf{consumer}(\kappa_{L2}) \wedge \mathsf{producer}(\kappa_{H2})$; or

$\mathsf{consumer}(\kappa_{L2}) \wedge \neg\mathsf{consumer}(\kappa_{H2})$

Then, from $\neg\mathsf{consumer}(\kappa_{L2})$, we know that $\mathsf{producer}(\kappa_{H2})$

And from $\mathcal{P}(\alpha_1) = L$ and $\alpha_1 = \alpha_2$, we know that $\mathcal{P}(\alpha_2) = L$

From all this, we know the second trace must end in SmeO-LL.

From $d_1 = d_2$ and $\kappa_{L1} = \kappa_{L2}$ and because the operational semantics are deterministic, we know

$d_2, \kappa_{L2} \xrightarrow{\alpha_2} \kappa'_{L2}$ and $\kappa'_{L2} = \kappa'_{L1}$

Thus, from $\Sigma_1 \approx_L \Sigma_2$ and $\kappa'_{L2} = \kappa'_{L1}$, we know $\Sigma'_1 \approx_L \Sigma'_2$

□

Lemma 37. *If $T = \mathcal{P} \vdash \Sigma_0 \Longrightarrow^* \Sigma_1$ and $\Sigma_1 \approx_L \Sigma_2$, then $T \approx_L^{\mathcal{P}} (T \Longrightarrow \Sigma_2)$*

Proof (sketch): By induction over the length of T. □

A.3 Robust Declassification Proofs

Theorem 12 (Robust Declassification) *$\forall \sigma_1, \sigma_2, \mathcal{P}, \mathcal{R}, \tau_i, \tau_\Delta$, s.t. $\mathcal{P} = (\Gamma, m_I)$ and $\Gamma \subseteq \mathsf{dom}(\sigma_1)$, $\sigma_1 <_A \sigma_2$, $\forall T_1 \in \mathsf{iruns}(\sigma_1, \mathcal{P}, \mathcal{R})$ s.t. T_1 is a complete run, $\forall T_2 \in \mathsf{iruns}(\sigma_2, \mathcal{P}, \mathcal{R})$ s.t. T_2 is a complete run, with $\tau_i = \mathsf{in}(T_1)$, $\mathsf{in}(T_2) = \tau_i \bowtie \tau_\Delta$, $\mathsf{dom}(\tau_\Delta) \cap \Gamma = \emptyset \implies \mathcal{K}(T_1, \sigma_1, \mathcal{P}, \mathcal{R}) \subseteq_{\ll} \mathcal{K}(T_2, \sigma_2, \mathcal{P}, \mathcal{R}) \cup \{\tau_i \,|\, \exists \tau'_i, s.t.\ \tau'_i \preceq \tau_i$ and $(\sigma_2, \mathcal{P}, \mathcal{R}, \tau'_i) \Uparrow\}$.*

Proof.

Want to show: $\forall \tau \in \mathcal{K}(T_1, \sigma_1, \mathcal{P}, \mathcal{R})$, either

(a) $\exists \tau_\Delta$ s.t. $\tau \bowtie \tau_\Delta \in \mathcal{K}(T_2, \sigma_2, \mathcal{P}, \mathcal{R})$; or (b) $\exists \tau'$ s.t. $\tau' \preceq \tau$ and $(\sigma_2, \mathcal{P}, \mathcal{R}, \tau') \Uparrow$

We make the following assumptions throughout:

(A1) $\sigma_1 <_A \sigma_2$; (A2) $T_1 \in \mathsf{iruns}(\sigma_1, \mathcal{P}, \mathcal{R})$; (A3) T_1 is a complete run; (A4) $T_2 \in \mathsf{iruns}(\sigma_2, \mathcal{P}, \mathcal{R})$;

(A5) T_2 is a complete run; (A6) $\mathsf{in}(T_1) = \tau_i$; (A7) $\mathsf{in}(T_2) = \tau_i \bowtie \tau_\Delta$; (A8) $\mathsf{dom}(\tau_\Delta) \cap \Gamma = \emptyset$

Let τ be s.t. $\tau \in \mathcal{K}(T_1, \sigma_1, \mathcal{P}, \mathcal{R})$

Then, $\exists T$ s.t. $\tau = \mathsf{in}(T)$ and $T_1 \approx_L^{\mathcal{P}} T_1$

From (A1) and Lemma 53, either

$\exists \tau_2, T_2'$ s.t. $T_2' \in \mathsf{iruns}(\sigma_2, \mathcal{P}, \mathcal{R})$ with $\mathsf{in}(T_2') = \mathsf{in}(T)$, or

$\exists \tau'$ s.t. $\tau' \preceq \tau$ and $(\sigma_2, \mathcal{P}, \mathcal{R}, \tau') \Uparrow$

In the second case, the second conclusion holds.

Otherwise, we assume the first is true and we know $\mathsf{prog}(T_2', \mathcal{P})$

Then, from Lemma 39, $\tau \approx_L^{\mathcal{P},\mathcal{R}} \tau_i$

From (A8) and Lemma 38, $\forall \alpha \in \tau_\Delta, \mathcal{P}(\alpha) = H_\Delta$

Then from (A7) and Lemma 50, $\mathsf{in}(T_2) \approx_L^{\mathcal{P},\mathcal{R}} \tau_i$

From $\tau \approx_L^{\mathcal{P},\mathcal{R}} \tau_i$ and $\mathsf{in}(T_2) \approx_L^{\mathcal{P},\mathcal{R}} \tau_i$, we know that $\mathsf{in}(T_2) \approx_L^{\mathcal{P},\mathcal{R}} \mathsf{in}(T_2')$

From Lemma 52, either

(1) T_2' is a complete run; or

(2) $\exists T_2'', T_2'''$ s.t. $T_2' = T_2'' \Longrightarrow T_2'''$ with T_2'' a complete run and $\mathsf{in}(T_2'') \approx_L^{\mathcal{P},\mathcal{R}} \mathsf{in}(T_2')$; or

(3) $\exists T_2'''$ s.t. $T_2' \Longrightarrow T_2'''$ and T_2''' is a complete segment

Case I: (2) is true

Then from (A5) and Lemma 44, $T_2'' \approx_L T_2$

Therefore, $\tau \in \mathcal{K}(T_2, \sigma_2, \mathcal{P}, \mathcal{R})$.

Case II: (1) is true

The proof proceeds the same as for **Case I**, except that $T_2'' = T_2'$

Case III: (3) is true

The proof proceeds the same as for **Case I**, except that $T_2'' = T_2' \Longrightarrow T_2'''$.

□

Lemma 38 (New Object H_Δ Label). *If* $\mathsf{dom}(\tau) \cap \Gamma = \emptyset$, *then* $\forall \alpha \in \tau$, $\mathcal{P}(\alpha) = H_\Delta$

Proof (sketch): Follows directly from definition of Γ and H_Δ □

Lemma 39 (Eq Trace Eq Inputs). $\forall \mathcal{P}, \mathcal{R}, \sigma_1, \sigma_2, T_1, T_2$, *s.t.* $T_1 \in \mathsf{iruns}(\sigma_1, \mathcal{P}, \mathcal{R})$, $T_2 \in \mathsf{iruns}(\sigma_2, \mathcal{P}, \mathcal{R})$, $T_1 \approx_L^{\mathcal{P}} T_2$, *then* $\mathsf{in}(T_1) \approx_L^{\mathcal{P},\mathcal{R}} \mathsf{in}(T_2)$.

Proof (sketch): By induction on $\mathsf{len}(T_1)$

Let $\Sigma_1, \Sigma_1', \Sigma_2, \Sigma_2'$ be s.t. $T_1 = \Sigma_1 \Longrightarrow^* \Sigma_1'$ and $T_2 = \Sigma_2 \Longrightarrow^* \Sigma_2'$

Base Case I: $\mathsf{len}(T_1) = 0$

The proof for this case follows from the assumption that $T_1 \Downarrow_L^{\mathcal{P}} = T_2 \Downarrow_L^{\mathcal{P}} = \cdot$ and Lemma 40

Inductive Case II: $\mathsf{len}(T_1) = k + 1, k \geq 1$

We let Σ_1'', α be s.t. $T_1 = \Sigma_1 \overset{\alpha}{\Longrightarrow} \Sigma_1'' \Longrightarrow^* \Sigma_1'$. Then there are 2 subcases

Subcase i: $\Sigma_1 \approx_L \Sigma_1''$. The proof follows from $(\Sigma_1 \approx_L \Sigma_1'') \approx_L^{\mathcal{P},\mathcal{R}} \cdot$ and the IH

Subcase ii: $\Sigma_1 \not\approx_L \Sigma_1''$. The proof follows from Lemma 42, Lemma 41, and the IH □

Lemma 40 (Eq Trace No Inputs). *$\forall \mathcal{P}, \mathcal{R}, T,$ s.t. $T \approx_L^{\mathcal{P}} \cdot$, then $\mathsf{in}(T) \approx_L^{\mathcal{P},\mathcal{R}} \cdot$.*

Proof (sketch): The proof is by induction on $\mathsf{len}(T)$. Then we split the proof on which rule was applied to the trace, and we only need to consider the cases where there is no observable input, which includes SmeI-NR1, SmeI-NR2, SmeO-HL, and SmeO-HH □

Lemma 41 (Eq Trace One Obs). *$\forall \mathcal{P}, \mathcal{R}, T, \Sigma, \Sigma'$ s.t. $T \approx_L^{\mathcal{P}} (\Sigma \Longrightarrow \Sigma')$ with $\Sigma \not\approx_L \Sigma'$, $\mathsf{in}(T) \approx_L^{\mathcal{P},\mathcal{R}} \mathsf{in}(\Sigma \Longrightarrow \Sigma')$*

Proof (sketch): The proof is by induction on $\mathsf{len}(T)$. For the inductive case, when the observable action is the last step, the proof follows from $T \approx_L^{\mathcal{P}} (\Sigma \Longrightarrow \Sigma')$ and Lemma 40. When the observable action is not the last step, the proof follows from the IH. □

Lemma 42 (Eq Exec Trace Split Exists). *$\forall \mathcal{P}, \mathcal{R}, \Sigma_1, \Sigma_1', \Sigma_1'', \Sigma_2, \Sigma_2''$ s.t. $(\Sigma_1 \Longrightarrow \Sigma_1' \Longrightarrow^* \Sigma_1'') \approx_L^{\mathcal{P}} (\Sigma_2 \Longrightarrow^* \Sigma_2')$ with $\Sigma_1 \not\approx_L \Sigma_1'$, then $\exists \Sigma_2''$ s.t. $(\Sigma_2 \Longrightarrow^* \Sigma_2') = (\Sigma_2 \Longrightarrow^* \Sigma_2'' \Longrightarrow^* \Sigma_2')$ with $(\Sigma_1 \Longrightarrow \Sigma_1') \approx_L^{\mathcal{P}} (\Sigma_2 \Longrightarrow^* \Sigma_2'')$ and $(\Sigma_1' \Longrightarrow^* \Sigma_1'') \approx_L^{\mathcal{P}} (\Sigma_2'' \Longrightarrow^* \Sigma_2')$*

Proof (sketch): By induction on $\mathsf{len}(\Sigma_2 \Longrightarrow^* \Sigma_2')$. For the inductive case, when the last step results in an equivalent state, the proof follows from the IH. Otherwise, the last step is an observable step and we make the split here. □

Lemma 43 (Eq Exec Trace Split Exists Back). *$\forall \mathcal{P}, \mathcal{R}, \Sigma_1, \Sigma_1', \Sigma_1'', \Sigma_2, \Sigma_2''$ s.t. $(\Sigma_1 \Longrightarrow^* \Sigma_1' \Longrightarrow \Sigma_1'') \approx_L^{\mathcal{P}} (\Sigma_2 \Longrightarrow^* \Sigma_2')$ with $\Sigma_1' \not\approx_L \Sigma_1''$, then $\exists \Sigma_2''$ s.t. $(\Sigma_2 \Longrightarrow^* \Sigma_2') = (\Sigma_2 \Longrightarrow^* \Sigma_2'' \Longrightarrow^* \Sigma_2')$ with $(\Sigma_1 \Longrightarrow^* \Sigma_1') \approx_L^{\mathcal{P}} (\Sigma_2 \Longrightarrow^* \Sigma_2'')$ and $(\Sigma_1' \Longrightarrow \Sigma_1'') \approx_L^{\mathcal{P}} (\Sigma_2'' \Longrightarrow^* \Sigma_2')$*

Proof (sketch): The proof is similar to Lemma 42. □

Lemma 44 (Eq Interleaving Eq Trace). *$\forall \mathcal{P}, \mathcal{R}, T_1, T_2, \Sigma_1, \Sigma_1', \Sigma_2, \Sigma_2'$ s.t. $T_1 = \Sigma_1 \Longrightarrow^* \Sigma_1'$ is a complete run and $T_2 = \Sigma_2 \Longrightarrow^* \Sigma_2'$ is a complete run with $\Sigma_1 \approx_L \Sigma_2$ and $\mathsf{in}(T_1) \approx_L^{\mathcal{P},\mathcal{R}} \mathsf{in}(T_2)$, imply $T_1 \approx_L^{\mathcal{P}} T_2$*

Proof.

By induction on $\mathsf{len}(T_1)$.

We refer to the following assumptions throughout:

(1) $\mathsf{in}(T_1) \approx_L^{\mathcal{P},\mathcal{R}} \mathsf{in}(T_2)$; (2) $\Sigma_1 \approx_L \Sigma_2$; (3) $\mathsf{consumer}(\Sigma_1')$; (4) $\mathsf{consumer}(\Sigma_2')$

Denote: $\Sigma_1 = \mathcal{R}_1, d_1, \kappa_{L1}, \kappa_{H1}$, $\Sigma_1' = \mathcal{R}_1', d_1', \kappa_{L1}', \kappa_{H1}'$ and similarly for Σ_2, Σ_2'

Base Case I: $\mathsf{len}(T_1) = 0$

By assumption, $T_1 = \Sigma_1 = \Sigma_1'$, which tells us that $T_1 \Downarrow_L^{\mathcal{P}} = \cdot$ and $\mathsf{in}(T_1) = \cdot$

From (1) and $\mathsf{in}(T_1) = \cdot$, $\mathsf{in}(T_2) \approx_L^{\mathcal{P},\mathcal{R}} \cdot$

From (2) and (3), we know that $\mathsf{consumer}(\Sigma_1)$ and $\mathsf{consumer}(\kappa_{L2})$

Then from Lemma 45, $T_2 \Downarrow_L^{\mathcal{P}} = \cdot$

Therefore, from $T_1 \Downarrow_L^{\mathcal{P}} = \cdot$ and $T_2 \Downarrow_L^{\mathcal{P}} = \cdot$, $T_1 \approx_L^{\mathcal{P}} T_2$

Inductive Case II: $\mathsf{len}(T_1) = k+1, k \geq 0$

Assume that the conclusion holds for $\mathsf{len}(T_1) = k$

Let α_1, Σ_1'' be s.t. $\Sigma_1 \stackrel{\alpha_1}{\Longrightarrow} \Sigma_1'' \Longrightarrow^* \Sigma_1'$

We split the proof into separate cases based on which rule the first trace begins with

Subcase i: SMEI-NR1

By assumption, $\mathsf{consumer}(\Sigma_1)$ and (i.1) $\alpha_1 \approx_L^{\mathcal{P},\mathcal{R}} \cdot$, or (i.2) $\exists r, a'$ s.t. $\mathcal{R}(\alpha_1) = (r, a')$

Subsubcase a: (i.1) is true

Then, from the definition of $\approx_L^{\mathcal{P}}$, we know that $\Sigma_1'' \approx_L \Sigma_1$ and (2) tells us $\Sigma_1'' \approx_L \Sigma_2$

From (1) and (3), $\Sigma_1'' \Longrightarrow^* \Sigma_1'$ is a complete run and $\mathsf{in}(\Sigma_1'' \Longrightarrow^* \Sigma_1') \approx_L^{\mathcal{P},\mathcal{R}} \mathsf{in}(T_2)$, meaning we can apply the IH

From IH on $\Sigma_1'' \Longrightarrow^* \Sigma_1'$ and T_2, we know that $(\Sigma_1'' \Longrightarrow^* \Sigma_1') \approx_L^{\mathcal{P}} T_2$

From $\Sigma_1'' \approx_L \Sigma_1$, we know that $(\Sigma_1 \stackrel{\alpha_1}{\Longrightarrow} \Sigma_1'') \approx_L^{\mathcal{P}} \cdot$

Then, from the definition of $\approx_L^{\mathcal{P}}$, $T_1 \approx_L^{\mathcal{P}} T_2$

Subsubcase b: (i.2) is true

From (2), (1), (i.2), and Lemma 46,

$\exists \Sigma_C, \Sigma_2'', \alpha_2$ s.t. $T_2 = \Sigma_2 \Longrightarrow^* \Sigma_C \stackrel{\alpha_2}{\Longrightarrow} \Sigma_2'' \Longrightarrow^* \Sigma_2'$ with $(\Sigma_2 \Longrightarrow^* \Sigma_C) \approx_L^{\mathcal{P}} \cdot$, $\alpha_2 \approx_L^{\mathcal{P},\mathcal{R}} \alpha_1$, and $\mathsf{in}(\Sigma_1'' \Longrightarrow^* \Sigma_1') \approx_L^{\mathcal{P},\mathcal{R}} \mathsf{in}(\Sigma_2'' \Longrightarrow^* \Sigma_2')$

Then from Lemma 48 $\Sigma_C \approx_L \Sigma_2$

From $\Sigma_C \approx_L \Sigma_2$ and (2), we know that $\Sigma_C \approx_L \Sigma_1$

From $\exists r, a'$ s.t. $\mathcal{R}(\alpha_1) = (r, a')$, $\alpha_2 \approx_L^{\mathcal{P},\mathcal{R}} \alpha_1$ and definition of $\approx_L^{\mathcal{P}}$, we know that $\Sigma_1 \not\approx_L \Sigma_1'$ and $\Sigma_C \not\approx_L \Sigma_2''$

Then, from Lemma 36, $\Sigma_1'' \approx_L \Sigma_2''$

From (3) and (4), $\Sigma_1'' \Longrightarrow^* \Sigma_1'$ is a complete run and $\Sigma_2'' \Longrightarrow \Sigma_2'$ is a complete run, so the IH may be applied on $\Sigma_2'' \Longrightarrow^* \Sigma_2'$ and $\Sigma_1'' \Longrightarrow^* \Sigma_1'$

From the IH, we know that $(\Sigma_1'' \Longrightarrow^* \Sigma_1') \approx_L^{\mathcal{P}} (\Sigma_2'' \Longrightarrow^* \Sigma_2')$

And from $\exists r, a'$ s.t. $\mathcal{R}(\alpha_1) = (r, a')$ and $\alpha_2 \approx_L^{\mathcal{P},\mathcal{R}} \alpha_1$ we know that $\mathcal{R}(\alpha_2) = (r, a')$

Thus, $T_1 \approx_L^{\mathcal{P}} T_2$

Subcase ii: SMEI-NR2, SMEI-R, or SMEI-L

The proof for SmeI-NR2 proceeds the same as the proof for subsubcase (i.a), above, and the proofs for SmeI-R or SmeI-L are similar to subsubcase (i.b).

Subcase v: SmeO-LL

By assumption,

$\mathsf{in}(\Sigma_1'' \Longrightarrow^* \Sigma_1') \approx_L^{\mathcal{P},\mathcal{R}} \mathsf{in}(T_1)$; and $\Sigma_1 \not\approx_L \Sigma_1''$; and $\alpha_1 \notin \mathsf{in}(T_1)$

From Lemma 47 gives $\exists \Sigma_2'', \alpha_2$ s.t. $T_2 = \Sigma_2 \overset{\alpha_2}{\Longrightarrow} \Sigma_2'' \Longrightarrow^* \Sigma_2'$ with

$(\Sigma_1 \overset{\alpha_1}{\Longrightarrow} \Sigma_1'') \approx_L^{\mathcal{P}} (\Sigma_2 \overset{\alpha_2}{\Longrightarrow} \Sigma_2'')$ and $\Sigma_1'' \approx_L \Sigma_2''$

From (3) and (4), $\Sigma_1'' \Longrightarrow^* \Sigma_1'$ is a complete run and $\Sigma_2'' \Longrightarrow^* \Sigma_2'$ is a complete run

From $(\Sigma_1 \overset{\alpha_1}{\Longrightarrow} \Sigma_1'') \approx_L^{\mathcal{P}} (\Sigma_2 \overset{\alpha_2}{\Longrightarrow} \Sigma_2'')$, we know that $\mathsf{in}(\Sigma_2'' \Longrightarrow^* \Sigma_2') \approx_L^{\mathcal{P},\mathcal{R}} \mathsf{in}(T_2)$

Then from (1) and $\mathsf{in}(\Sigma_1'' \Longrightarrow^* \Sigma_1') \approx_L^{\mathcal{P},\mathcal{R}} \mathsf{in}(T_1)$, we know that $\mathsf{in}(\Sigma_1'' \Longrightarrow^* \Sigma_1') \approx_L^{\mathcal{P},\mathcal{R}} \mathsf{in}(\Sigma_2'' \Longrightarrow^* \Sigma_2')$

From all of this, we know the IH may be applied on $\Sigma_1'' \Longrightarrow^* \Sigma_1'$ and $\Sigma_2'' \Longrightarrow^* \Sigma_2'$

By applying the IH on $\Sigma_1'' \Longrightarrow^* \Sigma_1'$ and $\Sigma_2'' \Longrightarrow^* \Sigma_2'$, we know that $(\Sigma_1'' \Longrightarrow^* \Sigma_1') \approx_L^{\mathcal{P}} (\Sigma_2'' \Longrightarrow^* \Sigma_2')$

Lemma 49 gives $T_1 \approx_L^{\mathcal{P}} T_2$

Subcase vi: SmeO-LH

The proof for this case proceeds the same as the proof for subcase (v), above.

Subcase vii: SmeO-HH

By assumption, $\Sigma_1'' \approx_L \Sigma_1$ and $\mathsf{in}(\Sigma_1 \Longrightarrow \Sigma_1'') \approx_L^{\mathcal{P},\mathcal{R}} \cdot$.

From (4), $\Sigma_1'' \Longrightarrow^* \Sigma_1'$ is a complete run

From (2) and $\Sigma_1'' \approx_L \Sigma_1$, we know that $\Sigma_1'' \approx_L \Sigma_2$

From $\mathsf{in}(\Sigma_1 \Longrightarrow \Sigma_1'') \approx_L^{\mathcal{P},\mathcal{R}} \cdot$, we know that $\mathsf{in}(\Sigma_1'' \Longrightarrow^* \Sigma_1') \approx_L^{\mathcal{P},\mathcal{R}} \mathsf{in}(T_1)$

Then, from (1), $\mathsf{in}(\Sigma_1'' \Longrightarrow^* \Sigma_1') \approx_L^{\mathcal{P},\mathcal{R}} \mathsf{in}(T_2)$

From all of this, we know the IH may be applied on $\Sigma_1'' \Longrightarrow^* \Sigma_1'$ and T_2

By applying the IH on $\Sigma_1'' \Longrightarrow^* \Sigma_1'$ and T_2, we know that $(\Sigma_1'' \Longrightarrow^* \Sigma_1') \approx_L^{\mathcal{P}} T_2$

Then from $\mathsf{in}(\Sigma_1 \Longrightarrow \Sigma_1'') \approx_L^{\mathcal{P},\mathcal{R}} \cdot$, we know that $T_1 \approx_L^{\mathcal{P}} T_2$

Subcase viii: SmeO-HL

The proof for this case proceeds the same as the proof for in subcase (vii), above.

□

Lemma 45 (High Input Empty Trace)**.** *$\forall \mathcal{P}, \mathcal{R}, T, \Sigma, \Sigma'$ s.t. $\Sigma = \mathcal{R}; \kappa_L; \kappa_H$ and $T = \Sigma \Longrightarrow^* \Sigma'$ with $\mathsf{consumer}(\kappa_L)$ and $\mathsf{in}(T) \approx_L^{\mathcal{P},\mathcal{R}} \cdot$, then $T \Downarrow_L^{\mathcal{P}} = \cdot$*

Proof (sketch): The proof is by induction on $\mathsf{len}(T)$. It uses the definition of $\approx_L^{\mathcal{P},\mathcal{R}}$ and $\Downarrow_L^{\mathcal{P}}$ for traces. □

Lemma 46 (Eq Input Split Exists)**.** *$\forall \mathcal{P}, T_1, T_1', T_2, \Sigma_1, \Sigma_2, \Sigma_2', \alpha_1$ s.t. $T_1 = \Sigma_1 \overset{\alpha_1}{\Longrightarrow} T_1'$, $T_2 = \Sigma_2 \Longrightarrow^* \Sigma_2'$, $\mathsf{consumer}(\Sigma_1)$, $\Sigma_1 \approx_L \Sigma_2$, $\Sigma_1 = \mathcal{R}, d; \kappa_L; \kappa_{H1}$, $\Sigma_2 = \mathcal{R}, d; \kappa_L; \kappa_{H2}$, $\mathsf{in}(T_1) \approx_L^{\mathcal{P},\mathcal{R}} \mathsf{in}(T_2)$, and $\mathsf{proj}_{\mathcal{R},\mathcal{P}}(\alpha_1) \neq \cdot$, then $\exists \Sigma_C, \Sigma_2'', \alpha_2$ s.t. $T_2 = \Sigma_2 \Longrightarrow^* \Sigma_C \overset{\alpha_2}{\Longrightarrow} \Sigma_2'' \Longrightarrow^* \Sigma_2'$ with $\mathsf{proj}_{\mathcal{R},\mathcal{P}}(\alpha_1) = \mathsf{proj}_{\mathcal{R},\mathcal{P}}(\alpha_2)$, $(\Sigma_2 \Longrightarrow^* \Sigma_C) \Downarrow_L^{\mathcal{P}} = \cdot$, and $\mathsf{in}(T_1') \approx_L^{\mathcal{P},\mathcal{R}} \mathsf{in}(\Sigma_2'' \Longrightarrow^* \Sigma_2')$*

Proof (sketch): The proof is by induction on $\mathsf{len}(T_2)$. When the first step of T_2 is not observable, the result follows from the IH. When the step is observable, the result follows from the definition of $\mathsf{proj}_{\mathcal{R},\mathcal{P}}$ and $\Downarrow_L^{\mathcal{P}}$.

□

Lemma 47 (Eq State Out Split Exists)**.** *$\forall \mathcal{P}, \mathcal{R}, \Sigma_1, \Sigma_1', \Sigma_1'', \Sigma_2, \Sigma_2'', \alpha_1$ s.t. $\Sigma_1 \overset{\alpha_1}{\Longrightarrow} \Sigma_1' \Longrightarrow^* \Sigma_1''$ is a complete run and $\Sigma_2 \Longrightarrow^* \Sigma_2''$ is a complete run, with $\Sigma_1 \approx_L \Sigma_2$, $\alpha_1 \notin \mathsf{in}(\Sigma_1 \Longrightarrow^* \Sigma_1'')$, and $\Sigma_1 \not\approx_L \Sigma_1'$, then $\exists \Sigma_2', \alpha_2$ s.t. $\Sigma_2 \overset{\alpha_2}{\Longrightarrow} \Sigma_2' \Longrightarrow^* \Sigma_2''$ with $(\Sigma_1 \overset{\alpha_1}{\Longrightarrow} \Sigma_1') \approx_L^{\mathcal{P}} (\Sigma_2 \overset{\alpha_2}{\Longrightarrow} \Sigma_2')$ and $\Sigma_1' \approx_L \Sigma_2'$*

Proof (sketch): The proof is by case analysis on the first rule applied to T_1. We only need to consider cases where the step produces an observable output, so rules SmeO-LL and SmeO-LH. □

Lemma 48 (Empty Exec Eq State)**.** *$\forall \mathcal{P}, T, \Sigma, \Sigma'$ s.t. $T = \Sigma \Longrightarrow^* \Sigma'$ and $T \Downarrow_L^{\mathcal{P}} = \cdot$, then $\Sigma \approx_L \Sigma'$*

Proof.

By induction on $\mathsf{len}(T)$

Base Case I: $\mathsf{len}(T) = 0$

$\Sigma \approx_L \Sigma'$ follows from $T = \Sigma = \Sigma'$

Inductive Case II: $\mathsf{len}(T) = k + 1$, $k \geq 0$

Assume that the conclusion holds for $\mathsf{len}(T) = k$

Let Σ'' be s.t. $T = \Sigma \overset{\alpha}{\Longrightarrow} \Sigma'' \Longrightarrow^* \Sigma'$

From (1) and definition of $\Downarrow_L^{\mathcal{P}}$ $(\Sigma \Longrightarrow \Sigma'') \Downarrow_L^{\mathcal{P}} = \cdot$ and $(\Sigma'' \Longrightarrow^* \Sigma') \Downarrow_L^{\mathcal{P}} = \cdot$

Then, we only need to consider cases which don't produce observations, which includes rules

SmeI-NR1, SmeI-NR2, SmeO-HH, and SmeO-HL

Subcase i: SmeI-NR1

From definition of $\mathsf{proj}_{\mathcal{R},\mathcal{P}}()$, $\mathcal{D}(\rho, \alpha) = (\rho, \mathsf{none}, \bullet)$ and from this, we know that $\mathcal{R}' = \mathcal{R}$ and $d' = d$

Thus, $\Sigma'' \approx_L \Sigma$ and $\Sigma \approx_L \Sigma'$ follows from the IH

Subcase ii: SmeI-NR2, SmeO-HH, or SmeO-HL

By assumption, $\Sigma' \approx_L \Sigma''$, thus, $\Sigma \approx_L \Sigma'$ follows from the IH

□

Lemma 49 (Eq Exec Concatenation)**.** *$\forall \mathcal{P}, T_1, T_1', T_1'', T_2, T_2', T_2''$, s.t. $T_1 = T_1' \Longrightarrow T_1''$ and $T_2 = T_2' \Longrightarrow T_2''$ with $T_1' \approx_L^{\mathcal{P}} T_2'$ and $T_1'' \approx_L^{\mathcal{P}} T_2''$, then $T_1 \approx_L^{\mathcal{P}} T_2$*

Proof (sketch): The proof is by induction on $\mathsf{len}(T_1'')$ and uses Lemma 43. □

Lemma 50 (Eq High Traces, Eq Traces)**.** *$\forall \mathcal{P}, \mathcal{R}, \tau_1, \tau_2, \tau, \tau_\Delta$ s.t. $\tau_1 = \tau \bowtie \tau_\Delta$ with $\forall \alpha \in \tau_\Delta$, $\mathcal{P}(\alpha) = H$ or $\mathcal{P}(\alpha) = H_\Delta$, and $\tau \approx_L^{\mathcal{P},\mathcal{R}} \tau_2$, imply $\tau_1 \approx_L^{\mathcal{P},\mathcal{R}} \tau_2$*

Proof.

By induction on $\mathsf{len}(\tau_1)$

We make the following assumptions throughout:

(1) $\tau_1 = \tau \bowtie \tau_\Delta$; (2) $\forall \alpha \in \tau_\Delta, \mathcal{P}(\alpha) = H$ or $\mathcal{P}(\alpha) = H_\Delta$; and (3) $\tau \approx_L^{\mathcal{P},\mathcal{R}} \tau_2$

Base Case I: $\mathsf{len}(\tau_1) = 0$

By assumption, $\tau_1 = \cdot$, and from (1) we know that $\tau = \cdot$ and $\tau_\Delta = \cdot$

Then, from (3) $\tau_2 \approx_L^{\mathcal{P},\mathcal{R}} \cdot$, and thus $\tau_1 \approx_L^{\mathcal{P},\mathcal{R}} \tau_2$

Inductive Case II: $\mathsf{len}(\tau_1) = k + 1, k \geq 0$

Let τ_1', α be s.t. $\tau_1 = \tau_1' :: \alpha$, then, from the definition $\bowtie$, either

(II.1) $\exists \tau'$ s.t. $\tau = \tau' :: \alpha$ or

(II.2) $\exists \tau_\Delta'$ s.t. $\tau_\Delta = \tau_\Delta :: \alpha$

Subcase i: (II.1) is true

From (II.1) and definition of $\bowtie$, $\tau_1' = \tau' \bowtie \tau_\Delta$

Subsubcase a: $\alpha \approx_L^{\mathcal{P},\mathcal{R}} \cdot$

By assumption and from (3), $\tau' \approx_L^{\mathcal{P},\mathcal{R}} \tau_2$

Then from (2) and $\tau_1' = \tau' \bowtie \tau_\Delta$, we know that the IH may be applied on τ_1', τ', and τ_2

From applying the IH on τ_1', τ', and τ_2 we know that $\tau_1' \approx_L^{\mathcal{P},\mathcal{R}} \tau_2$

Thus, from the definition of $\approx_L^{\mathcal{P},\mathcal{R}}$ we know that $\tau_1 \approx_L^{\mathcal{P},\mathcal{R}} \tau_2$

Subsubcase b: $\alpha \not\approx_L^{\mathcal{P},\mathcal{R}} \cdot$

From Lemma 51, $\exists \tau_2', \tau_2''$ s.t.

$\tau_2 = \tau_2' :: \tau_2''$, $\tau_2' \approx_L^{\mathcal{P},\mathcal{R}} \tau'$, and $\tau_2'' \approx_L^{\mathcal{P},\mathcal{R}} \alpha$

From (2), $\tau_1' = \tau' \bowtie \tau_\Delta$, and $\tau_2' \approx_L^{\mathcal{P},\mathcal{R}} \tau'$, we know that the IH may be applied on τ_1', τ', and τ_2'

By applying the IH on τ_1', τ', and τ_2', we know that $\tau_1' \approx_L^{\mathcal{P},\mathcal{R}} \tau_2'$

Thus, from the definition of $\approx_L^{\mathcal{P},\mathcal{R}}$ $\tau_1 \approx_L^{\mathcal{P},\mathcal{R}} \tau_2$

Subcase ii: (II.2) is true

From (II.2) and definition of $\bowtie$, $\tau_1' = \tau \bowtie \tau_\Delta'$

By assumption and from (2) and definition of $\approx_L^{\mathcal{P},\mathcal{R}}$, we know that $\alpha \approx_L^{\mathcal{P},\mathcal{R}} \cdot$

From (2), (3), and $\tau_1' = \tau \bowtie \tau_\Delta'$, we know that the IH may be applied on τ_1', τ, and τ_2

By applying the IH on τ_1', τ, and τ_2, we know that $\tau_1' \approx_L^{\mathcal{P},\mathcal{R}} \tau_2$

Then, from the definition of $\approx_L^{\mathcal{P},\mathcal{R}}$, we know that $\tau_1' \approx_L^{\mathcal{P},\mathcal{R}} \tau_1$

Thus, $\tau_1 \approx_L^{\mathcal{P},\mathcal{R}} \tau_2$

□

Lemma 51 (Eq Trace Split Exists). *$\forall \mathcal{P}, \mathcal{R}, \tau_1, \tau_1', \tau_2, \alpha$ s.t. $\tau_1 = \tau_1' :: \alpha$ with $\tau_1 \approx_L^{\mathcal{P},\mathcal{R}} \tau_2$, then $\exists \tau_2', \tau_2''$ s.t. $\tau_2 = \tau_2' :: \tau_2''$ with $\tau_2' \approx_L^{\mathcal{P},\mathcal{R}} \tau_1'$ and $\tau_2'' \approx_L^{\mathcal{P},\mathcal{R}} \alpha$*

Proof (sketch): The proof is by induction on $\mathsf{len}(\tau_2)$ □

Lemma 52 (Trace Truncation, Extension). *$\forall T$ s.t. $\mathsf{prog}(T, \mathcal{P})$, either*

1. *T is a complete run,*

2. *$\exists T', T''$ s.t. $T = T' \Longrightarrow T''$ with $\mathsf{in}(T) \approx_L^{\mathcal{P},\mathcal{R}} \mathsf{in}(T')$, or*

3. *$\exists T'$ s.t. $T \Longrightarrow T'$ and T' is a complete segment*

Proof (sketch): The proof is by induction on the length of T □

Lemma 53 (Active simulate passive multi-step). *If $\Sigma_1 <_A \Sigma_2$, and $\mathcal{P} \vdash \Sigma_1 \stackrel{\tau_1}{\Longrightarrow}^* \Sigma_1'$, then either (1) exists τ_2, Σ_2', and $\mathcal{P} \vdash \Sigma_2 \stackrel{\tau_2}{\Longrightarrow}^* \Sigma_2'$, $\mathsf{in}(\tau_1) = \mathsf{in}(\tau_2)$, and $\Sigma_1' <_A \Sigma_2'$ or (2) exists τ, s.t. $\tau \prec \mathsf{in}(\tau_1)$ and $\mathcal{P} \vdash \Sigma_2(\tau) \Uparrow$.*

Proof.

By induction over the length of τ_1.

Case I: $|\tau_1| = 0$

Conclusion (1) follows from $\Sigma_1' = \Sigma_1$ and $\Sigma_2' = \Sigma_2$

Case II: $|\tau_1| = k + 1$, where $k \geq 0$

By assumption, $\Sigma_1 \stackrel{\tau_1'}{\Longrightarrow}^* \Sigma_1'' \stackrel{\alpha_1}{\Longrightarrow} \Sigma_1'$ and $\tau_1 = \tau_1' :: \alpha_1$

By I.H. on τ_1', there are two subcases

Subcase i

By assumption, $\exists \tau_2', \Sigma_2''$ with $\mathcal{P} \vdash \Sigma_2 \stackrel{\tau_2'}{\Longrightarrow}^* \Sigma_2''$, $\mathsf{in}(\tau_1') = \mathsf{in}(\tau_2')$, and $\Sigma_1'' <_A \Sigma_2''$.

Then by Lemma 54, there are two subsubcases

Subsubcase a

By assumption, $\exists \tau_2'', \Sigma_2'$ with $\mathcal{P} \vdash \Sigma_2'' \stackrel{\tau_2''}{\Longrightarrow}^* \Sigma_2'$, $\mathsf{in}(\alpha_1) = \mathsf{in}(\tau_2'')$, and $\Sigma_1' <_A \Sigma_2'$.

Then, $\mathcal{P} \vdash \Sigma_2 \stackrel{\tau_2'}{\Longrightarrow}^* \Sigma_2'' \stackrel{\tau_2''}{\Longrightarrow}^* \Sigma_2'$ and $\mathsf{in}(\tau_1' \, \alpha_1) = \mathsf{in}(\tau_2' \, \tau_2'')$

Thus, conclusion (1) holds

Subsubcase b

By assumption, $\exists \tau$, s.t. $\tau \prec \mathsf{in}(\alpha_1)$ and $\mathcal{P} \vdash \Sigma_2''(\tau) \Uparrow$.

Then, $\mathcal{P} \vdash \Sigma_2(\mathsf{in}(\tau_2')\ \tau) \Uparrow$. and $\mathsf{in}(\tau_2')\ \tau \prec \mathsf{in}(\tau_1'\ \alpha_1)$

Thus, conclusion (2) holds

Subcase ii

By assumption $\exists \tau$, s.t. $\tau \prec \mathsf{in}(\tau_1')$ and $\mathcal{P} \vdash \Sigma_2(\tau) \Uparrow$ and from (1) $\tau \prec \mathsf{in}(\tau_1'\ \alpha_1)$

Thus, conclusion (2) holds

□

Lemma 54 (Active simulate passive single step)**.** *If* $\Sigma_1 <_A \Sigma_2$, $\mathsf{wf}(\Sigma_1)$, $\mathsf{wf}(\Sigma_2)$ *and* $\mathcal{P} \vdash \Sigma_1 \stackrel{\alpha}{\Longrightarrow} \Sigma_1'$, *then either (1) exists* τ, $\mathcal{P} \vdash \Sigma_2 \stackrel{\tau}{\Longrightarrow}^* \Sigma_2'$, $\mathsf{in}(\alpha) = \mathsf{in}(\tau)$, *and* $\Sigma_1' <_A \Sigma_2'$ *(2) exists* τ, *s.t.* $\tau \prec \mathsf{in}(\alpha)$ *and* $\mathcal{P} \vdash \Sigma_2(\tau) \Uparrow$.

Proof. We examine each case of $\mathcal{E} :: \mathcal{P} \vdash \Sigma_1 \stackrel{\alpha}{\Longrightarrow} \Sigma_1'$

Case I: $\mathcal{E}$ ends in SMEI-NR1

By assumption,

$\Sigma_1 = (\rho, \mathcal{D}), d; \sigma_L, \mathsf{skip}, C, \cdot; \sigma_H, \mathsf{skip}, C, \cdot;\ \alpha = id.\mathsf{Ev}(v);$

$\Sigma_1' = (\rho', \mathcal{D}), d'; \sigma_L, \mathsf{skip}, C, \cdot; \sigma_H, c_H, P, \cdot;\ \mathcal{P}(id.\mathsf{Ev}(v)) = H;$

$\mathcal{D}(\rho, id.\mathsf{Ev}(v)) = (r, \bullet, \rho');\ d' = \mathsf{update}(d, r);\ \sigma_H(id.\mathsf{Ev}(v)) = c_H$

From $\Sigma_1 <_A \Sigma_2$, we know that $\Sigma_2 = (\rho, \mathcal{D}), d; \sigma_{L2}, c_{L2}, C, E_L; \sigma_{H2}, c_{H2}, C, E_H$

and $\sigma_L <_A \sigma_{L2}$ and $\sigma_H <_A \sigma_{H2}$

And from $\mathsf{wf}(\Sigma_2)$, we know that $\Sigma_2 = (\rho, \mathcal{D}), d; \sigma_{L2}, \mathsf{skip}, C, \cdot; \sigma_{H2}, \mathsf{skip}, C, \cdot$

Then, applying SMEI-NR1 on Σ_2 gives $\Sigma_2' = (\rho', \mathcal{D}), d'; \sigma_{L2}, \mathsf{skip}, C, \cdot; \sigma_{H2}, c_{H2}', P, \cdot$, where

$c_{H2}' = \sigma_{H2}(id.\mathsf{Ev}(v))$

From Lemma 55, $\sigma_H(id.\mathsf{Ev}(v)) <_A \sigma_{H2}(id.\mathsf{Ev}(v))$

Thus, $\Sigma_1' <_A \Sigma_2'$

Case II: $\mathcal{E}$ ends in SMEI-NR2, SMEI-R, or SMEI-L

The proofs for these cases are similar to **Case I Case III:** $\mathcal{E}$ ends in SMEO-LL

By assumption,

$\Sigma_1 = \mathcal{R}, d; \kappa_L; \kappa_H$ where $\kappa_L = \sigma_L, c_L, s_L, E_L$

$\Sigma_1' = \mathcal{R}, d; \kappa_L'; \kappa_H$ where $\kappa_L' = \sigma_L', c_L', s_L', E_L'$ and $\mathcal{P}(\alpha) = L$, $s_L = P/LC$, κ_H is in producer state and

$d, \sigma_L, c_L, s_L, E_L \stackrel{\alpha}{\longrightarrow} \sigma_L', c_L', s_L', E_L'$

From $\Sigma_1 <_A \Sigma_2$, we know that $\Sigma_2 = \mathcal{R}, d; \kappa_{L2}; \kappa_{H2}$ where $\kappa_{L2} = \sigma_{L2}, c_{L2}, s_L, (E_L, E_{L2}')$

$\sigma_L <_A \sigma_{L2}$, $c_L <_A c_{L2}$, $\kappa_L <_A \kappa_{L2}$, $\kappa_H <_A \kappa_{H2}$

From κ_H in producer state and $\kappa_H <_A \kappa_{H2}$, we also know that κ_{H2} is in producer state

From Lemma 56, we have 2 cases

Subcase i:

By assumption, $\exists \tau, \kappa'_{L2}.d; \kappa_{L2} \xrightarrow{\tau}{}^{*} \kappa'_{L2}$, $\mathsf{in}(\alpha) = \mathsf{in}(\tau) = \cdot$ and $\kappa'_L <_A \kappa'_{L2}$

From Lemma 59 $\exists \tau', \Sigma'_2$ s.t.

$\mathcal{P} \vdash \Sigma_2 \overset{\tau'}{\Longrightarrow}{}^{*} \Sigma'_2$; $\Sigma'_2 = d, \mathcal{R}; \kappa'_{L2}; \kappa_{H2}$; $\mathsf{producer}(\kappa_{H2})$; and $\mathsf{in}(\tau') = \cdot$

Thus, $\Sigma'_1 <_A \Sigma'_2$

Subcase ii:

By assumption, $\exists \tau. \tau \prec \mathsf{in}(\alpha)$ and $\kappa_{L2}(\tau) \Uparrow$

So there exists τ' s.t. $\tau' \prec \mathsf{in}(\alpha)$ and

$\mathcal{P} \vdash \Sigma_2(\tau') \Uparrow$, where $\tau' = \tau \downarrow^{\mathcal{P}}_{l}$.

Case IV: $\mathcal{E}$ ends in SmeO-LH, SmeO-HH, or SmeO-HL

The proof for these cases is similar to the one for **Case III**

□

Lemma 55. *$\sigma_1 <_A \sigma_2$ implies $\sigma_1(id.\mathsf{Ev}(v)) <_A \sigma_2(id.\mathsf{Ev}(v))$.*

Proof (sketch): Follows directly from the definition of $\sigma_1 <_A \sigma_2$ □

Lemma 56 (Active simulate passive single execution)**.** *If $\kappa_1 <_A \kappa_2$, and $d; \kappa_1 \xrightarrow{\alpha} \kappa'_1$, then either (1) exists τ, κ'_2, $d; \kappa_2 \xrightarrow{\tau}{}^{*} \kappa'_2$, $\mathsf{in}(\alpha) = \mathsf{in}(\tau) = \cdot$, and $\kappa'_1 <_A \kappa'_2$ or (2) exists τ, $\tau \prec \mathsf{in}(\alpha)$, $\kappa_2(\tau) \Uparrow$.*

Proof (sketch): We examine each case of $\mathcal{E} :: d; \kappa_1 \xrightarrow{\alpha} \kappa'_1$. The proof for rule P uses Lemma 57. The case for rule LCtoP uses Lemma 55, and the case for rules PtoLC and PtoC uses Lemma 58. □

Lemma 57. *If $\sigma_1 <_A \sigma_2$, $c_1 <_A c_2$, $d, \sigma_1, c_1 \xrightarrow{\alpha} \sigma'_1, c'_1, E_1$ then exists σ'_2, c'_2, E_2 s.t. $d, \sigma_2, c_2 \xrightarrow{\alpha} \sigma'_2, c'_2, E_2$ and $\sigma'_1 <_A \sigma'_2$, $c'_1 <_A c'_2$ and $E_2 = E_1, E'_2$.*

Lemma 58 (State increases)**.** *If $d, \sigma_1, c_1 \xrightarrow{\alpha} \sigma_2, c_2, E$ and $\mathsf{noUpd}(c_1)$ then $\sigma_1 <_A \sigma_2$.*

Lemma 59 (Single execution to multi-execution)**.**

1. *If $d, \kappa_1 \xrightarrow{\tau}{}^{*} \kappa_2$, $\kappa_L = (\sigma, \mathsf{skip}, C, \cdot)$ and $\Sigma_1 = d, \mathcal{R}; \kappa_L; \kappa_1$ then exists τ', Σ_2, $\mathcal{P} \vdash \Sigma_1 \overset{\tau'}{\Longrightarrow}{}^{*} \Sigma_2$, $\Sigma_2 = d, \mathcal{R}; \kappa_L; \kappa_2$ and $\mathsf{in}(\tau') = \cdot$.*

2. *If $d, \kappa_1 \xrightarrow{\tau}{}^{*} \kappa_2$, $\kappa_H = (_, _, P, _)$ and $\Sigma_1 = d, \mathcal{R}; \kappa_1; \kappa_H$ then exists τ', Σ_2, $\mathcal{P} \vdash \Sigma_1 \overset{\tau'}{\Longrightarrow}{}^{*} \Sigma_2$, $\Sigma_2 = d, \mathcal{R}; \kappa_2; \kappa_H$ and $\mathsf{in}(\tau') = \cdot$.*

Proof (sketch): By induction over the length of $d, \kappa_1 \xrightarrow{\tau}{}^{*} \kappa_2$ (for the first case) and induction over the length of $d, \kappa_1 \xrightarrow{\tau}{}^{*} \kappa_2$ (for the second case). □

A.4 Precision Proofs

Lemma 60 (Precision H). *$\mathcal{P} \vdash T = \Sigma_1 \Longrightarrow^* \Sigma_2$, T is a complete run, $\mathcal{P} \vdash t = d_1, \mathcal{R}_1; \kappa_1 \longrightarrow^* d_2, \mathcal{R}_2; \kappa_2$ is a complete run, $\Sigma_1 \sim^C_H d_1, \mathcal{R}_1; \kappa_1$, and $\mathsf{in}(T) = \mathsf{in}(t)$, implies $\mathsf{out}(T)|^{\mathcal{P}}_H = \mathsf{out}(t)|^{\mathcal{P}}_H$.*

Proof. By induction over the number of complete segments in T

Base case: T contains 0 complete segments

By assumption, $\Sigma_1 \sim^C_H d_1, \mathcal{R}_1; \kappa_1$, $\mathsf{in}(T) = \mathsf{in}(t)$, and $|T| = 0$

From $\mathsf{in}(T) = \mathsf{in}(t)$ and the semantics, we know that $|t| = 0$

Then, from $|T| = 0$ and $|t| = 0$, we know that $\mathsf{out}(T)|^{\mathcal{P}}_H = \mathsf{out}(t)|^{\mathcal{P}}_H = \cdot$.

Inductive case: $T = T' \Longrightarrow T''$ where T' is a complete segment

Let $T' = \Sigma_1 \Longrightarrow^* \Sigma'_1$. Then, by assumption, $\Sigma_1 \sim^C_H d_1, \mathcal{R}_1; \kappa_1$, $\mathsf{in}(T) = \mathsf{in}(t)$, and $\mathsf{in}(T) = \mathsf{in}(T') :: \mathsf{in}(T'')$

Then, $t = t \longrightarrow t''$ where t is a complete segment,

Let $t' = d_1, \mathcal{R}_1; \kappa_1 \longrightarrow^* d'_1, \mathcal{R}'_1; \kappa'_1$, then we have $\mathsf{in}(t) = \mathsf{in}(t') :: \mathsf{in}(t'')$ and $\mathsf{in}(T') = \mathsf{in}(t')$

From Lemma 61, $\mathsf{out}(T')|^{\mathcal{P}}_H = \mathsf{out}(t')|^{\mathcal{P}}_H$ and $\Sigma'_1 \sim^C_H d'_1, \mathcal{R}'_1; \kappa'_1$

Applying the IH on $\Sigma'_1 \Longrightarrow T''$ and $d'_1, \mathcal{R}'_1; \kappa'_1 \longrightarrow t''$ gives us $\mathsf{out}(\Sigma'_1 \Longrightarrow T'')|^{\mathcal{P}}_H = \mathsf{out}(d'_1, \mathcal{R}'_1; \kappa'_1 \longrightarrow t'')|^{\mathcal{P}}_H$

Thus, $\mathsf{out}(T)|^{\mathcal{P}}_H = \mathsf{out}(t)|^{\mathcal{P}}_H = \cdot$.

□

Lemma 61 (Precision H one segment). *$\mathcal{P} \vdash T = \Sigma_1 \Longrightarrow^* \Sigma_2$, T is a complete segment, $\mathcal{P} \vdash t = d_1, \mathcal{R}_1; \kappa_1 \longrightarrow^* d_2, \mathcal{R}_2; \kappa_2$ is a complete segment, $\Sigma_1 \sim^C_H d_1, \mathcal{R}_1; \kappa_1$, and $\mathsf{in}(T) = \mathsf{in}(t)$ implies $\mathsf{out}(T)|^{\mathcal{P}}_H = \mathsf{out}(t)|^{\mathcal{P}}_H$ and $\Sigma_2 \sim^C_H d_2, \mathcal{R}_2; \kappa_2$.*

Proof. By examining the first step of T. When $|T| = 0$, the conclusion trivially holds. We only consider cases where $T = \Sigma_1 \overset{\alpha}{\Longrightarrow} \Sigma'_1 \Longrightarrow^* \Sigma_2$.

Case I: T begins with SmeI-NR1 or SmeI-NR2

By assumption, $\Sigma_1 \sim^C_H d_1, \mathcal{R}_1; \kappa_1$

Then, we know that $t = d_1, \mathcal{R}_1; \kappa_1 \overset{\alpha}{\longrightarrow} d'_1, \mathcal{R}'_1; \kappa'_1 \longrightarrow^* d_2, \mathcal{R}_2; \kappa_2$ via In-H rule and $\Sigma'_1 \sim^P_H d'_1, \mathcal{R}'_1; \kappa'_1$

Then, from Lemma 62, we know that $\mathcal{P} \vdash T \sim^T_H t$

Then from Lemma 63, we have $\mathsf{out}(T)|^{\mathcal{P}}_H = \mathsf{out}(t)|^{\mathcal{P}}_H$ **Case II:** T begins with SmeI-R or SmeI-L

By assumption, $\Sigma_1 \sim^C_H d_1, \mathcal{R}_1; \kappa_1$

Then, we know that $t = d_1, \mathcal{R}_1; \kappa_1 \overset{\alpha}{\longrightarrow} d'_1, \mathcal{R}'_1; \kappa'_1 \longrightarrow^* d_2, \mathcal{R}_2; \kappa_2$ via In-H rule and

$\exists \kappa_L$ s.t., $\Sigma'_1 = d'_1, \mathcal{R}'_1; \kappa_L; \kappa'_1$ and κ'_1 is not in consumer state

Let $T' = \Sigma'_1 \Longrightarrow^* \Sigma_2$ and $t' = d'_1, \mathcal{R}'_1; \kappa'_1 \longrightarrow^* d_2, \mathcal{R}_2; \kappa_2$

Then, from Lemma 65, $T' = T_1 \overset{\bullet}{\Longrightarrow} T_2$, and $\mathsf{fst}(T_2) = d'', \mathcal{R}''; \kappa''_L; \kappa''_H$, $\kappa''_H = (\sigma''_H, c'', P, \cdot)$, and $\mathsf{consumer}(\kappa''_L)$, and $\mathcal{P} \vdash T_1 \sim^{TP}_H d'', \mathcal{R}''; \kappa''_H$.

And from Lemma 64, $\mathsf{out}(T_1)|^{\mathcal{P}}_H = \cdot$ and $d'' = d'_1$, $\mathcal{R}'' = \mathcal{R}'_1$, and $\kappa''_H = \kappa'_1$ and $\neg\mathsf{consumer}(\mathsf{Lexe}(\mathsf{last}(T_1)))$

Then, we know that $\mathsf{fst}(T_2)^P_H d'_1, \mathcal{R}'_1; \kappa'_1$

From Lemma 63 on T_2 and t', we have $\mathsf{out}(T_2)|^{\mathcal{P}}_H = \mathsf{out}(t')|^{\mathcal{P}}_H$

Thus, $\mathsf{out}(T)|^{\mathcal{P}}_H = \mathsf{out}(t)|^{\mathcal{P}}_H$

□

Lemma 62. *$\mathcal{P} \vdash T = \Sigma_1 \Longrightarrow^* \Sigma_2$, T is a complete segment, and $\mathcal{P} \vdash t = d, \mathcal{R}; \kappa_1 \longrightarrow^* \kappa_2$, t is a complete run, and $\Sigma_1 \sim^P_H d, \mathcal{R}; \kappa_1$, implies $\mathcal{P} \vdash T \sim^T_H t$.*

Proof (sketch): By induction over the length of T. □

Lemma 63. *$\mathcal{E} :: \mathcal{P} \vdash T \sim^T_H t$ implies $\mathsf{out}(T)|^{\mathcal{P}}_H = \mathsf{out}(t)|^{\mathcal{P}}_H$.*

Proof (sketch): By induction over the structure of $\mathcal{E}$. □

Lemma 64. *$\mathcal{E} :: \mathcal{P} \vdash T \sim^{TP}_H d, \mathcal{R}; \kappa_H$ implies $\mathsf{out}(T)|^{\mathcal{P}}_H = \cdot$ and $\mathsf{last}(T) \sim^{PP}_H d, \mathcal{R}; \kappa_H$*

Proof (sketch): By induction over the structure of $\mathcal{E}$. □

Lemma 65. *$\mathcal{P} \vdash T = \Sigma_1 \Longrightarrow^* \Sigma_2$, T is a complete segment, and the high execution of Σ_1 is not in consumer state implies $T = T_1 \overset{\alpha}{\Longrightarrow} T_2$, $\mathsf{fst}(T_2) = d, \mathcal{R}; \kappa_L; \kappa_H$, $\kappa_H = (\sigma_H, c, P, \cdot)$, $\mathsf{consumer}(\kappa_L)$, $\mathcal{P} \vdash T_1 \sim^{TP}_H d, \mathcal{R}; \kappa_H$.*

Proof (sketch): By induction over the length of T. □

Lemma 66 (Precision L)**.** *$\Sigma_1 = d_0, \mathcal{R}; (\sigma, \mathsf{skip}, C, \cdot); (\sigma, \mathsf{skip}, C, \cdot)$ $\mathcal{P} \vdash T = \Sigma_1 \Longrightarrow^* \Sigma_2$, $tr = (\sigma, \mathsf{skip}, C, \cdot) \longrightarrow^* \kappa$, and T and tr are complete run starting from consumer configurations, and σ does not leak beyond declassification, and $\mathsf{in}(T) = \mathsf{in}(tr)$, then $\mathsf{out}(T)|^{\mathcal{P}}_L = \mathsf{out}(tr)|^{\mathcal{P}}_L$.*

Proof.

Let $\mathcal{P} \vdash t_1 = d_0, \mathcal{R}; (\sigma, \mathsf{skip}, C, \cdot) \longrightarrow^* \kappa_1$ and $(\tau', \mathcal{R}') = PR(\mathcal{P}, \mathcal{R}, \mathsf{in}(T))$.

By Lemma 71 and Lemma 67,

$\exists t'$ s.t. $\mathcal{P} \vdash t' = d_0, \mathcal{R}'; (\sigma, \mathsf{skip}, C, \cdot) \longrightarrow^* \kappa'$ and $\mathsf{in}(t') = \tau'$ and $\mathsf{out}(T)|^{\mathcal{P}}_L = \mathsf{out}(t')|^{\mathcal{P}}_L$

Because we assume there are no leaks outside declassification, we know that $\mathsf{out}(t')|^{\mathcal{P}}_L = \mathsf{out}(t_1)|^{\mathcal{P}}_L$.

Because we assume the state is compatible with the declassification policy, $\mathsf{out}(t_1)|^{\mathcal{P}}_L = \mathsf{out}(tr)|^{\mathcal{P}}_L$

Thus, $\mathsf{out}(T)|^{\mathcal{P}}_L = \mathsf{out}(tr)|^{\mathcal{P}}_L$

□

Lemma 67 (L eq multi-step). *$\mathcal{P} \vdash \Sigma_1 \sim_L^C d, \mathcal{R}'; \kappa_1 @ (\tau_i, \tau_i')$, $|\tau_i'| = |\mathcal{R}'^*_{\mathcal{P}}(\tau_i')|$, $\mathcal{P} \vdash T = \Sigma_1 \Longrightarrow^* \Sigma_2$, T is a complete run starting from a consumer configuration, $\mathsf{in}(T) = \tau_i$, implies exists t, $\mathcal{P} \vdash t = d, \mathcal{R}'; \kappa_1 \longrightarrow^* d_2, \mathcal{R}_2, \kappa_2$, and $\mathsf{in}(t) = \tau_i'$, and $\mathsf{out}(T)|_L^{\mathcal{P}} = \mathsf{out}(t)|_L^{\mathcal{P}}$.*

Proof. By induction over the length of τ_i.

Base case: $|\tau_i| = 0$

By assumption, $\mathcal{P} \vdash \Sigma_1 \sim_L^C d, \mathcal{R}'; \kappa_1 @ (\tau_i, \tau_i')$

Then, we know that $\tau_i' = \cdot$

Let $t = d, \mathcal{R}'; \kappa_L$ and the conclusion holds

Inductive case: $|\tau_i| = k + 1$

Let $\tau_i = \alpha :: \tau_{i1}$, $T = T' \Longrightarrow T''$ where T' is a complete segment, and $T' = \Sigma_1 \Longrightarrow^* \Sigma_1'$

By assumption, $\mathcal{P} \vdash \Sigma_1 \sim_L^C d, \mathcal{R}'; \kappa_1 @ (\tau_i, \tau_i')$

Then, we know that $\mathcal{R}^*_{\mathcal{P}}(\tau_i) = \mathcal{R}'^*_{\mathcal{P}}(\tau_i')$

We split the proof and consider each case of $\mathcal{R}_{\mathcal{P}}(\alpha)$

Subcase i: $\mathcal{R}_{\mathcal{P}}(\alpha) = \mathsf{NR}$

By assumption and $\mathcal{R}^*_{\mathcal{P}}(\tau_i) = \mathcal{R}'^*_{\mathcal{P}}(\tau_i')$, we know that $\mathcal{R}^*_{\mathcal{P}}(\tau_{i1}) = \mathcal{R}'^*_{\mathcal{P}}(\tau')$

Then, from Lemma 69, $\mathsf{out}(T')|_L^{\mathcal{P}} = \cdot$ and $\mathcal{P} \vdash \Sigma_2 \sim_L^C d, \mathcal{R}'; \kappa_L @ (\tau_{i1}, \tau_i')$

By applying the IH on $\Sigma_2 \Longrightarrow T''$, we know that

$\exists \mathcal{P} \vdash t = d, \mathcal{R}'; \kappa_1 \longrightarrow^* d_2, \mathcal{R}_2, \kappa_2$, with

$\mathsf{in}(t) = \tau_i'$, and $\mathsf{out}(\Sigma_2 \Longrightarrow T'')|_L^{\mathcal{P}} = \mathsf{out}(t)|_L^{\mathcal{P}}$

Then, from $\mathsf{out}(T')|_L^{\mathcal{P}} = \cdot$, we know that $\mathsf{out}(T)|_L^{\mathcal{P}} = \mathsf{out}(t)|_L^{\mathcal{P}}$

Subcase ii: $\mathcal{R}_{\mathcal{P}}(\alpha) \neq \mathsf{NR}$

By assumption and $\mathcal{R}^*_{\mathcal{P}}(\tau_i) = \mathcal{R}'^*_{\mathcal{P}}(\tau_i')$, we know that $\tau_i' = \alpha' :: \tau_{i1}'$ and $\mathcal{R}^*_{\mathcal{P}}(\tau_{i1}) = \mathcal{R}'^*_{\mathcal{P}}(\tau_{i1}')$

Then, from Lemma 68,

$t, \mathcal{P} \vdash t_1 = d, \mathcal{R}'; \kappa_1 \longrightarrow^* d_2, \mathcal{R}_2, \kappa_2$, and $\mathsf{in}(t_1) = \alpha'$, and $\mathsf{out}(T')|_L^{\mathcal{P}} = \mathsf{out}(t_1)|_L^{\mathcal{P}}$, and

$\mathcal{P} \vdash \Sigma_2 \sim_L^C d_2, \mathcal{R}_2; \kappa_2 @ (\tau_{i1}, \tau_{i1}')$

Applying the IH on $\Sigma_2 \Longrightarrow T''$, we know that

$\exists \mathcal{P} \vdash t_2 = d_2, \mathcal{R}_2; \kappa_2 \longrightarrow^* d_2', \mathcal{R}_2', \kappa_2'$, and $\mathsf{in}(t_2) = \tau_{i1}'$, and $\mathsf{out}(\Sigma_2 \Longrightarrow T'')|_L^{\mathcal{P}} = \mathsf{out}(t_2)|_L^{\mathcal{P}}$

Let $t = t_1 :: t_2$, then we know that $\mathsf{out}(T)|_L^{\mathcal{P}} = \mathsf{out}(t)|_L^{\mathcal{P}}$

If the current input is released to low execution, apply Lemma 68. If the current input is not released to low execution, apply Lemma 69. □

Lemma 68 (L eq single-step R). *If* $\mathcal{P} \vdash \Sigma_1 \sim_L^C d, \mathcal{R}'; \kappa_1 @(\alpha\, \tau_i, \alpha'\, \tau_i')$, $\mathcal{R}_{\mathcal{P}}(\alpha) \neq \mathsf{NR}$, $|\alpha'\, \tau_i'| = |\mathcal{R}'^*_{\mathcal{P}}(\alpha'\, \tau_i')|$, $\mathcal{P} \vdash T = \Sigma_1 \Longrightarrow^* \Sigma_2$, *T is a complete segment* $\mathsf{in}(T) = \alpha$, *then exists* t, $\mathcal{P} \vdash t = d, \mathcal{R}'; \kappa_1 \longrightarrow^* d_2, \mathcal{R}_2, \kappa_2$, *and* $\mathsf{in}(t) = \alpha'$, *and* $\mathsf{out}(T)|_L^{\mathcal{P}} = \mathsf{out}(t)|_L^{\mathcal{P}}$, *and* $\mathcal{P} \vdash \Sigma_2 \sim_L^C d_2, \mathcal{R}_2; \kappa_2 @(\tau_i, \tau_i')$

Proof.

By assumption,

$\mathsf{consumer}(\Sigma_1)$ and $\mathsf{consumer}(\Sigma_2)$, $\mathsf{in}(T) = \alpha$, $\mathcal{P} \vdash \Sigma_1 \sim_L^C d, \mathcal{R}'; \kappa_L @(\alpha\, \tau_i, \alpha'\, \tau_i')$, and $\mathcal{R}_{\mathcal{P}}(\alpha) \neq \mathsf{NR}$,

From $\mathcal{P} \vdash \Sigma_1 \sim_L^C d, \mathcal{R}'; \kappa_L @(\alpha\, \tau_i, \alpha'\, \tau_i')$, we know that

$\mathcal{R}_{\mathcal{P}}(\alpha) = \mathcal{R}'_{\mathcal{P}}(\alpha') = \alpha_i$ and $\mathcal{R}1^*_{\mathcal{P}}(\tau_i) = \mathcal{R}1'^*_{\mathcal{P}}(\tau_i')$

where $\mathcal{R}1 = (\rho_1, \mathcal{D})$, $(\rho_1, r, \alpha_i) = \mathcal{D}(\rho, \alpha)$ and $\mathcal{R}1' = (\rho_1', \mathcal{D}')$ is similarly defined

By the semantic rules, SMEI-NR1 or SMEI-R or SMEI-L could apply

Case I: SmeI-NR1 applies

By assumption,

$T = \Sigma_1 \Longrightarrow \Sigma_1' \Longrightarrow^* \Sigma_2$ and $\mathcal{P} \vdash \Sigma_1' \sim_L^{PP} d_1, \mathcal{R}1; \kappa_L$ where $d_1 = \mathsf{update}(d, r)$

From Lemma 70, $\Sigma_1' \Longrightarrow^* \Sigma_2 = T' \Longrightarrow \Sigma_2$ and $\mathcal{P} \vdash T' \sim_L^{TP} d_1, \mathcal{R}1; \kappa_L$ and $\mathcal{P} \vdash \Sigma_2 \sim_L^C d_1, \mathcal{R}1; \kappa_L$

By semantic rules, $d, \mathcal{R}'; \kappa_L$ can step using IN-H rule

Let $\kappa_L = \sigma, \mathsf{skip}, C, \cdot$ and $t = d, \mathcal{R}'; \kappa_L \xrightarrow{\mathsf{emp}} d_1, \mathcal{R}1'; \sigma, \mathsf{skip}, P, \cdot \xrightarrow{\bullet} d_1, \mathcal{R}1'; \kappa_L$

From Lemma 72, $\mathsf{out}(T)|_L^{\mathcal{P}} = \mathsf{out}(t)|_L^{\mathcal{P}} = \cdot$

Thus, $\mathcal{P} \vdash \Sigma_2 \sim_L^C d_1, \mathcal{R}1'; \kappa_L @(\tau_i, \tau_i')$

Case II: SmeI-R or SmeI-L applies.

We only show the case for SMEI-R and the proof for the other case is similar

By assumption,

$T = \Sigma_1 \Longrightarrow \Sigma_1' \Longrightarrow^* \Sigma_2' \Longrightarrow^* \Sigma_2$ where $\Sigma_2' = d_2', \mathcal{R}_2''; \kappa_{L2}'; \kappa_{H2}'$ and $\mathsf{consumer}(\kappa_{L1}')$ and $\neg\mathsf{consumer}(\kappa_{H1}')$ and all low executions between Σ_1' and Σ_2' are not in consumer state

Let $T_1 = \Sigma_1' \Longrightarrow^* \Sigma_2'$ and $T_2 = \Sigma_2' \Longrightarrow^* \Sigma_2$

By semantic rules, $d, \mathcal{R}'; \kappa_1$ can step using IN-H rule: $d, \mathcal{R}'; \kappa_1 \xrightarrow{d}_1, \mathcal{R}1'; \kappa_1'$, and we also know that

$\mathcal{P} \vdash \Sigma_1' \sim_L^P d_1, \kappa_1'$

From Lemma 74, $\exists t$, $\mathcal{P} \vdash t = d_1, \mathcal{R}1'; \kappa_1' \longrightarrow^* d_1, \mathcal{R}1', \kappa_2$, and $\mathcal{P} \vdash T \sim_L^T t$

From, Lemma 73, $\mathsf{out}(T_1)|_L^{\mathcal{P}} = \mathsf{out}(t)|_L^{\mathcal{P}}$, and $\mathcal{P} \vdash \Sigma_2' \sim_L^C d_2, \kappa_2$

Let $\Sigma_2' = d_1, \mathcal{R}_1; \kappa_2; \kappa_H$

From Lemma 70, $\mathcal{P} \vdash \Sigma_2' \Longrightarrow^* \Sigma_2 \sim_L^{TP} d_2, \mathcal{R}1; \kappa_2$ and $\mathcal{P} \vdash \Sigma_2 \sim_L^C d_1, \mathcal{R}; \kappa_2$

From Lemma 72, $\mathsf{out}(\Sigma_2' \Longrightarrow^* \Sigma_2)|_L^{\mathcal{P}} = \cdot$

Then, $\mathsf{out}(T)|_L^{\mathcal{P}} = \mathsf{out}(t)|_L^{\mathcal{P}}$,

Thus, $\mathcal{P} \vdash \Sigma_2 \sim^C_L d_1, \mathcal{R}1'; \kappa_2 @(\tau_i, \tau'_i)$

□

Lemma 69 (L eq single-step NR). *If* $\mathcal{P} \vdash \Sigma_1 \sim^C_L d, \mathcal{R}'; \kappa_L @(\alpha\, \tau_i, \tau'_i)$, $\mathcal{R}_{\mathcal{P}}(\alpha) = \mathsf{NR}$, $\mathcal{P} \vdash T = \Sigma_1 \Longrightarrow^* \Sigma_2$, *$T$ is a complete segment,* $\mathsf{in}(T) = \alpha$, *then* $\mathsf{out}(T)|^{\mathcal{P}}_L = \cdot$ *and* $\mathcal{P} \vdash \Sigma_2 \sim^C_L d, \mathcal{R}'; \kappa_L @(\tau_i, \tau'_i)$

Proof.

Assume $T = \Sigma_1 \Longrightarrow \Sigma_2$.

By assumption,

$\mathsf{consumer}(\Sigma_1)$ and $\mathsf{consumer}(\Sigma_2)$; $\mathsf{in}(T) = \alpha$; $\mathcal{P} \vdash \Sigma_1 \sim^C_L d, \mathcal{R}'; \kappa_L @(\alpha\, \tau_i, \tau'_i)$; and $\mathcal{R}_{\mathcal{P}}(\alpha) = \mathsf{NR}$,

Then, we also know that $\mathcal{R}^*_{\mathcal{P}}(\alpha\tau_i) = \mathcal{R}'^*_{\mathcal{P}}(\tau'_i)$ and $\mathcal{R}^*_{\mathcal{P}}(\tau_i) = \mathcal{R}'^*_{\mathcal{P}}(\tau'_i)$

By the semantic rules, only SmeI-NR1 or SmeI-NR2 could apply

$T = \Sigma_1 \Longrightarrow \Sigma'_1 \Longrightarrow^* \Sigma_2$ and $\mathcal{P} \vdash \Sigma'_1 \sim^{PP}_L d, \mathcal{R}; \kappa_L$

From Lemma 70, $\Sigma'_1 \Longrightarrow^* \Sigma_2 = T' \Longrightarrow \Sigma_2$ and $\mathcal{P} \vdash T' \sim^{TP}_L d, \mathcal{R}; \kappa_L$ and $\mathcal{P} \vdash \Sigma_2 \sim^C_L d, \mathcal{R}; \kappa_L$

From Lemma 72 $\mathsf{out}(T)|^{\mathcal{P}}_L = \cdot$

Thus, $\mathcal{P} \vdash \Sigma_2 \sim^C_L d, \mathcal{R}'; \kappa_L @(\tau_i, \tau'_i)$

□

Lemma 70 (L eq single-step NR producer). *If* $\mathcal{P} \vdash \Sigma_1 \sim^{PP}_L d, \mathcal{R}; \kappa_L$, $\mathcal{P} \vdash T = \Sigma_1 \Longrightarrow^* \Sigma_2$, *$T$ is a complete segment, then* $T = T' \Longrightarrow \Sigma_2$ *and* $\mathcal{P} \vdash T' \sim^{TP}_L d, \mathcal{R}; \kappa_L$ *and* $\mathcal{P} \vdash \Sigma_2 \sim^C_L d, \mathcal{R}; \kappa_L$

Proof. By induction over the length of T.

Base case: $|T| = 1$

Assume $T = \Sigma_1 \Longrightarrow \Sigma_2$.

By assumption, $\mathsf{consumer}(\Sigma_2)$ $\mathcal{P} \vdash \Sigma_1 \sim^{PP}_L d, \mathcal{R}; \kappa_L$,

Then, $\neg\mathsf{consumer}(\mathsf{Hexe}(\Sigma_1))$

By the semantic rules, only SmeO-HL rule could apply

Thus, let $T_1 = \Sigma_1$, $T_2 = \Sigma_2$ and the conclusion holds

Inductive case: $|T| = k + 1$

Assume $T = \Sigma_1 \Longrightarrow T_1$, let $T_1 = \Sigma'_1 \Longrightarrow^* \Sigma_2$

By. I.H on T_1 $T_1 = T'_1 \Longrightarrow \Sigma_2$, and $\mathcal{P} \vdash T'_1 \sim^{TP}_L d, \mathcal{R}; \kappa_L$ and $\mathcal{P} \vdash \Sigma_2 \sim^C_L d, \mathcal{R}; \kappa_L$

Let $T' = \Sigma_1 \Longrightarrow T'_1$, thus, $\mathcal{P} \vdash T' \sim^{TP}_L d, \mathcal{R}; \kappa_L$

□

Lemma 71 (Projected release eq). *$PR(\mathcal{P}, \mathcal{R}, \tau) = (\tau', \mathcal{R}')$ implies* $\mathcal{R}^*_{\mathcal{P}}(\tau) = \mathcal{R}'^*_{\mathcal{P}}(\tau') = \tau'$.

Proof (sketch): By induction over the length of τ. □

Lemma 72. *If* $\mathcal{E} :: \mathcal{P} \vdash \Sigma \overset{\alpha}{\Longrightarrow} T \sim_L^{TP} d, \mathcal{R}; \kappa_L$ *then* $\mathsf{out}(T)|_L^{\mathcal{P}} = \cdot$.

Proof (sketch): By induction on the derivation $\mathcal{E}$. □

Lemma 73. *If* $\mathcal{E} :: \mathcal{P} \vdash T \sim_L^T t$ *then* $\mathsf{out}(T)|_L^{\mathcal{P}} = \mathsf{out}(t)|_L^{\mathcal{P}}$*, and* $\mathcal{P} \vdash \mathsf{last}(T) \sim_L^C \mathsf{last}(t)$.

Proof (sketch): By induction on the derivation $\mathcal{E}$. □

Lemma 74. *If* $\mathcal{P} \vdash \Sigma_1 \sim_L^P d, \kappa_L$*,* $\mathcal{P} \vdash T = \Sigma_1 \Longrightarrow^* \Sigma_2$ *and for all the configurations in* T*, only the low-execution configuration in* Σ_2 *is in consumer state, then for all* $\mathcal{R}'$*, exists* t*,* $\mathcal{P} \vdash t = d, \mathcal{R}'; \kappa_1 \longrightarrow^* d, \mathcal{R}', \kappa_2$*, and* $\mathcal{P} \vdash T \sim_L^T t$.

Proof. By induction over the length of T.

Base case: $|T| = 1$

Assume $T = \Sigma_1 \Longrightarrow \Sigma_2$, $\Sigma_1 = d; \mathcal{R}; \kappa_L; \kappa_H$, and $\Sigma_2 = d'; \mathcal{R}'; \kappa_L'; \kappa_H'$,

By assumption, $\mathsf{consumer}(\Sigma_2)$ and $\neg\mathsf{consumer}(\kappa_L)$ and $\neg\mathsf{consumer}(\kappa_H)$

By the semantic rules, only SmeO-LL or SmeO-LH rule could apply so $d = d'$, $\mathcal{R} = release'$, $\kappa_H = \kappa_H'$

By semantic rules, we can apply Out rule, giving $t = d; \mathcal{R}'; \kappa_L \longrightarrow \kappa_L'$

By applying $\sim_L^T$ rules, $\mathcal{P} \vdash T \sim_L^T t$.

Inductive case: $|T| = k + 1$, we can directly apply I.H.

□

Appendix B

Supporting Materials for Chapter 4

B.1 Additional Definitions

B.1.1 Operations on labels

$\boxed{pc \downarrow^p}$

$$\frac{}{(l_c, l_i) \downarrow^c = l_c} \qquad \frac{}{(l_c, l_i) \downarrow^i = l_i}$$

B.1.2 SME Configuration Equivalence

Two SME configurations are equivalent if they have the same release and endorsement modules and equivalent SME states and configuration stacks. We define a single equivalence definition for both observational ($p = c$) and behavioral equivalence ($p = i$).

Definition 75 (Configuration equivalence)**.** *Given two SME configurations K_1 and K_2 where $K_1 = \mathcal{R}_1, \mathcal{S}_1, \Sigma_1, ks_1$ and $K_2 = \mathcal{R}_2, \mathcal{S}_2, \Sigma_2, ks_2$, $K_1 \approx_l^p K_2$ iff $\mathcal{R}_1 = \mathcal{R}_2$, $\mathcal{S}_1 = \mathcal{S}_2$, $\Sigma_1 \approx_l^p \Sigma_2$, and $ks_1 \approx_l^p ks_2$*

SME state equivalence Two SME states are observationally equivalent at l if their l-observations are the same. Similarly, they are behaviorally equivalent at l if their l-behaviors are the same.

Definition 76 (SME state equivalence)**.** *Given two SME states Σ_1, Σ_2, $\Sigma_1 \approx_l^p \Sigma_2$ iff $\Sigma_1 \downarrow_l^p = \Sigma_2 \downarrow_l^p$*

For the copies of the store at pc s.t. $pc \downarrow^p \sqsubseteq l$, the observation (or behavior) is the store. Otherwise, the observation (or behavior) is the l-projection of the store which includes page elements with label l' s.t. $l' \sqsubseteq l$ and the l-projection of the event handlers. We show the rules for computing the l-projection of an SME store, single store, and event handler map in Figure B.1.

$\boxed{\Sigma \downarrow_l^p = \Sigma'}$

$$\frac{\Sigma = pc \mapsto \sigma^G_{pc}, \Sigma' \qquad pc \downarrow^p \sqsubseteq l}{\Sigma \downarrow_l^p = pc \mapsto \sigma^G_{pc}, \Sigma' \downarrow_l^p} \qquad \frac{\Sigma = pc \mapsto (_, \sigma^{EH}) \qquad pc \downarrow^p \not\sqsubseteq l}{\Sigma \downarrow_l^p = pc \mapsto (\cdot, \sigma^{EH} \downarrow_l^p), \Sigma' \downarrow_l^p} \qquad \frac{}{\cdot \downarrow_l^p = \cdot}$$

$\boxed{\sigma^{EH} \downarrow_l^p = \sigma^{EH'}}$

$$\frac{pc \downarrow^p \sqsubseteq l}{(id \mapsto (v, M, pc), \sigma^{EH}) \downarrow_l^p = (id \mapsto (\mathsf{dv}, M \downarrow_l^p, pc)), \sigma^{EH} \downarrow_l^p} \qquad \frac{pc \downarrow^p \not\sqsubseteq l}{(id \mapsto (v, M, pc), \sigma^{EH}) \downarrow_l^p = \sigma^{EH} \downarrow_l^p}$$

$$\frac{}{\cdot \downarrow_l^p = \cdot}$$

$\boxed{M \downarrow_l^p = M'}$

$$\frac{}{(\mathsf{Ev} \mapsto EH, M) \downarrow_l^p = (\mathsf{Ev} \mapsto EH \downarrow_l^p), M \downarrow_l^p} \qquad \frac{}{\cdot \downarrow_l^p = \cdot}$$

$\boxed{EH \downarrow_l^p = EH'}$

$$\frac{pc \downarrow^p \sqsubseteq l}{(\{(eh, pc)\} \cup EH) \downarrow_l^p = \{(eh, pc)\} \cup EH \downarrow_l^p} \qquad \frac{pc \downarrow^p \not\sqsubseteq l}{(\{(eh, pc)\} \cup EH) \downarrow_l^p = EH \downarrow_l^p} \qquad \frac{}{\cdot \downarrow_l^p = \cdot}$$

Figure B.1: Rules for the observation of an SME store, single store, and event handler map at l (when $p = c$) and behavior of an SME store at l (when $p = i$).

$\boxed{ks \downarrow_l^p = ks'}$

$$\frac{ks = (\kappa, pc_{src}, pc) :: ks' \qquad pc \downarrow^p \sqsubseteq l}{ks \downarrow_l^p = (\kappa, pc_{src}, pc) :: ks' \downarrow_l^p} \qquad \frac{ks = (\kappa, pc_{src}, pc) :: ks' \qquad pc \downarrow^p \not\sqsubseteq l}{ks \downarrow_l^p = ks' \downarrow_l^p} \qquad \frac{}{\cdot \downarrow_l^p = \cdot}$$

Figure B.2: Rules for the observation (when $p = c$) or behavior (when $p = i$) of a configuration stack at l

Configuration stack equivalence Two SME states are observationally equivalent at l if their l-observations are the same. Similarly, they are behaviorally equivalent at l if their l-behaviors are the same. We show the rules for computing the l-projection of a configuration stack in Figure B.2.

Definition 77 (Configuration stack equivalence)**.** *Given two SME states* Σ_1, Σ_2, $\Sigma_1 \approx_l^p \Sigma_2$ *iff* $\Sigma_1 \downarrow_l^p = \Sigma_2 \downarrow_l^p$

B.1.3 Additional syntax/terminology

τ is a sequence of actions visible at some security level. This includes standard actions α, declassifications and endorsements, and the creation of a new element/event handler which is capable of robust declassification or transparent endorsement.

Sequence of actions:	τ	$::=$	$\cdot \mid \tau :: \alpha \mid \tau_{\mathsf{in}} \mid \tau_{\mathsf{down}} \mid \tau_{\mathsf{nm}}$
Input actions:	τ_{in}	$::=$	$\cdot \mid (id.\mathsf{Ev}(v), pc)$
Release actions:	τ_{rls}	$::=$	$\tau_{\mathsf{in}} \mid \mathsf{rls}(id.\mathsf{Ev}(v), \rho, v, E, pc)$
Sanitize actions:	τ_{sntz}	$::=$	$\tau_{\mathsf{in}} \mid \mathsf{sntz}(id.\mathsf{Ev}(v), \rho, v, E, pc)$
Downgraded actions:	τ_{down}	$::=$	$\mathsf{down}(id.\mathsf{Ev}(v), \tau_{\mathsf{rls}}, \tau_{\mathsf{sntz}}, E, pc)$
Nonmalleable actions:	τ_{nm}	$::=$	$\mathsf{r}(id, pc) \mid \mathsf{t}(id, pc) \mid \mid \mathsf{r}(id, eh, pc) \mid \mathsf{t}(id, eh, pc)$

We have two versions of downgrade actions. One includes an updated state and downgraded value (ρ, v, respectively), the other does not. We need both because we prove security against attackers at any place in the security lattice. The first case is when the original event is visible to the attacker. In this case, they would also see the updated state and downgraded value. In the second case, the attacker does not see the event being downgraded, only the events which are downgraded to their visibility.

B.1.4 Execution trace equivalence

Two execution traces are observationally equivalent if their l-observations are the same. Similarly, two execution traces are behaviorally equivalent if their l-behaviors are the same.

Definition 78 (Execution trace equivalence)**.** *Given two execution traces, T, T', $T \approx_l^p T'$ iff $T \downarrow_l^p = T' \downarrow_l^p$*

The trace projection rules for non-input actions are shown in Figure B.3 and rules for inputs are shown in Figure B.4. Trace projection rules for inputs use the helper functions in Figures B.5 and B.6. The helper functions rely on robust and transparent so that declassifications/endorsements only appear in the projected trace if they are robust/transparent. Note that to prove robust declassification, we want to treat all declassifications as trusted so we have separate rules for $p = c$ (rls(_) event only if declassification happens in a visible execution $pc' \downarrow^c \sqsubseteq l$) and $p = i$ (rls(_) for any declassification). We do something similar to prove transparent endorsement.

For inputs, we use helper functions to decide if the declassifications are observable (and likewise for endorsements). This depends on whether the decision to declassify was robust, and whether the declassification policy actually declassified anything. We show the rules for declassification, but the ones for endorsement are similar.

B.1.5 Knowledge definitions

Knowledge

Definition 13 (Attacker Knowledge at l). *Formally, $\mathcal{K}(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l)$ is defined as $\{\tau \mid \exists T' \in \mathsf{runs}(\Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}), T \approx_l^c T', \tau = \mathsf{in}(T')\}$*

$\boxed{T \downarrow_l^p = \tau}$

$$\frac{}{\mathcal{P} \vdash K \downarrow_l^p = \cdot} \text{ TP-BASE}$$

$$\frac{pc \downarrow^p \sqsubseteq l \qquad \alpha \notin \{id.\mathsf{Ev}(v), ch(v)\}}{(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K \stackrel{(\alpha,pc)}{\Longrightarrow} T') \downarrow_l^p = \alpha :: T' \downarrow_l^p} \text{ TP-OUT1}$$

$$\frac{pc \downarrow^p \sqsubseteq l \vee \mathcal{P}(ch) \downarrow^p \sqsubseteq l}{(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K \stackrel{(ch(v),pc)}{\Longrightarrow} T') \downarrow_l^p = ch(v) :: T' \downarrow_l^p} \text{ TP-OUT2}$$

$$\frac{pc \downarrow^p \not\sqsubseteq l \downarrow^p \qquad \mathcal{P}(ch) \downarrow^p \not\sqsubseteq l}{(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K \stackrel{(ch(v),pc)}{\Longrightarrow} T') \downarrow_l^p = T' \downarrow_l^p} \text{ TP-OUT-SILENT}$$

$$\frac{\alpha \notin \{id.\mathsf{Ev}(v), ch(v), \mathsf{new}(_)\} \qquad pc \downarrow^p \not\sqsubseteq l}{(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K \stackrel{(\alpha,pc)}{\Longrightarrow} T') \downarrow_l^p = T' \downarrow_l^p} \text{ TP-OUT-SILENT2}$$

$$\frac{\begin{array}{c}\alpha \in \{\mathsf{new}(id, pc_{src}), \mathsf{newEH}(id, eh, pc_{id}, pc_{src})\} \qquad pc \downarrow^c \not\sqsubseteq l \\ \tau = \mathsf{t}(id, pc) \text{ if } \alpha = \mathsf{new}(...) \wedge pc_{src} \downarrow^c \sqsubseteq pc \downarrow^c \\ \tau = \mathsf{t}(id, eh, pc) \text{ if } \alpha = \mathsf{newEH}(...) \wedge pc_{src} \downarrow^c \sqsubseteq pc \downarrow^c \wedge pc_{id} \downarrow^c \sqsubseteq pc \downarrow^c \qquad \tau = \cdot \text{ otherwise}\end{array}}{(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K \stackrel{(\alpha,pc)}{\Longrightarrow} T') \downarrow_l^c = \tau :: T' \downarrow_l^c} \text{ TP-NEWC}$$

$$\frac{\begin{array}{c}\alpha \in \{\mathsf{new}(id, pc_{src}), \mathsf{newEH}(id, eh, pc_{id}, pc_{src})\} \qquad pc \downarrow^i \not\sqsubseteq l \\ \tau = \mathsf{r}(id, pc) \text{ if } \alpha = \mathsf{new}(...) \wedge pc_{src} \downarrow^i \sqsubseteq pc \downarrow^i \\ \tau = \mathsf{r}(id, eh, pc) \text{ if } \alpha = \mathsf{newEH}(...) \wedge pc_{src} \downarrow^i \sqsubseteq pc \downarrow^i \wedge pc_{id} \downarrow^i \sqsubseteq pc \downarrow^i \qquad \tau = \cdot \text{ otherwise}\end{array}}{(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K \stackrel{(\alpha,pc)}{\Longrightarrow} T') \downarrow_l^i = \tau :: T' \downarrow_l^i} \text{ TP-NEWI}$$

Figure B.3: Trace projection for output and dynamic behaviors

$\mathsf{in}(T)$ is the sequence of input events provided to the system resulting in trace T, which includes both user interactions with the system ($id.\mathsf{Ev}(v)$) and dynamically-generated page elements ($\mathsf{new}(id, pc_{src})$). We consider dynamically-generated page inputs to model an active attacker, who may control some of the code running on the webpage.

Progress-Insensitive (PI) Knowledge At attacker at $l \in \mathcal{L}_c$ should not be able to refine their knowledge of the secret inputs, besides what is leaked by observing that the system makes progress.

Definition 79 (Progress Knowledge at l). *Formally, $\mathcal{K}_p(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l)$ is defined as* $\{\tau_i \mid \exists T' \in \mathsf{runs}(\Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}),\ T \approx_l^c T', \tau_i = \mathsf{in}(T'), \mathsf{prog}(T')\}$

$\mathsf{prog}(T)$ holds if the trace T eventually returns to the consumer state to process another user event.

$$\mathsf{prog}(T) \text{ iff } T = \mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_0 \Longrightarrow^* K \wedge \exists K_C \text{ s.t.}$$
$$\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K \stackrel{\tau}{\Longrightarrow}^* K_C \wedge \mathsf{consumer}(K_C) \wedge \forall \alpha_l \in \tau, \mathsf{output}(\alpha_l)$$

$\mathsf{consumer}(K)$ holds if there are no pending event handlers in the event handler queue (i.e., $K = \mathcal{R}, \mathcal{S}; \Sigma; \cdot$)

$\boxed{T \downarrow_l^p = \tau}$

$$\frac{\begin{array}{c}\mathcal{P}(id.\mathsf{Ev}(v)) = pc' \qquad K = _,_;\Sigma;_ \qquad \Sigma(pc) = (_,\sigma^{EH}) \\ \sigma^{EH}(id) \downarrow^i \not\sqsubseteq pc \downarrow^i \qquad \sigma^{EH}(id) \downarrow^c \not\sqsubseteq pc \downarrow^c \\ \tau = \mathsf{trInput}(pc', pc, id.\mathsf{Ev}(v), l, p)\end{array}}{(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K \stackrel{(id.\mathsf{Ev}(v),pc)}{\Longrightarrow} T') \downarrow_l^p = \tau :: T' \downarrow_l^p} \text{ TP-IN}$$

$$\frac{\begin{array}{c}\mathcal{P}(id.\mathsf{Ev}(v)) = pc' \qquad K = _,_;\Sigma;_ \qquad \Sigma(pc) = (_,\sigma^{EH}) \\ \sigma^{EH}(id) \downarrow^i \sqsubseteq pc \downarrow^i \qquad \sigma^{EH}(id) \downarrow^c \not\sqsubseteq pc \downarrow^c \\ \tau = \mathsf{trRobust}((\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K \stackrel{(id.\mathsf{Ev}(v),pc)}{\Longrightarrow} T'), l, p)\end{array}}{(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K \stackrel{(id.\mathsf{Ev}(v),pc)}{\Longrightarrow} T') \downarrow_l^p = \tau :: T' \downarrow_l^p} \text{ TP-IN-R}$$

$$\frac{\begin{array}{c}\mathcal{P}(id.\mathsf{Ev}(v)) = pc' \qquad K = _,_;\Sigma;_ \qquad \Sigma(pc) = (_,\sigma^{EH}) \\ \sigma^{EH}(id) \downarrow^i \sqsubseteq pc \downarrow^i \qquad \sigma^{EH}(id) \downarrow^c \sqsubseteq pc \downarrow^c \\ \tau = \mathsf{trRobustTransparent}((\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K \stackrel{(id.\mathsf{Ev}(v),pc)}{\Longrightarrow} T'), l, p)\end{array}}{(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K \stackrel{(id.ev(v),pc)}{\Longrightarrow} T') \downarrow_l^p = \tau :: T' \downarrow_l^p} \text{ TP-IN-RT}$$

Figure B.4: Trace projection for inputs

PI Release Knowledge

Definition 80 (Release Knowledge at l)**.** *Formally, $\mathcal{K}_{rp}(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, \alpha_l, l)$ is defined as* $\{\tau_i \mid \exists T' \in \mathsf{runs}(\Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}), T \approx_l^c T', \tau_i = \mathsf{in}(T'), \mathsf{prog}(T'), \tau_r = (\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash \mathsf{last}(T) \stackrel{\alpha_l}{\Longrightarrow} K)) \downarrow_l^c, \mathsf{releaseT}(T', \tau_r, l))\}$

Where $\mathsf{release}(T, \tau, l)$ holds if the trace T will eventually produce the same release event(s), τ, at level l. Note: we use τ here because a downgraded event may appear different to different security levels, so a downgraded event may result in multiple events.

$$\mathsf{releaseT}(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_0 \Longrightarrow^* K, \tau, l) \text{ iff } \exists T, K_C, K' \text{ s.t.}, \mathsf{consumer}(K_C) \wedge$$
$$T = \mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K \Longrightarrow^* K_C \Longrightarrow K' \text{ with } (\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K \Longrightarrow^* K_C) \downarrow_l^c = \cdot \text{ and...}$$
$$T \downarrow_l^c = \begin{cases} \tau & \text{when } \tau = \mathsf{rls}(_) \\ \mathsf{down}(id.\mathsf{Ev}(v), \tau_{\mathsf{rls}}, _, E, pc) & \text{when } \tau = \mathsf{down}(id.\mathsf{Ev}(v), \tau_{\mathsf{rls}}, \tau_{\mathsf{sntz}}, E, pc) \end{cases}$$

PI Transparent Knowledge Secure downgrading involves both confidentiality and integrity. To securely endorse an event, the source should have enough privilege to see the event. When we define equivalent traces, we need to also consider the confidentiality level of the source of events (even those in executions that the attacker does not have enough privilege to see). We define transparent knowledge to measure the amount of information leaked to an attacker when a transparent endorsement originates in an execution

$\boxed{E \downarrow_l^p = E'}$

$$\frac{pc \downarrow^p \sqsubseteq l}{((id.\mathsf{Ev}(v), pc) :: E') \downarrow_l^p = (id.\mathsf{Ev}(v), pc) :: E' \downarrow_l^p} \qquad \frac{pc \downarrow^p \not\sqsubseteq l}{((id.\mathsf{Ev}(v), pc) :: E') \downarrow_l^p = E' \downarrow_l^p}$$

$\boxed{\mathsf{trInput}(pc, pc', id.\mathsf{Ev}(v), l, p) = \tau}$

$$\frac{pc_{src} \downarrow^p \sqcup pc_{\mathcal{P}} \downarrow^p \sqsubseteq l}{\mathsf{trInput}(pc_{\mathcal{P}}, pc_{src}, id.\mathsf{Ev}(v), l, p) = (id.\mathsf{Ev}(v), pc_{src})}\ \textsc{input} \qquad \frac{pc_{src} \downarrow^p \sqcup pc_{\mathcal{P}} \downarrow^p \not\sqsubseteq l}{\mathsf{trInput}(pc', pc, id.\mathsf{Ev}(v), l, p) = \cdot}\ \textsc{noInput}$$

$\boxed{\mathsf{trRobust}(K \overset{(id.\mathsf{Ev}(v), pc)}{\Longrightarrow} T, l, p) = \tau}$

$$\frac{\begin{array}{c} \mathcal{P}(id.\mathsf{Ev}(v)) = pc' \\ \mathcal{R} = (\rho, d) \qquad E = ((id.ev(v), (l_c, l_i)) \mid pc' \downarrow^c \sqsubseteq l_c \sqsubset pc \downarrow^c \wedge l_i = pc \downarrow^i \sqcup pc' \downarrow^i) \\ \mathcal{D}((id.ev(v), pc), pc', \rho) = (\rho', v, E_d) \qquad E' = \mathsf{robust}(\Sigma, E :: E_d, pc) \\ pc, \mathsf{r} \vdash \Sigma, E' \rightsquigarrow ks \qquad \tau' = \mathsf{rls}(id.\mathsf{Ev}(v), \rho', v, E' \downarrow_l^c, pc) \text{ if } \rho \neq \rho' \vee v \neq \mathsf{none} \vee ks \downarrow_l^c \neq \cdot \\ \tau' = \mathsf{trInput}(pc', pc, id.\mathsf{Ev}(v), l, c) \text{ otherwise} \end{array}}{\mathsf{trRobust}((\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash \mathcal{R}, _; \Sigma; _ \overset{(id.\mathsf{Ev}(v), pc)}{\Longrightarrow} T'), l, c) = \tau'}\ \textsc{robust-C}$$

$$\frac{\begin{array}{c} \mathcal{P}(id.\mathsf{Ev}(v)) = pc' \\ \mathcal{R} = (\rho, d) \qquad E = ((id.ev(v), (l_c, l_i)) \mid pc' \downarrow^c \sqsubseteq l_c \sqsubset pc \downarrow^c \wedge l_i = pc \downarrow^i \sqcup pc' \downarrow^i) \\ \mathcal{D}((id.ev(v), pc), pc', \rho) = (\rho', v, E_d) \qquad E' = \mathsf{robust}(\Sigma, E :: E_d, pc) \\ pc, \mathsf{r} \vdash \Sigma, E' \rightsquigarrow ks \qquad \tau' = \mathsf{rls}(id.\mathsf{Ev}(v), \rho', v, E', pc) \text{ if } pc \downarrow^i \sqsubseteq l \text{ and } \rho \neq \rho' \vee v \neq \mathsf{none} \vee ks \neq \cdot \\ \tau' = \mathsf{trInput}(pc', pc, id.\mathsf{Ev}(v), l, i) \text{ otherwise} \end{array}}{\mathsf{trRobust}((\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash \mathcal{R}, _; \Sigma; _ \overset{(id.\mathsf{Ev}(v), pc)}{\Longrightarrow} T'), l, i) = \tau'}\ \textsc{robust-I}$$

Figure B.5: Helper functions for trace projection for input rules

$\boxed{\mathsf{trDowngrade}(K \overset{(id.\mathsf{Ev}(v), pc)}{\Longrightarrow} T, l, p) = \tau}$

$$\frac{\begin{array}{c} K = \mathcal{R}, \mathcal{S}; \Sigma; _ \qquad \alpha_I = (id.\mathsf{Ev}(v), pc) \qquad \mathcal{P}(id.\mathsf{Ev}(v)) = pc' \\ \tau_{\mathsf{in}} = \mathsf{trInput}(pc', pc, id.\mathsf{Ev}(v), l, p) \\ \tau_d = \mathsf{trRobust}((\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K \overset{\alpha_I}{\Longrightarrow} T'), l, p) \qquad \tau_e = \mathsf{trTransparent}((\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K \overset{\alpha_I}{\Longrightarrow} T'), l, p) \\ E_{d,e} = \mathsf{downgrade}_{\mathcal{D},\mathcal{E}}(\mathcal{R}, \mathcal{S}, \Sigma, \alpha_I, pc') \qquad pc, \mathsf{rt} \vdash \Sigma, E_{d,e} \rightsquigarrow ks \qquad ks \downarrow_l^p = \cdot \\ \tau = \tau_{\mathsf{in}} \text{ if } \tau_d \neq \mathsf{rls}(_) \wedge \tau_e \neq \mathsf{sntz}(_) \\ \tau = \tau_d \text{ if } \tau_d = \mathsf{rls}(_) \wedge \tau_e \neq \mathsf{sntz}(_) \qquad \tau = \tau_e \text{ if } \tau_e = \mathsf{sntz}(_) \wedge \tau_d \neq \mathsf{rls}(_) \end{array}}{\mathsf{trDowngrade}((\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K \overset{\alpha_I}{\Longrightarrow} T'), l, p) = \tau}\ \textsc{rls-or-sntz}$$

$$\frac{\begin{array}{c} K = \mathcal{R}, \mathcal{S}; \Sigma; _ \qquad \alpha_I = (id.\mathsf{Ev}(v), pc) \qquad \mathcal{P}(id.\mathsf{Ev}(v)) = pc' \\ \tau_d = \mathsf{trRobust}((\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K \overset{\alpha_I}{\Longrightarrow} T'), l, p) \\ \tau_e = \mathsf{trTransparent}((\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K \overset{\alpha_I}{\Longrightarrow} T'), l, p) \qquad E_{d,e} = \mathsf{downgrade}_{\mathcal{D},\mathcal{E}}(\mathcal{R}, \mathcal{S}, \Sigma, \alpha_I, pc') \\ pc, \mathsf{rt} \vdash \Sigma, E_{d,e} \rightsquigarrow ks \qquad ks \downarrow_l^p \neq \cdot \text{ or } \tau_d = \mathsf{rls}(_) \wedge \tau_e = \mathsf{sntz}(_) \\ \tau = \mathsf{down}(id.\mathsf{Ev}(v), \tau_d, \tau_e, E_{d,e}, pc) \end{array}}{\mathsf{trDowngrade}((\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K \overset{\alpha_I}{\Longrightarrow} T'), l, p) = \tau}\ \textsc{down}$$

Figure B.6: Helper functions for trace projection for input rules

they do not have privilege to see (i.e., they learn that there is an element on that copy of the page which the principal generating the event has enough privilege to see).

Definition 81 (Transparent Knowledge at l). *Formally,* $\mathcal{K}_{tp}(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, \alpha_l, l)$ *is defined as* $\{\tau_i \mid \exists T' \in \mathsf{runs}(\Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}), T \approx_l^c T', \tau_i = \mathsf{in}(T'), \mathsf{prog}(T'), \tau = (\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash \mathsf{last}(T) \xrightarrow{\alpha_l} K)) \downarrow_l^c, \mathsf{transparentT}(T', \tau, l))\}$

Where $\mathsf{transparentT}(T, \tau, l)$ holds if the trace T will eventually produce the same elements capable of transparent endorsement, given by τ. We also need to consider the case where T does *not* produce any elements capable of transparent endorsement. In this case, T has reached a new input event and an equivalent trace should be able to get to a consumer state without producing any visible events (like $\mathsf{t}(_)$). This is why input events are also considered transparent actions, in addition to $\mathsf{t}(_)$. For a similar reason, we need to consider outputs made in executions that are visible to the attacker. A single event may trigger event handlers in several executions, not all of which are visible to the attacker. If an event handler is running in an execution that is visible to the attacker (i.e., the trace is producing $ch(_)$ or $\bullet$ events), then we know an equivalent trace running an event handler in an execution that is *not* visible to the attacker should not produce any visible events (like $\mathsf{t}(_)$).

$$\mathsf{transparentT}(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_0 \Longrightarrow^* K, \tau, l) \text{ iff } \exists T, K', \tau \text{ s.t. } T = \mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K \xRightarrow{\tau}{}^* K' \text{ and...}$$
$$T \downarrow_l^c = \tau \text{ when } \tau = \mathsf{t}(_)$$
$$\mathsf{consumer}(K') \wedge T \downarrow_l^c = \cdot \text{ when } \tau \in \{(id.\mathsf{Ev}(v), _), \mathsf{sntz}(_)\}$$
$$\mathsf{lowEH}(K') \wedge \forall(\alpha, pc) \in \tau', \alpha \in \{ch(_), \bullet\} \wedge pc \downarrow^c \not\sqsubseteq l \text{ when } \tau \in \{ch(_), \bullet\}$$

where $\mathsf{lowEH}(K)$ holds if the current event handler running in K is running with $pc \downarrow^c \sqsubseteq l$

Confidentiality Security (with Declassification)

Definition 18 (Knowledge-based PINI w/ Declassification, Transparency). *A system is progress-insensitive secure against l-observers for $l \in \mathcal{L}_c$ iff given any initial global store Σ_0 and downgrade policy $\mathcal{R}, \mathcal{S}, \mathcal{P}$, it is the case that for all traces T, actions α_l, and configurations K s.t. $(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash T \xrightarrow{\alpha_l} K) \in \mathsf{runs}(\Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P})$, then, the following holds*

- *If* $\mathsf{rlsA}(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash \mathsf{last}(T) \xrightarrow{\alpha_l} K, l)$*:* $\mathcal{K}(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash T \xrightarrow{\alpha_l} K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l) \supseteq_{\preceq} \mathcal{K}_{rp}(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, \alpha_l, l)$
- *If* $\mathsf{trnsprntA}(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash \mathsf{last}(T) \xrightarrow{\alpha_l} K, l)$*:*
  $\mathcal{K}(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash T \xrightarrow{\alpha_l} K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l) \supseteq_{\preceq} \mathcal{K}_{tp}(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, \alpha_l, l)$
- *Otherwise:* $\mathcal{K}(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash T \xrightarrow{\alpha_l} K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l) \supseteq_{\preceq} \mathcal{K}_p(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l)$

where $\mathsf{rlsA}(\mathcal{P},\mathcal{D},\mathcal{E} \vdash T \overset{\alpha_l}{\Longrightarrow} K, l)$ iff $T \downarrow_l^c \in \{\mathsf{rls}(_), \mathsf{down}(_)\}$ and $\mathsf{trnsprntA}(\mathcal{P},\mathcal{D},\mathcal{E} \vdash T \Longrightarrow K, l)$ iff $T \downarrow_l^c \in \{\mathsf{t}(_), (id.\mathsf{Ev}(v), _), \mathsf{sntz}(_), ch(_), \bullet\}$

B.1.6 Influence definitions

Influence

Definition 15 (Attacker Influence over l). *Formally,* $\mathcal{I}(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l)$ *is defined as* $\{\tau \mid \exists T' \in \mathsf{runs}(\Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}),$ $T \approx_l^i T', \tau = \mathsf{in}(T')\}$ $\mathsf{in}(T)$ is the sequence of input events provided to the system resulting in trace T, which includes both user interactions with the system ($id.\mathsf{Ev}(v)$) and dynamically-generated page elements ($\mathsf{new}(id, pc_{src})$).

Integrity Security The possible inputs (including new page elements) supplied by an untrusted attacker should not be refined as more l-trusted actions are taken by the system; if they are, it means the attacker must have influence something trusted.

$$\mathcal{I}(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l) \subseteq_{\preceq} \mathcal{I}(\mathcal{P},\mathcal{D},\mathcal{E} \vdash T \Longrightarrow K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l)$$

Progress-Insensitive (PI) Influence If a loop condition depends on an untrusted value, untrusted parties would be in control of whether any trusted operations following the loop occur. We permit this influence, so we only consider the traces which return to consumer states (i.e., the ones which make progress).

The possible inputs supplied by an attacker at should not be refined as more l-trusted actions are taken by the system, outside of what influence they have over whether the system makes progress.

Definition 82 (Progress Influence over l). *Formally,* $\mathcal{I}_p(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l)$ *is defined as* $\{\tau_i \mid \exists T' \in \mathsf{runs}(\Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}),$ $T \approx_l^i T', \tau_i = \mathsf{in}(T'), \mathsf{prog}(T')\}$

PI Sanitization Influence

Definition 83 (Sanitization Influence). *Formally,* $\mathcal{I}_{ep}(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, \alpha_l, l)$ *is defined as* $\{\tau_i \mid \exists T' \in \mathsf{runs}(\Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}),$ $T \approx_l^i T', \tau_i = \mathsf{in}(T'), \mathsf{prog}(T'), \tau_s = (\mathcal{P},\mathcal{D},\mathcal{E} \vdash \mathsf{last}(T) \overset{\alpha_l}{\Longrightarrow} K)) \downarrow_l^i, \mathsf{sanitizeT}(T', \tau_s, l))\}$

Where $\mathsf{sanitize}(T, \tau, l)$ holds if trace T will eventually produce the same endorsement(s), τ, at level l.

$$\begin{aligned}
&\mathsf{sanitizeT}(\mathcal{P},\mathcal{D},\mathcal{E} \vdash K_0 \Longrightarrow^* K, \tau, l) \text{ iff } \exists T, K_C, K' \text{ s.t., } \mathsf{consumer}(K_C) \wedge \\
&\quad T = \mathcal{P},\mathcal{D},\mathcal{E} \vdash K \Longrightarrow^* K_C \Longrightarrow K' \text{ with } (\mathcal{P},\mathcal{D},\mathcal{E} \vdash K \Longrightarrow^* K_C) \downarrow_l^i = \cdot \text{ and...} \\
&\quad T \downarrow_l^i = \begin{cases} \tau & \text{when } \tau = \mathsf{sntz}(_) \\ \mathsf{down}(id.\mathsf{Ev}(v), _, \tau_{\mathsf{sntz}}, E, pc) & \text{when } \tau = \mathsf{down}(id.\mathsf{Ev}(v), \tau_{\mathsf{rls}}, \tau_{\mathsf{sntz}}, E, pc) \end{cases}
\end{aligned}$$

PI Robust Influence Secure downgrading involves both confidentiality and integrity. To securely declassify an event, the principal triggering the event should trust the source of the event handler. When we define equivalent traces, we need to also consider the integrity level of the source of events (even those in executions that the attacker has direct influence over). We define robust influence to measure the amount of influence the attacker has over robust page elements (i.e., we allow their influence to be refined by the existence of robust page elements that they must not have had influence over).

Definition 84 (Robust Influence over l). *Formally,* $\mathcal{I}_{rp}(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, \alpha_l, l)$ *is defined as* $\{\tau_i \mid \exists T' \in \mathsf{runs}(\Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}), T \approx_l^c T', \tau_i = \mathsf{in}(T'), \mathsf{prog}(T'), \tau = (\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash \mathsf{last}(T) \stackrel{\alpha_l}{\Longrightarrow} K)) \downarrow_l^c, \mathsf{robustT}(T', \tau, l))\}$

Where $\mathsf{robustT}(T, \tau, l)$ holds if the trace T will eventually produce the same elements capable of robust declassification, given by τ. We also need to consider the case where T does *not* produce any elements capable of robust declassification. In this case, T has reached a new input event and an equivalent trace should be able to get to a consumer state without producing any visible events (like $\mathsf{r}(_)$). This is why input events are also considered robust actions, in addition to $\mathsf{r}(_)$. For a similar reason, we need to consider outputs made in executions that are not under the influence of the attacker. A single event may trigger event handlers in several executions, not all of which are under the attacker's influence. If an event handler is running in an execution that is not under the attacker's influence (i.e., the trace is producing $ch(_)$ or $\bullet$ events), then we know an equivalent trace running an event handler in an execution that *is* under the attacker's influence should not produce any visible events (like $\mathsf{r}(_)$).

$$\mathsf{robustT}(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_0 \Longrightarrow^* K, \tau, l) \text{ iff } \exists T, K', \tau \text{ s.t. } T = \mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K \stackrel{\tau}{\Longrightarrow}{}^* K' \text{ and...}$$
$$T \downarrow_l^i = \tau \text{ when } \tau = \mathsf{r}(_)$$
$$\mathsf{consumer}(K') \wedge T \downarrow_l^i = \cdot \text{ when } \tau \in \{(id.\mathsf{Ev}(v), _), \mathsf{rls}(_)\}$$
$$\mathsf{lowEH}(K') \wedge \forall (\alpha, pc) \in \tau, \alpha \in \{ch(_), \bullet\} \wedge pc \downarrow^i \not\sqsubseteq l \text{ when } \tau \in \{ch(_), \bullet\}$$

where $\mathsf{lowEH}(K)$ holds if the current event handler running in K is running with $pc \downarrow^i \sqsubseteq l$

Integrity Security (with Endorsement)

Definition 17 (Influence-based PINI w/ Endorsement, Robustness). *The l-trusted components of a system are progress-insensitive secure against untrusted influence iff given any initial global store Σ_0 and downgrade policy $\mathcal{R}, \mathcal{S}, \mathcal{P}$, it is the case that for all traces T, actions α_l, and configurations K s.t. $(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash T \stackrel{\alpha_l}{\Longrightarrow} K) \in \mathsf{runs}(\Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P})$, then, the following holds*

- *If* $\mathsf{sntzA}(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash \mathsf{last}(T) \stackrel{\alpha_l}{\Longrightarrow} K, l)$: $\mathcal{I}(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash T \stackrel{\alpha_l}{\Longrightarrow} K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l) \supseteq_{\preceq} \mathcal{I}_{ep}(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, \alpha_l, l)$
- *If* $\mathsf{rbstA}(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash \mathsf{last}(T) \stackrel{\alpha_l}{\Longrightarrow} K, l)$: $\mathcal{I}(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash T \stackrel{\alpha_l}{\Longrightarrow} K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l) \supseteq_{\preceq} \mathcal{I}_{rp}(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, \alpha_l, l)$

- *Otherwise:* $\mathcal{I}(\mathcal{P},\mathcal{D},\mathcal{E} \vdash T \xrightarrow{\alpha_l} K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l) \supseteq_{\preceq} \mathcal{I}_p(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l)$

where $\mathsf{sntzA}(\mathcal{P},\mathcal{D},\mathcal{E} \vdash T \xrightarrow{\alpha_l} K, l)$ iff $T \downarrow_l^i \in \{\mathsf{sntz}(_), \mathsf{down}(_)\}$ $\mathsf{rbstA}(\mathcal{P},\mathcal{D},\mathcal{E} \vdash T \Longrightarrow K, l)$ iff $T \downarrow_l^i \in \{\mathsf{r}(_), (id.\mathsf{Ev}(v), _), \mathsf{rls}(_), ch(_), \bullet\}$

B.2 Proofs

B.2.1 Top-level theorems

Theorem 19(a) (Soundness-Confidentiality). *For any downgrade policy $\mathcal{R}, \mathcal{S}, \mathcal{P}$, SME state Σ_0, and for traces, states, and actions T, K, α_l s.t. $\mathcal{P},\mathcal{D},\mathcal{E} \vdash T \xrightarrow{\alpha_l} K \in \mathsf{runs}(\Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P})$, then an attacker's knowledge of events secret to l is not refined:*

- *If* $\mathsf{rlsA}(\mathcal{P},\mathcal{D},\mathcal{E} \vdash \mathsf{last}(T) \xrightarrow{\alpha_l} K, l)$*:* $\mathcal{K}(\mathcal{P},\mathcal{D},\mathcal{E} \vdash T \xrightarrow{\alpha_l} K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l) \supseteq_{\preceq} \mathcal{K}_{rp}(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, \alpha_l, l)$
- *If* $\mathsf{trnsprntA}(\mathcal{P},\mathcal{D},\mathcal{E} \vdash \mathsf{last}(T) \xrightarrow{\alpha_l} K, l)$*:*
  $\mathcal{K}(\mathcal{P},\mathcal{D},\mathcal{E} \vdash T \xrightarrow{\alpha_l} K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l) \supseteq_{\preceq} \mathcal{K}_{tp}(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, \alpha_l, l)$
- *Otherwise:* $\mathcal{K}(\mathcal{P},\mathcal{D},\mathcal{E} \vdash T \xrightarrow{\alpha_l} K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l) \supseteq_{\preceq} \mathcal{K}_p(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l)$

Proof.

The proof is split between three cases depending on the action, shown below. In each case, we want to show that $\exists \tau' \in \mathcal{K}(\mathcal{P},\mathcal{D},\mathcal{E} \vdash T \xrightarrow{\alpha_l} K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l)$ s.t. $\tau \preceq \tau'$ for τ defined below

Case I: $\mathsf{rlsA}(\mathcal{P},\mathcal{D},\mathcal{E} \vdash \mathsf{last}(T) \xrightarrow{\alpha_l} K, l)$

Let $\tau \in \mathcal{K}_{rp}(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, \alpha_l, l)$ and $\tau_r = (\mathcal{P},\mathcal{D},\mathcal{E} \vdash \mathsf{last}(T) \xrightarrow{\alpha_l} K) \downarrow_l^c$

Then from the definition of $\mathcal{K}_{rp}()$, there is a trace for which τ is the input $T_1 = \mathcal{P},\mathcal{D},\mathcal{E} \vdash K_0 \Longrightarrow^* K_1$ and $\tau = \mathsf{in}(T_1)$ and all of the following:

$T_1 \approx_l^c T$, $\mathsf{prog}(T_1)$, and $\mathsf{releaseT}(T_1, \tau_r, l)$

We also know from $\mathcal{K}_{rp}()$ that there is a trace $T = \mathcal{P},\mathcal{D},\mathcal{E} \vdash K_0 \Longrightarrow^* K_2$ and $\mathcal{P},\mathcal{D},\mathcal{E} \vdash K_2 \xrightarrow{\alpha_l} K$

Subcase i: $\tau_r = \mathsf{rls}(_)$

By assumption and from $\mathsf{releaseT}(T_1, \tau_r, l)$, $\exists K_1'$ s.t. $\mathcal{P},\mathcal{D},\mathcal{E} \vdash T_1 \Longrightarrow^* K_1'$ with $(\mathcal{P},\mathcal{D},\mathcal{E} \vdash K_1 \Longrightarrow^* K_1') \downarrow_l^c = \tau_r$

From $T_1 \approx_l^c T$, we know that $(\mathcal{P},\mathcal{D},\mathcal{E} \vdash T \xrightarrow{\alpha_l} K) \approx_l^c (\mathcal{P},\mathcal{D},\mathcal{E} \vdash T_1 \Longrightarrow^* K_1')$

From this and the definition of $\mathcal{K}()$,

$\mathsf{in}(T_1) :: \mathsf{in}(\mathcal{P},\mathcal{D},\mathcal{E} \vdash K_1 \Longrightarrow^* K_1') \in \mathcal{K}(\mathcal{P},\mathcal{D},\mathcal{E} \vdash T \xrightarrow{\alpha_l} K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l)$

Let $\tau' = \mathsf{in}(T_1) :: \mathsf{in}(\mathcal{P},\mathcal{D},\mathcal{E} \vdash K_1 \Longrightarrow^* K_1')$ then

$\tau' \in \mathcal{K}(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash T \xrightarrow{\alpha_l} K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l)$ and $\tau \preceq \tau'$

Subcase ii: $\tau_r = \mathsf{down}(\tau_{\mathsf{rls}}, _)$

From $T_1 \approx_l^c T$ and Lemma 85, $K_1 \approx_l^c K_2$

By assumption and from $K_1 \approx_l^c K_2$, $T_1 \approx_l^c T$ $\mathsf{prog}(T_1)$, $\mathsf{releaseT}(T_1, \tau_r, l)$, and Lemma 106,

$\exists K_1'$ s.t. $\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_1 \Longrightarrow^* K_1'$ with $(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_2 \Longrightarrow K) \approx_l^c (\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_1 \Longrightarrow^* K_1')$

From this and the definition of $\mathcal{K}()$,

$\mathsf{in}(T_1) :: \mathsf{in}(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_1 \Longrightarrow^* K_1') \in \mathcal{K}(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash T \xrightarrow{\alpha_l} K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l)$

Let $\tau' = \mathsf{in}(T_1) :: \mathsf{in}(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_1 \Longrightarrow^* K_1')$ then

$\tau' \in \mathcal{K}(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash T \xrightarrow{\alpha_l} K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l)$ and $\tau \preceq \tau'$

Case II: $\mathsf{trnsprntA}(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash \mathsf{last}(T) \xrightarrow{\alpha_l} K, l)$

Let $\tau \in \mathcal{K}_{tp}(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, \alpha_l, l)$ and $\tau_t = (\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash \mathsf{last}(T) \xrightarrow{\alpha_l} K) \downarrow_l^c$

Then from the definition of $\mathcal{K}_{tp}()$, there is a trace for which τ is the input $T_1 = \mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_0 \Longrightarrow^* K_1$ and $\tau = \mathsf{in}(T_1)$ and all of the following:

$T_1 \approx_l^c T$, $\mathsf{prog}(T_1)$, and $\mathsf{transparentT}(T_1, \tau_t, l)$

We also know from $\mathcal{K}_{tp}()$ that there is a trace $T = \mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_0 \Longrightarrow^* K_2$ with $\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_2 \xrightarrow{\alpha_l} K$

Subcase i: $\tau_t = \mathsf{t}(_)$

By assumption and from $\mathsf{transparentT}(T_1, \tau_t, l)$, $\exists K_1'$ s.t. $\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash T_1 \Longrightarrow^* K_1'$ with $(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_1 \Longrightarrow^* K_1') \downarrow_l^c = \tau_t$

From $T_1 \approx_l^c T$, we know that $(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash T \xrightarrow{\alpha_l} K) \approx_l^c (\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash T_1 \Longrightarrow^* K_1')$

From this and the definition of $\mathcal{K}()$,

$\mathsf{in}(T_1) :: \mathsf{in}(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_1 \Longrightarrow^* K_1') \in \mathcal{K}(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash T \xrightarrow{\alpha_l} K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l)$

Let $\tau' = \mathsf{in}(T_1) :: \mathsf{in}(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_1 \Longrightarrow^* K_1')$ then

$\tau' \in \mathcal{K}(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash T \xrightarrow{\alpha_l} K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l)$ and $\tau \preceq \tau'$

Subcase ii: $\tau_t = \{(id.\mathsf{Ev}(v), _), \mathsf{sntz}(_), ch(_), \bullet\}$

From $T_1 \approx_l^c T$ and Lemma 85, $K_1 \approx_l^c K_2$

By assumption and from $K_1 \approx_l^c K_2$, $T_1 \approx_l^c T$, $\mathsf{prog}(T_1)$, $\mathsf{transparentT}(T_1, \tau_t, l)$, and Lemma 98,

$\exists K_1'$ s.t. $\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_1 \Longrightarrow^* K_1'$ with $(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_2 \Longrightarrow K) \approx_l^c (\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_1 \Longrightarrow^* K_1')$

Then from $T_1 \approx_l^c T$, $(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash T_1 \Longrightarrow^* K_1') \approx_l^c (\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash T \Longrightarrow K)$

From this and the definition of $\mathcal{K}()$,

$\mathsf{in}(T_1) :: \mathsf{in}(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_1 \Longrightarrow^* K_1') \in \mathcal{K}(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash T \xrightarrow{\alpha_l} K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l)$

Let $\tau' = \mathsf{in}(T_1) :: \mathsf{in}(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_1 \Longrightarrow^* K_1')$ then

$\tau' \in \mathcal{K}(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash T \xrightarrow{\alpha_l} K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l)$ and $\tau \preceq \tau'$

Case III: $\neg\mathsf{rlsA}(\mathcal{P},\mathcal{D},\mathcal{E} \vdash \mathsf{last}(T) \xrightarrow{\alpha_l} K, l)$

and $\neg\mathsf{trnsprntA}(\mathcal{P},\mathcal{D},\mathcal{E} \vdash \mathsf{last}(T) \xrightarrow{\alpha_l} K, l)$

Let $\tau \in \mathcal{K}_p(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, \alpha_l, l)$

Then from the definition of $\mathcal{K}_p()$, there is a trace for which τ is the input $T_1 = \mathcal{P},\mathcal{D},\mathcal{E} \vdash K_0 \Longrightarrow^* K_1$ and $\tau = \mathsf{in}(T_1)$ and both of the following:

$T_1 \approx_l^c T$ and $\mathsf{prog}(T_1)$

We also know from $\mathcal{K}_p()$ that there is a trace $T = \mathcal{P},\mathcal{D},\mathcal{E} \vdash K_0 \Longrightarrow^* K_2$ and $\mathcal{P},\mathcal{D},\mathcal{E} \vdash K_2 \xrightarrow{\alpha_l} K$

By assumption and from the definition of rlsA and trnsprntA,

$(\mathcal{P},\mathcal{D},\mathcal{E} \vdash \mathsf{last}(T) \Longrightarrow K) \downarrow_l^c \notin \{\mathsf{rls}(_), \mathsf{down}(_), \mathsf{sntz}(_), \mathsf{t}(_), (id.\mathsf{Ev}(v), _), ch(_), \bullet\}$

From this and the definition of $\downarrow_l^c$ for T, $(\mathcal{P},\mathcal{D},\mathcal{E} \vdash \mathsf{last}(T) \xrightarrow{\alpha_l} K) \downarrow_l^c = \cdot$

Then, $T \approx_l^c (\mathcal{P},\mathcal{D},\mathcal{E} \vdash T \xrightarrow{\alpha_l} K)$

Then from $T_1 \approx_l^c T$, we know that $T_1 \approx_l^c (\mathcal{P},\mathcal{D},\mathcal{E} \vdash T \xrightarrow{\alpha_l} K)$

Let $\tau' = \mathsf{in}(T_1)$

Then from $T_1 \approx_l^c (\mathcal{P},\mathcal{D},\mathcal{E} \vdash T \xrightarrow{\alpha_l} K)$, $\tau' \in \mathcal{K}(\mathcal{P},\mathcal{D},\mathcal{E} \vdash T \xrightarrow{\alpha_l} K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l)$ and $\tau \preceq \tau'$

$\square$

Theorem 19(b) (Soundness-Integrity). *For any downgrade policy $\mathcal{R},\mathcal{S},\mathcal{P}$, SME state Σ_0, and for traces, states, and actions T, K, α_l s.t. $\mathcal{P},\mathcal{D},\mathcal{E} \vdash T \xrightarrow{\alpha_l} K \in \mathsf{runs}(\Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P})$, then an attacker does not have influence over trusted behaviors at l:*

- *If* $\mathsf{sntzA}(\mathcal{P},\mathcal{D},\mathcal{E} \vdash \mathsf{last}(T) \xrightarrow{\alpha_l} K, l)$: $\mathcal{I}(\mathcal{P},\mathcal{D},\mathcal{E} \vdash T \xrightarrow{\alpha_l} K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l) \supseteq_\preceq \mathcal{I}_{ep}(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, \alpha_l, l)$
- *If* $\mathsf{rbstA}(\mathcal{P},\mathcal{D},\mathcal{E} \vdash \mathsf{last}(T) \xrightarrow{\alpha_l} K, l)$: $\mathcal{I}(\mathcal{P},\mathcal{D},\mathcal{E} \vdash T \xrightarrow{\alpha_l} K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l) \supseteq_\preceq \mathcal{I}_{rp}(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, \alpha_l, l)$
- *Otherwise:* $\mathcal{I}(\mathcal{P},\mathcal{D},\mathcal{E} \vdash T \xrightarrow{\alpha_l} K, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l) \supseteq_\preceq \mathcal{I}_p(T, \Sigma_0, \mathcal{R}, \mathcal{S}, \mathcal{P}, l)$

Proof (sketch): The proof is similar to the one for Theorem 19(a). Instead of $p = c$ for $\downarrow_l^p$ and $\approx_l^p$, we use $p = i$. The action considered in each case is different (but dual to the ones from the previous proof) and the same Lemmas are used in the proof. $\square$

B.2.2 Supporting lemmas

Lemma 85 (Equivalent Trace, Equivalent State). *If $T_1 = \mathcal{P},\mathcal{D},\mathcal{E} \vdash K_1 \Longrightarrow^* K_1'$ and $T_2 = \mathcal{P},\mathcal{D},\mathcal{E} \vdash K_2 \Longrightarrow^* K_2'$ with $K_1 \approx_l^p K_2$ and $T_1 \approx_l^p T_2$, then $K_1' \approx_l^p K_2'$*

Proof (sketch):

By induction on $\mathsf{len}(T_1)$ and $\mathsf{len}(T_2)$

Base Case I: $\mathsf{len}(T_1) = 0$ and $\mathsf{len}(T_2) = n$

The conclusion follows from $T_1 \approx_l^p T_2$ and Lemma 86

Base Case II: $\mathsf{len}(T_1) = n$ and $\mathsf{len}(T_2) = 0$

The proof is similar to **Base Case I**

Inductive Case III: $\mathsf{len}(T_1) = n + 1$ and $\mathsf{len}(T_2) = m + 1$

Subcase i: $(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_1'' \Longrightarrow K_1') \downarrow_l^p = \cdot$

The conclusion follows from the IH and Lemma 86

Subcase ii: $(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_2'' \Longrightarrow K_2') \downarrow_l^p = \cdot$

The proof is similar to **Subcase i**

Subcase iii: $(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_1'' \Longrightarrow K_1') \downarrow_l^p \neq \cdot$ and $(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_2'' \Longrightarrow K_2') \downarrow_l^p \neq \cdot$

The conclusion follows from IH and Lemma 91

□

Lemma 86 (Empty Traces, Equivalent States). *If $T = \mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K \Longrightarrow^* K'$ and $T \downarrow_l^p = \cdot$, then $K \approx_l^p K'$*

Proof.

By induction on the length of T. We highlight some interesting cases.

Base Case I: $\mathsf{len}(T) = 0$

By assumption, $T = K$ and $K' = K$

Therefore, $K \approx_l^p K'$

Inductive Case II: $\mathsf{len}(T) = n + 1$

By assumption, $T = \mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K \Longrightarrow^* K_1 \Longrightarrow K_2$

Want to show $K \approx_l^p K_2$

Then from $T \downarrow_l^p = \cdot$, we know that $(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K \Longrightarrow^* K_1) \downarrow_l^p = \cdot$

The IH may be applied on $(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K \Longrightarrow^* K_1)$

IH on $(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K \Longrightarrow^* K_1)$ gives $K \approx_l^p K_1$

Let $T' = \mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_1 \xRightarrow{\alpha_l} K_2$ with $T' \downarrow_l^p = \cdot$

Therefore, from $K \approx_l^p K_1$, want to show $K_1 \approx_l^p K_2$

From $T' \downarrow_l^p = \cdot$ and the definition of $T \downarrow_l^p$, we know that $pc \downarrow^p \not\sqsubseteq l$

From $T' \downarrow_l^p = \cdot$,

When $\alpha_l = \mathsf{new}(id, pc_{src}), pc)$, then $pc \downarrow^p \not\sqsubseteq l$ with $pc_{src} \not\sqsubseteq pc$;

When $\alpha_l = \mathsf{newEH}(id, eh, pc_{id}, pc_{src})$, then $pc \downarrow^p \not\sqsubseteq l$ with $pc_{src} \not\sqsubseteq pc$ or $pc_{id} \not\sqsubseteq pc$

From this, $pc \downarrow^p \not\sqsubseteq l$ and $\alpha_l \in \{(\mathsf{new}(id, pc_{src}), pc), \mathsf{newEH}(id, eh, pc_{id}, pc_{eh})\}$ implies $pc_{src} \downarrow^p \not\sqsubseteq l$ and/or $pc_{id} \downarrow^p \not\sqsubseteq l$

Subcase i: T' ends in In-D

By assumption, all of the following:

$\Sigma_1 = \Sigma_2 = (_, \sigma^{EH})$; $\mathcal{S}_1 = \mathcal{S}_2$, $ks_1 = \cdot$; $\mathcal{P}(id.\mathsf{Ev}(v)) = pc'$; $E = ((id.\mathsf{Ev}(v), pc'') \mid pc \sqcup pc' \sqsubseteq pc'')$; $\mathsf{downgrade}_{\mathcal{D}}(\mathcal{R}_1, \Sigma_1, (id.\mathsf{Ev}(v), pc), pc') = (\mathcal{R}_2, E')$; $\Sigma, E \rightsquigarrow ks$; $pc, \mathsf{r} \vdash \Sigma, E' \rightsquigarrow ks'$; $ks_2 = ks :: ks'$; $\sigma^{EH}(id) \downarrow^i \sqsubseteq pc \downarrow^i$; and $\sigma^{EH}(id) \downarrow^c \not\sqsubseteq pc \downarrow^c$

From $\mathsf{downgrade}_{\mathcal{D}}(\mathcal{R}_1, \Sigma_1, (id.\mathsf{Ev}(v), pc), pc') = (\mathcal{R}_2, E')$ and the definition of $\mathsf{downgrade}_{\mathcal{D}}$, all of the following:

$E_d = ((id.\mathsf{Ev}(v), (l_c, l_i)) \mid pc' \downarrow^c \sqsubseteq l_c \sqsubset pc \downarrow^c \wedge l_i = pc \downarrow^i \sqcup pc' \downarrow^i)$; $\mathcal{R}_1 = (\rho_1, d_1)$; $\mathcal{D}((id.\mathsf{Ev}(v), pc), pc', \rho_1) = (\rho_2, v_d, E'_d)$; $d_2 = \mathsf{update}(d_1, v_d)$; $\mathcal{R}_2 = (\rho_2, d_2)$; and $E' = \mathsf{robust}(\Sigma_1, E_d :: E'_d, pc)$

From $\sigma^{EH}(id) \downarrow^i \sqsubseteq pc \downarrow^i$ and $\sigma^{EH}(id) \downarrow^c \not\sqsubseteq pc \downarrow^c$, we know that $T' \downarrow^p_l = \mathsf{trRobust}(...)$

From $T' \downarrow^p_l = \mathsf{trRobust}(...)$, $T' \downarrow^p_l = \cdot$, and the definition of $\mathsf{trInput}$, $pc \downarrow^p \sqcup pc' \downarrow^p \not\sqsubseteq l$

Then from $E = ((id.\mathsf{Ev}(v), pc'') \mid pc \sqcup pc' \sqsubseteq pc'')$, we know that $E \downarrow^p_l = \cdot$

Then from Lemma 89, $ks \approx^p_l \cdot$

From the definition of $\mathsf{trRobust}$, $\mathcal{D}((id.\mathsf{Ev}(v), pc), pc', \rho_1) = (\rho_1, \mathsf{none}, E'_d)$ and $ks' \downarrow^p_l = \cdot$

Then from $ks_1 = \cdot$ and $ks_2 = ks :: ks'$, we know that $ks_1 \approx^p_l ks_2$

From $\mathcal{D}((id.\mathsf{Ev}(v), pc), pc', \rho_1) = (\rho_2, v_d, E'_d) = (\rho_1, \mathsf{none}, E'_d)$ and $d_2 = \mathsf{update}(d_1, v_d)$, we know that $d_2 = d_1$

From $\mathcal{D}((id.\mathsf{Ev}(v), pc), pc', \rho_1) = (\rho_2, v_d, E'_d) = (\rho_1, \mathsf{none}, E'_d)$ and $d_2 = d_1$, we know that $\mathcal{R}_1 = \mathcal{R}_2$

Therefore, from $\mathcal{R}_1 = \mathcal{R}_2$, $\mathcal{S}_1 = \mathcal{S}_2$, $\Sigma_1 = \Sigma_2$, and $ks_1 \approx^p_l ks_2$, we know that $K_1 \approx^p_l K_2$

Subcase ii: T' ends in Out

By assumption, all of the following:

$\mathcal{R}_1 = \mathcal{R}_2$; $\mathcal{S}_1 = \mathcal{S}_2$; $ks_1 = (\kappa, pc_{src}, pc) :: ks$; $\mathcal{F} :: pc_{src}, d_d, d_e \vdash \Sigma_1, \kappa \longrightarrow_{pc} \Sigma_2, ks'$; $ks_2 = ks' :: ks$; and $\alpha = ch(v)$

By assumption and from $T' \downarrow^p_l = \cdot$, $\alpha = ch(v)$, and the definition of $T \downarrow^p_l$, it must be the case that $pc \downarrow^p \not\sqsubseteq l$

Then from Lemma 87, $\Sigma_1 \approx^p_l \Sigma_2$ and $(\kappa, pc_{src}, pc) \approx^p_l ks'$

From this and $pc \downarrow^p \not\sqsubseteq l$, we know that $ks' \downarrow^p_l = \cdot$ and $(\kappa, pc_{src}, pc) \downarrow^p_l = \cdot$

Then from $ks_1 = (\kappa, pc_{src}, pc) :: ks$, $ks_2 = ks' :: ks$, and $(\kappa, pc_{src}, pc) \approx^p_l ks'$ $ks_1 \approx^p_l ks_2$

Therefore, from $\mathcal{R}_1 = \mathcal{R}_2$, $\mathcal{S}_1 = \mathcal{S}_2$, $\Sigma_1 \approx^p_l \Sigma_2$, and $ks_1 \approx^p_l ks_2$, we know that $K_1 \approx^p_l K_2$

Subcase iii: T' ends in OUT-NEXT

The proof for this case follows from the definition of $T \downarrow_l^p$

$\square$

Lemma 87. *If* $pc_{src}, d_d, d_e \vdash \Sigma_1, \kappa \xrightarrow{\alpha}_{pc} \Sigma_2, ks$ *with* $pc \downarrow^p \not\sqsubseteq l$ *and* $\alpha \in \{\mathsf{new}(id, pc_{src}), \mathsf{newEH}(id, eh, pc_{src})\}$ *implies* $pc_{src} \downarrow^p \not\sqsubseteq l$ *and/or* $pc_{id} \downarrow^p \not\sqsubseteq l$*, then* $\Sigma_1 \approx_l^p \Sigma_2$ *and* $(\kappa, pc_{src}, pc) \approx_l^p ks$

Proof (sketch):

We examine each case of $\mathcal{F} :: pc_{src}, d_d, d_e \vdash \Sigma_1, \kappa \longrightarrow_{pc} \Sigma_2, ks$. The proof uses Lemmas 89 and 88.

$\square$

Lemma 88. *If* $pc_{src}, d_d, d_e \vdash \Sigma_1, \sigma_1, c_1 \xrightarrow{\alpha}_{pc} \Sigma_2, \sigma_2, c_2, E$ *with* $pc \downarrow^p \not\sqsubseteq l$ *and* $\alpha \in \{\mathsf{new}(id, pc_{src}), \mathsf{newEH}(id, eh, pc_{src})\}$ *implies* $pc_{src} \downarrow^p \not\sqsubseteq l$ *and/or* $pc_{id} \downarrow^p \not\sqsubseteq l$*, then* $\Sigma_1 \approx_l^p \Sigma_2$

Proof (sketch): By induction on the structure of $\mathcal{F} :: pc_{src}, d_d, d_e \vdash \Sigma_1, \sigma_1, c_1 \longrightarrow_{pc} \Sigma_2, \sigma_2, c_2, E$ $\square$

Lemma 89 (Secret EH Lookups are Not Observable)**.** *If* $\Sigma, E \rightsquigarrow ks$ *with* $E \downarrow_l^p = \cdot$ *then* $ks \approx_l^p \cdot$

Proof.

By induction on the structure of $\mathcal{F} :: \Sigma, E \rightsquigarrow ks$.

Case I: $\mathcal{F}$ ends in LOOKUP

By assumption, all of the following:

$E = (id.\mathsf{Ev}(v), pc) :: E'$; $\Sigma(pc) = (_, \sigma^{EH})$ and $\sigma^{EH}(id) = (_, M, pc_{id})$; $\exists \mathcal{G} :: pc, pc_{id}, v \vdash M(\mathsf{Ev}) \rightsquigarrow ks_1$; $\exists \mathcal{G}' :: \Sigma, E' \rightsquigarrow ks_2$; and $ks = ks_1 :: ks_2$

From $E \downarrow^p = \cdot$ and $E = (id.\mathsf{Ev}(v), pc) :: E'$, we know that $pc \downarrow^p \not\sqsubseteq l$ and $E' \downarrow_l^p = \cdot$

From Lemma 90, $ks_1 \approx_l^p \cdot$

Applying the IH on $\mathcal{G}$, gives $ks_2 \approx_l^p \cdot$

Therefore, from $ks = ks_1 :: ks_2$, $ks_1 \approx_l^p \cdot$, and $ks_2 \approx_l^p \cdot$, we have $ks \approx_l^p \cdot$

Case II: $\mathcal{F}$ ends in LOOKUP-MISSING

The proof for this case follows from the IH

Case III: $\mathcal{F}$ ends in LOOKUP-EMPTY

By assumption, $ks = \cdot$

$\square$

Lemma 90. *If* $pc, pc_{id}, v \vdash EH \rightsquigarrow ks$ *with* $pc \downarrow^p \not\sqsubseteq l$*, then* $ks \approx_l^p \cdot$

Proof (sketch): By induction on the structure of $\mathcal{F} :: pc, pc_{id}, v \vdash EH \rightsquigarrow ks$ $\square$

Lemma 91 (Weak One-Step). *If* $T_1 = \mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_1 \stackrel{\alpha_{l,1}}{\Longrightarrow} K_1'$ *and* $T_2 = \mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_2 \stackrel{\alpha_{l,2}}{\Longrightarrow} K_2'$*, with* $T_1 \approx_l^p T_2$*,* $K_1 \approx_l^p K_2$*,* $T_1 \downarrow_l^p \neq \cdot$*, and* $T_2 \downarrow_l^p \neq \cdot$*, then* $K_1' \approx_l^p K_2'$

Proof.

We examine each case of $\mathcal{F} :: \mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_1 \stackrel{\alpha_{l,1}}{\Longrightarrow} K_1'$

Denote $\mathcal{G} :: \mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_2 \stackrel{\alpha_{l,2}}{\Longrightarrow} K_2'$

We refer to the following assumptions throughout:

(1) $K_1 \approx_l^p K_2$; (2) $T_1 \approx_l^p T_2$; (3) $T_1 \downarrow_l^p \neq \cdot$; and (4) $T_2 \downarrow_l^p \neq \cdot$

Case I: $\mathcal{F}$ ends in In

By assumption and from $\Sigma_1(pc_1) = (_, \sigma_1^{EH})$, all of the following:

$\mathcal{R}_1 = \mathcal{R}_1'$; $\mathcal{S}_1 = \mathcal{S}_1'$; $\Sigma_1 = \Sigma_1'$; $\alpha_{l,1} = (id.\mathsf{Ev}(v), pc_1)$; $\mathcal{P}(id.\mathsf{Ev}(v)) = pc_1'$;

$E_1 = ((id.\mathsf{Ev}(v), pc_1'') \mid pc_1 \sqcup pc_1' \sqsubseteq pc_1'')$; $\sigma_1^{EH}(id) \downarrow^i \not\sqsubseteq pc_1 \downarrow^i$; $\sigma_1^{EH}(id) \downarrow^c \not\sqsubseteq pc_1 \downarrow^c$; and $\Sigma_1, E_1 \rightsquigarrow ks_1'$

From (3), $\alpha_{l,1} = (id.\mathsf{Ev}(v), pc_1)$, $\mathcal{P}(id.\mathsf{Ev}(v)) = pc_1'$, $\sigma_1^{EH}(id) \downarrow^i \not\sqsubseteq pc_1 \downarrow^i$, $\sigma_1^{EH}(id) \downarrow^c \not\sqsubseteq pc_1 \downarrow^c$, and the definition of $T \downarrow_l^p$, we know that $T_1 \downarrow_l^p = (id.\mathsf{Ev}(v), pc_1)$

Then from (3), $pc_1 \downarrow^p \sqcup pc_1' \downarrow^p \sqsubseteq l$

From $T_1 \downarrow_l^p = (id.\mathsf{Ev}(v), pc_1)$ and (2), $T_2 \downarrow_l^p = (id.\mathsf{Ev}(v), pc_1)$

Then from (4), $\alpha_{l,2} = (id.\mathsf{Ev}(v), pc_2)$ and $\mathcal{P}(id.\mathsf{Ev}(v)) = pc_2'$

From $\mathcal{P}(id.\mathsf{Ev}(v)) = pc_1'$ and $\mathcal{P}(id.\mathsf{Ev}(v)) = pc_2'$, we know that $pc_1' = pc_2'$

And from $T_2 \downarrow_l^p = (id.\mathsf{Ev}(v), pc_1)$ and $\alpha_{l,2} = (id.\mathsf{Ev}(v), pc_2)$, we know that $pc_1 = pc_2$

Subcase i: $\mathcal{G}$ ends in In

By assumption, all of the following:

$\mathcal{R}_2 = \mathcal{R}_2'$; $\mathcal{S}_2 = \mathcal{S}_2'$; $\Sigma_2 = \Sigma_2'$; $E_2 = ((id.\mathsf{Ev}(v), pc_2'') \mid pc_2 \sqcup pc_2' \sqsubseteq pc_2'')$; and $\Sigma_2, E_2 \rightsquigarrow ks_2'$

From $pc_1' = pc_2'$, $pc_1 = pc_2$, $E_1 = (...)$, and $E_2 = (...)$, we know that $E_1 \approx_l^p E_2$

The from Lemma 95,

$ks_1 \approx_l^p ks_2$

From (1), $\mathcal{R}_1 = \mathcal{R}_1'$, and $\mathcal{R}_2 = \mathcal{R}_2'$, we know that $\mathcal{R}_1' = \mathcal{R}_2'$

And from (1) $\mathcal{S}_1 = \mathcal{S}_1'$, and $\mathcal{S}_2 = \mathcal{S}_2'$, we know that $\mathcal{S}_1' = \mathcal{S}_2'$

And from (1), $\Sigma_1 = \Sigma_1'$, and $\Sigma_2 = \Sigma_2'$, we know that $\Sigma_1' \downarrow^p = \Sigma_2' \downarrow^p$

From $\mathcal{R}_1' = \mathcal{R}_2'$, $\mathcal{S}_1' = \mathcal{S}_2'$, $\Sigma_1' \downarrow^p = \Sigma_2' \downarrow^p$, and $ks_1 \approx_l^p ks_2$, we know that $K_1' \approx_l^p K_2'$

Subcase ii: $\mathcal{G}$ ends in In-D, In-E or In-DE

These cases hold vacuously based on (1) and $pc_1 = pc_2$

Case II: $\mathcal{F}$ ends in In-D, In-E, or In-DE

The proof is similar to **Case I**. We split the proof into two cases: one where the step produces a

declassification, and one where it doesn't. It also uses Lemma 96.

Case III: $\mathcal{F}$ ends in OUT

By assumption, all of the following:

$\alpha_{l,1} = (ch(v), pc_1)$; $ks_1 = (\kappa_1, pc_{src,1}, pc_1) :: ks_1''$; $\mathcal{P}(ch) = pc_1$; $\exists \mathcal{F}' :: pc_{src,1}, d_{d,1}, d_{e,1} \vdash \Sigma_1, \kappa_1 \xrightarrow{ch(v)} \Sigma_1', \kappa_1'''$; $ks_1' = ks_1''' :: ks_1''$; $\mathcal{R}_1' = \mathcal{R}_1$; and $\mathcal{S}_1' = \mathcal{S}_1$

From (3), $\alpha_{l,1} = (ch(v), pc_1)$, and $\mathcal{P}(ch) = pc_1$, $T_1 \downarrow_l^p = ch(v)$ with $pc_1 \downarrow^p \sqsubseteq l \vee \mathcal{P}(ch) \downarrow^p \sqsubseteq l$

Then, from $\mathcal{P}(ch) = pc_1$, $pc_1 \downarrow^p \sqsubseteq l$

From (2) and $T_1 \downarrow_l^p = ch(v)$, we know that $T_2 \downarrow_l^p = ch(v)$

Then, from the definition of $T \downarrow_l^p$, $\alpha_{l,2} = (ch(v), pc_2)$ with $pc_2 \downarrow^p \sqsubseteq l \vee \mathcal{P}(ch) \downarrow^p \sqsubseteq l$

Then, from our output rules, $\mathcal{G}$ must end with OUT with all of the following:

$\mathcal{P}(ch) = pc_2$; $ks_2 = (\kappa_2, pc_{src,2}, pc_2) :: ks_2''$; $\exists \mathcal{G}' :: pc_{src,2}, d_{d,2}, d_{e,2}, \vdash \Sigma_2, \kappa_2, \xrightarrow{ch(v)} \Sigma_2', \kappa_2'''$; $ks_2' = ks_2''' :: ks_2''$; $\mathcal{R}_2' = \mathcal{R}_2$; and $\mathcal{S}_2' = \mathcal{S}_2$

From $\mathcal{P}(ch) = pc_1$ and $\mathcal{P}(ch) = pc_2$, we know that $pc_1 = pc_2$

From (1), $pc_1 \downarrow^p \sqsubseteq l$, $pc_1 = pc_2$, $ks_1 = (\kappa_1, pc_{src,1}, pc_1) :: ks_1''$, and $ks_2' = ks_2''' :: ks_2''$, we know that $(\kappa_1, pc_{src,1}, pc_1) = (\kappa_2, pc_{src,2}, pc_2)$ with $ks_1'' \approx_l^p ks_2''$

Then from Lemma 92, $\Sigma_1' \approx_l^p \Sigma_2'$ and $ks_1''' \approx_l^p ks_2'''$

From $ks_1' = ks_1''' :: ks_1''$, $ks_2' = ks_2''' :: ks_2''$, $ks_1'' \approx_l^p ks_2''$, and $ks_1''' \approx_l^p ks_2'''$, we know that $ks_1' \approx_l^p ks_2'$

From (1), $\mathcal{R}_1' = \mathcal{R}_1$, and $\mathcal{R}_2' = \mathcal{R}_2$, we know that $\mathcal{R}_1' = \mathcal{R}_2'$

Similarly, from (1), $\mathcal{S}_1' = \mathcal{S}_1$, and $\mathcal{S}_2' = \mathcal{S}_2$, we know that $\mathcal{S}_1' = \mathcal{S}_2'$

Thus, from $\mathcal{R}_1' = \mathcal{R}_2'$, $\mathcal{S}_1' = \mathcal{S}_2'$, $\Sigma_1' \approx_l^p \Sigma_2'$, and $ks_1' \approx_l^p ks_2'$, we know that $K_1' \approx_l^p K_2'$

Case IV: $\mathcal{F}$ ends in OUT-SKIP or OUT-SILENT

The proof for this case is similar to **Case III**

Case V: $\mathcal{F}$ ends in OUT-SILENT and $T_1 \downarrow_l^p = \in \{\mathsf{t}(_), \mathsf{r}(_)\}$

Without loss of generality, assume $T_1 \downarrow_l^p = \mathsf{r}(id, pc_1)$. The proof for the other cases are similar. The most important difference is that when $T_1 \downarrow_l^p = \mathsf{r}(id, eh, pc_1)$, then we also have $pc_{id,1} \downarrow^i \sqsubseteq pc_1 \downarrow^i$ and $pc_{id,2} \downarrow^i \sqsubseteq pc_1 \downarrow^i$

In general:

$pc_{src,1} \downarrow^p \sqsubseteq pc_1 \downarrow^p$

$pc_{src,2} \downarrow^p \sqsubseteq pc_2 \downarrow^p$ and

$pc_{id,1} \downarrow^p \sqsubseteq pc_1 \downarrow^p$

$pc_{id,2} \downarrow^p \sqsubseteq pc_2 \downarrow^p$

(which is the premise for Lemma 94)

By assumption, all of the following:

$\alpha_{l,1} = (\mathsf{new}(id, pc_{src,1}), pc_1)$; $ks_1 = (\kappa_1, pc_{src,1}, pc_1) :: ks_1''$; $\mathsf{producer}(\kappa_1)$;

$\exists \mathcal{F}' :: pc_{src,1}, d_{d,1}, d_{e,1} \vdash \Sigma_1, \kappa_1 \xrightarrow{\mathsf{new}(id, pc_{src,1})} \Sigma_1', \kappa_1'''$; $ks_1' = ks_1''' :: ks_1''$; $\mathcal{R}_1' = \mathcal{R}_1$; and $\mathcal{S}_1' = \mathcal{S}_1$

By assumption and from (3) and $\alpha_{l,1} = (\mathsf{new}(id, pc_{src,1}), pc_1)$, we know that $pc_1 \downarrow^c \not\sqsubseteq l$ and $pc_{src,1} \downarrow^i \sqsubseteq pc_1 \downarrow^i$

By assumption and from (2), we know that $T_2 \downarrow^c = \mathsf{r}(id, pc_2)$ and $pc_1 = pc_2$

Then, we know that $\alpha_{l,2} = (\mathsf{new}(id, pc_{src,2}), pc_2)$ with $pc_2 \downarrow^c \not\sqsubseteq l$ and $pc_{src,2} \downarrow^i \sqsubseteq pc_2 \downarrow^i$

Since NEW is the only rule to produce $\alpha = \mathsf{new}(_)$, we know $\kappa_1 = (_, \mathsf{new}(id, e_1), _, _)$ and $ks_2 = (\kappa_2, pc_{src,2}, pc_2) :: ks_2''$ with $\kappa_2 = (_, \mathsf{new}(id, e_2), _, _)$ and $ks_1'' \approx_l^p ks_2''$

From $\kappa_2 = (_, \mathsf{new}(id, e_2), _, _)$, we know that $\mathsf{producer}(\kappa_2)$

Therefore, $\mathcal{G}$ must end in OUT-SILENT, with all of the following:

$\exists \mathcal{G}' :: pc_{src,2}, d_{d,2}, d_{e,2}, \vdash \Sigma_2, \kappa_2, \xrightarrow{\mathsf{new}(id, pc_{src,2})} \Sigma_2', \kappa_2'''$; $ks_2' = ks_2''' :: ks_2''$; $\mathcal{R}_2' = \mathcal{R}_2$; and $\mathcal{S}_2' = \mathcal{S}_2$

From Lemma 94, $\Sigma_1' \approx_l^p \Sigma_2'$ and $ks_1''' \approx_l^p ks_2'''$

From $ks_1' = ks_1''' :: ks_1''$, $ks_2' = ks_2''' :: ks_2''$, $ks_1'' \approx_l^p ks_2''$, and $ks_1''' \approx_l^p ks_2'''$, we know that $ks_1' \approx_l^p ks_2'$

From (1), $\mathcal{R}_1' = \mathcal{R}_1$, and $\mathcal{R}_2' = \mathcal{R}_2$, we know that $\mathcal{R}_1' = \mathcal{R}_2'$

And from (1), $\mathcal{S}_1' = \mathcal{S}_1$, and $\mathcal{S}_2' = \mathcal{S}_2$, we know that $\mathcal{S}_1' = \mathcal{S}_2'$

Thus, from $\mathcal{R}_1' = \mathcal{R}_2'$, $\mathcal{S}_1' = \mathcal{S}_2'$, $\Sigma_1' \approx_l^p \Sigma_2'$, and $ks_1' \approx_l^p ks_2'$, we know that $K_1' \approx_l^p K_2'$

Case VI: $\mathcal{F}$ ends in OUT-NEXT

The proof for this case is straightforward. It uses the assumptions at the beginning to establish that $pc_1 \downarrow^p \sqsubseteq l$, and $pc_2 \downarrow^p \sqsubseteq l$, which helps show $K_1' \approx_l^p K_2'$

□

Lemma 92. *If $pc_{src}, d_d, d_e \vdash \Sigma_1, \kappa \xrightarrow{\alpha_1}_{pc} \Sigma_1', ks_1$ and $pc_{src}, d_d, d_e \vdash \Sigma_2, \kappa \xrightarrow{\alpha_2}_{pc} \Sigma_2', ks_2$, with $\Sigma_1 \approx_l^p \Sigma_2$ and $pc \downarrow^p \sqsubseteq l$, then $\Sigma_1' \approx_l^p \Sigma_2'$ and $ks_1 \approx_l^p ks_2$*

Proof (sketch): We examine each case of $\mathcal{F} :: pc_{src}, d_d, d_e \vdash \Sigma_1, \kappa \xrightarrow{\alpha_1}_{pc} \Sigma_1', ks_1$ The proof uses Lemmas 95 and 93. □

Lemma 93. *If $pc_{src}, d_d, d_e \vdash \Sigma_1, \sigma, c \xrightarrow{\alpha_1}_{pc} \Sigma_1', \sigma_1, c_1, E_1$ and $pc_{src}, d_d, d_e \vdash \Sigma_2, \sigma, c \xrightarrow{\alpha_2}_{pc} \Sigma_2', \sigma_2, c_2, E_2$, with $\Sigma_1 \approx_l^p \Sigma_2$ and $pc \downarrow^p \sqsubseteq l$, then $\Sigma_1' \approx_l^p \Sigma_2'$, $\sigma_1 = \sigma_2$, $c_1 = c_2$, and $E_1 = E_2$*

Proof (sketch): By induction on the structure of $\mathcal{F} :: pc_{src}, d_d, d_e \vdash \Sigma_1, \sigma, c \xrightarrow{\alpha_1}_{pc} \Sigma_1', \sigma_1, c_1, E_1$ and $\mathcal{G} :: pc_{src}, d_d, d_e \vdash \Sigma_2, \sigma, c \xrightarrow{\alpha_2}_{pc} \Sigma_2', \sigma_2, c_2, E_2$ □

Lemma 94. *If $pc_{src,1}, d_{d,1}, d_{e,1} \vdash \Sigma_1, \kappa_1 \xrightarrow{\alpha_1}_{pc} \Sigma_1', ks_1'$ and $pc_{src,2}, d_{d,2}, d_{e,2} \vdash \Sigma_2, \kappa_2 \xrightarrow{\alpha_2}_{pc} \Sigma_2', ks_2'$, with $\alpha_1 = \mathsf{new}(id, pc_{src,1})$ and $\alpha_2 = \mathsf{new}(id, pc_{src,2})$, or $\alpha_1 = \mathsf{newEH}(id, eh, pc_{id,1}, pc_{src,1})$ and $\alpha_2 = \mathsf{newEH}(id, eh, pc_{id,2}, pc_{src,2})$*

$\Sigma_1 \approx_l^p \Sigma_2$, $pc \downarrow^p \not\sqsubseteq l$, and $pc_{src,1} \downarrow^p \sqsubseteq pc_1 \downarrow^p$ $pc_{src,2} \downarrow^p \sqsubseteq pc_2 \downarrow^p$ and $pc_{id,1} \downarrow^p \sqsubseteq pc_1 \downarrow^p$ $pc_{id,2} \downarrow^p \sqsubseteq pc_2 \downarrow^p$, then $\Sigma_1' \approx_l^p \Sigma_2'$ and $ks_1' \approx_l^p ks_2'$

Proof.

We examine each case of $\mathcal{F} :: pc_{src,1}, d_{d,1}, d_{e,1} \vdash \Sigma_1, \kappa_1 \xrightarrow{\alpha_1}_{pc} \Sigma_1', ks_1'$

Denote $\mathcal{G} :: pc_{src,2}, d_{d,2}, d_{e,2} \vdash \Sigma_2, \kappa_2 \xrightarrow{\alpha_2}_{pc} \Sigma_2', ks_2'$

We refer to the following assumptions throughout:

(1) $\Sigma_1 \approx_l^p \Sigma_2$;

(2) $pc \downarrow^p \not\sqsubseteq l$;

(3) $pc_{src,1} \downarrow^p \sqsubseteq pc_1 \downarrow^p$ and $pc_{src,2} \downarrow^p \sqsubseteq pc_2 \downarrow^p$;

(4) $pc_{id,1} \downarrow^p \sqsubseteq pc_1 \downarrow^p$ and $pc_{id,2} \downarrow^p \sqsubseteq pc_2 \downarrow^p$; and either

(5) $\alpha_1 = \mathsf{new}(id, pc_{src,1})$ and $\alpha_2 = \mathsf{new}(id, pc_{src,2})$ or

(6) $\alpha_1 = \mathsf{newEH}(id, eh, pc_{id,1}, pc_{src,1})$ and $\alpha_2 = \mathsf{newEH}(id, eh, pc_{id,2}, pc_{src,2})$

From (5) and (6) and since only P could produce $\mathsf{new}(_)$ or $\mathsf{newEH}(_)$, we know that $\mathcal{F}$ and $\mathcal{G}$ must end in P; and we know the following:

(7) $\exists \mathcal{F}' :: pc_{src,1}, d_{d,1}, d_{e,1} \vdash \Sigma_1, \sigma_1, c_1 \xrightarrow{\alpha_1}_{pc} \Sigma_1', \sigma_1', c_1', E_1$; (8) $c_1 \in \{\mathsf{new}(id, e_1), \mathsf{addEH}(id, eh)\}$; $ks_1' = ((\sigma_1', c_1', P, E :: E_1), pc_{src,1}, pc)$; (9) $\exists \mathcal{G}' :: pc_{src,2}, d_{d,2}, d_{e,2} \vdash \Sigma_2, \sigma_2, c_2 \xrightarrow{\alpha_2}_{pc} \Sigma_2', \sigma_2', c_2', E_2$;

(10) $c_2 \in \{\mathsf{new}(id, e_2), \mathsf{addEH}(id, eh)\}$; $ks_2' = ((\sigma_2', c_2, P, E :: E_2), pc_{src,2}, pc)$

From (2), $ks_1' = ((\sigma_1', c_1', P, E :: E_1), pc_{src,1}, pc)$, and $ks_2' = ((\sigma_2', c_2, P, E :: E_2), pc_{src,2}, pc)$, we know that $ks_1' \approx_l^p ks_2'$

From (7)-(10), $\mathcal{F}'$ and $\mathcal{G}'$ end in NEW or ADD-EH

Case I: $\mathcal{F}'$ and $\mathcal{G}'$ end in NEW

By assumption and from $\Sigma_1(pc) = (_, \sigma_1^{EH})$ and $\Sigma_2(pc) = (_, \sigma_2^{EH})$, we know the following:

(I.1) $\sigma_1^{EH'} = \sigma_1^{EH}[id \mapsto (_, \cdot, pc_{src,1})]$; (I.2) $\sigma_2^{EH'} = \sigma_2^{EH}[id \mapsto (_, \cdot, pc_{src,2})]$; $\Sigma_1' = \Sigma_1[pc \mapsto (_, \sigma_1^{EH'})]$; and $\Sigma_2' = \Sigma_2[pc \mapsto (_, \sigma_2^{EH'})]$

From (1) and (2), $\Sigma_1 \approx_l^p \Sigma_2$ and $\sigma_1^{EH} \approx_l^p \sigma_2^{EH}$

Then from (I.1), (I.2), and (3), we know that $\sigma_1^{EH'} \approx_l^p \sigma_2^{EH'}$

Thus, from (2), $\Sigma_1 \approx_l^p \Sigma_2$ $\Sigma_1' = \Sigma_1[pc \mapsto (_, \sigma_1^{EH'})]$, $\Sigma_2' = \Sigma_2[pc \mapsto (_, \sigma_2^{EH'})]$, and $\sigma_1^{EH'} \approx_l^p \sigma_2^{EH'}$ we know that $\Sigma_1' \approx_l^p \Sigma_2'$

Case II: $\mathcal{F}'$ and $\mathcal{G}'$ end in ADD-EH

By assumption and from $\Sigma_1(pc) = (_, \sigma_1^{EH})$, $\Sigma_2(pc) = (_, \sigma_2^{EH})$, and $eh = \mathsf{onEv}(x)\{c\}$ we know the following:

(II.1) $\sigma_1^{EH}(id) = (_, M_1, pc_{id,1})$; (II.2) $M_1(\mathsf{Ev}) = EH_1$; (II.3) $M_1' = M_1[\mathsf{Ev} \mapsto EH_1 \cup \{eh, pc_{src,1}\}]$;

(II.4) $\sigma_1^{EH'} = \sigma_1^{EH}[id \mapsto (_, M_1', pc_{id,1})]$; (II.5) $\Sigma_1' = \Sigma_1[pc \mapsto (_, \sigma_1^{EH'})]$; (II.6) $\sigma_2^{EH}(id) = (_, M_2, pc_{id,2})$; (II.7) $M_2(\mathsf{Ev}) = EH_2$; (II.8) $M_2' = M_2[\mathsf{Ev} \mapsto EH_2 \cup \{eh, pc_{src,2}\}]$; (II.9) $\sigma_2^{EH'} = \sigma_2^{EH}[id \mapsto (_, M_2', pc_{id,2})]$; and (II.10) $\Sigma_2' = \Sigma_2[pc \mapsto (_, \sigma_2^{EH'})]$

From (1) and (2), $\Sigma_1 \approx_l^p \Sigma_2$ and $\sigma_1^{EH} \approx_l^p \sigma_2^{EH}$

From (4), (II.1), (II.2), (II.6), (II.7), and $\sigma_1^{EH} \approx_l^p \sigma_2^{EH}$, we know that $M_1 \downarrow_l^p = M_2 \downarrow_l^p$ and $EH_1 \downarrow_l^p = EH_2 \downarrow_l^p$

Then from (3), (II.3), and (III.8), we know that $M_1' \downarrow_l^p = M_2' \downarrow_l^p$

From (4), (II.4), (II.9), and $M_1' \downarrow_l^p = M_2' \downarrow_l^p$, we know that $\sigma_1^{EH'} \approx_l^p \sigma_2^{EH'}$

Thus, from (2), (II.5), (II.10), $\Sigma_1 \approx_l^p \Sigma_2$, and $\sigma_1^{EH'} \approx_l^p \sigma_2^{EH'}$, we know that $\Sigma_1' \approx_l^p \Sigma_2'$

□

Lemma 95 (Equivalent State, Equivalent Event Handlers). *If $\Sigma_1 \approx_l^p \Sigma_2$ and $E_1 \approx_l^p E_2$, with $\Sigma_1, E_1 \rightsquigarrow ks_1$, $\Sigma_2, E_2 \rightsquigarrow ks_2$ then $ks_1 \approx_l^p ks_2$*

Proof (sketch): By induction on $\mathcal{F} :: \Sigma_1, E_1 \rightsquigarrow ks_1$ and $\mathcal{G} :: \Sigma_2, E_2 \rightsquigarrow ks_2$ The proof uses Lemmas 90 and 89.

□

Lemma 96. *If $\Sigma_1 \approx_l^p \Sigma_2$ and $E_1 \approx_l^p E_2$, with $pc_{\mathsf{Ev}}, \mathsf{f} \vdash \Sigma_1, E_1 \rightsquigarrow ks_1$, $pc_{\mathsf{Ev}}, \mathsf{f} \vdash \Sigma_2, E_2 \rightsquigarrow ks_2$ and $\mathsf{f} \in \{\mathsf{r}, \mathsf{t}, \mathsf{rt}\}$ then $ks_1 \approx_l^p ks_2$*

Proof (sketch): By induction on $\mathcal{F} :: pc_{\mathsf{Ev}}, \mathsf{f} \vdash \Sigma_1, E_1 \rightsquigarrow ks_1$ and $\mathcal{G} :: pc_{\mathsf{Ev}}, \mathsf{f} \vdash \Sigma_2, E_2 \rightsquigarrow ks_2$ for $\mathsf{f} \in \{\mathsf{r}, \mathsf{t}, \mathsf{rt}\}$. The proof also

uses Lemma 90. □

Lemma 97. *If $pc, \mathsf{f} \vdash \Sigma, E \rightsquigarrow ks$ with $E \downarrow_l^p = \cdot$ and $\mathsf{f} \in \{\mathsf{r}, \mathsf{t}, \mathsf{rt}\}$, then $ks \approx_l^p \cdot$*

Proof (sketch): By induction on the structure of $\mathcal{F} :: pc, \mathsf{f} \vdash \Sigma, E \rightsquigarrow ks$ for $\mathsf{f} \in \{\mathsf{r}, \mathsf{t}, \mathsf{rt}\}$ The proof also uses Lemma 90. □

Lemma 98 (Strong One-step). *If $K_1 \approx_l^p K_2$, $T_1 = \mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_1 \stackrel{\alpha_{l,1}}{\Longrightarrow} K_1'$ with $T_1 \downarrow_l^p = \tau \neq \cdot$ and $\mathsf{prog}(K_2)$, with $\neg\mathsf{rlsA}(T_1), \mathsf{trnsprntA}(T_1, l), \mathsf{transparentT}(K_2, \tau, l)$ if $p = c$, $\neg\mathsf{sntzA}(T_1), \mathsf{rbstA}(T_1, l), \mathsf{robustT}(K_2, \tau, l)$ if $p = i$, and $T_1 \downarrow_l^p \notin \{\mathsf{t}(_), \mathsf{r}(_)\}$, then $\exists K_2', T_2$ s.t. $T_2 = \mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_2 \Longrightarrow^* K_2'$ with $T_1 \approx_l^p T_2$ and $K_1' \approx_l^p K_2'$*

Proof.

We examine each case of $\mathcal{F} :: T_1 = \mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_1 \stackrel{\alpha_{l,1}}{\Longrightarrow} K_1'$

We refer to the following assumptions throughout:

(1) $T_1 \downarrow_l^p = \tau \neq \cdot$; (2) $K_1 \approx_l^p K_2$; (3) $\mathsf{prog}(K_2)$; (4) $\neg\mathsf{rlsA}(T_1), \mathsf{trnsprntA}(T_1, l), \mathsf{transparentT}(K_2, \tau, l)$ if $p = c$; (5) $\neg\mathsf{sntzA}(T_1), \mathsf{rbstA}(T_1, l), \mathsf{robustT}(K_2, \tau, l)$ if $p = i$; and (6) $T_1 \downarrow_l^p \notin \{\mathsf{t}(_), \mathsf{r}(_)\}$

Case I: $\mathcal{F}$ ends in In

The proof for this case is straightforward. It uses $\mathsf{prog}(K_2)$ and Lemma 86 to step to an equivalent consumer state. Lemma 95 is used to establish equivalence of the resulting event handlers.

Case II: $\mathcal{F}$ ends in IN-D

By assumption and from (4) and (5), $\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_2 \Longrightarrow^* K_C$ with $\mathsf{consumer}(K_C)$, and $(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_2 \Longrightarrow^* K_C) \downarrow_l^p = \cdot$

From $\mathsf{consumer}(K_C)$, we know that $ks_C = \cdot$

From Lemma 86, $K_2 \approx_l^p K_C$

From (2) and $K_2 \approx_l^p K_C$, we know that $K_1 \approx_l^p K_C$

By assumption, all of the following:

$\mathcal{S}_1' = \mathcal{S}_1$; $\Sigma_1' = \Sigma_1$; $\alpha_1 = (id.\mathsf{Ev}(v), pc)$; $\mathcal{P}(id.\mathsf{Ev}(v)) = pc'$; $\Sigma_1(pc) = (_, \sigma_1^{EH})$; $\sigma_1^{EH}(id) \downarrow^i \sqsubseteq pc \downarrow^i$; $\sigma_1^{EH}(id) \downarrow^c \not\sqsubseteq pc \downarrow^c$; $E_1 = ((id.\mathsf{Ev}(v), pc'') \mid (pc \sqcup pc' \sqsubseteq pc''))$;
$\mathsf{downgrade}_{\mathcal{D}}(\mathcal{R}_1, \Sigma_1, (id.\mathsf{Ev}(v), pc), pc') = (\mathcal{R}_1', E_1')$; $\Sigma_1, E_1 \rightsquigarrow ks_1''$; $pc, \mathsf{r} \vdash \Sigma_1, E_1' \rightsquigarrow ks_1'''$; and $ks_1' = ks_1'' :: ks_1'''$

From the definition of $\mathsf{downgrade}_{\mathcal{D}}$, all of the following:

$\mathcal{R}_1 = (\rho_1, d_1)$; $\mathcal{D}(id.\mathsf{Ev}(v), pc, \rho_1) = (\rho_1', v_1, E_{d,1}')$; $d_1' = \mathsf{update}(d_1, v_1)$; $\mathcal{R}_1' = (\rho_1', d_1')$;
$E_{d,1} = ((id.\mathsf{Ev}(v), (l_c, l_i)) \mid pc' \downarrow^c \sqsubseteq l_c \sqsubset pc \downarrow^c \wedge l_i = pc \downarrow^i \sqcup pc' \downarrow^i)$; and $E_1' = \mathsf{robust}(\Sigma_1, E_{d,1} :: E_{d,1}', pc)$

From (1), $\mathcal{P}(id.\mathsf{Ev}(v)) = pc'$, $\sigma_1^{EH}(id) \downarrow^i \sqsubseteq pc \downarrow^i$, and $\sigma_1^{EH}(id) \downarrow^c \not\sqsubseteq pc \downarrow^c$, we know that $T_1 \downarrow_l^p = (id.\mathsf{Ev}(v), pc)$ or $T_1 \downarrow_l^p = \mathsf{rls}(id.\mathsf{Ev}(v), \rho_1', v_1, E_1'', pc)$

Subcase i: $T_1 \downarrow_l^p = (id.\mathsf{Ev}(v), pc)$

By assumption and from $pc, \mathsf{r} \vdash \Sigma_1, E_1' \rightsquigarrow ks_1'''$ and $E_1' = \mathsf{robust}(\Sigma_1, E_{d,1} :: E_{d,1}', pc)$, we know that $pc \downarrow_l^p \sqsubseteq l$; $\mathcal{D}(id.\mathsf{Ev}(v), pc', \rho_1) = (\rho_1, \mathsf{none}, E_{d,1}')$; and $ks_1''' \downarrow_l^p = \cdot$ if $p = c$ and $ks_1''' = \cdot$ if $p = i$

From $\mathcal{D}(id.\mathsf{Ev}(v), pc, \rho_1) = (\rho_1', v_1, E_{d,1}') = (\rho_1, \mathsf{none}, E_{d,1}')$ and $d_1' = \mathsf{update}(d_1, v_1)$, we know that $d_1' = d_1$

Similarly, from $\mathcal{D}(id.\mathsf{Ev}(v), pc, \rho_1) = (\rho_1', v_1, E_{d,1}') = (\rho_1, \mathsf{none}, E_{d,1}')$, $d_1' = d_1$, and $\mathcal{R}_1' = (\rho_1', d_1')$, we know that $\mathcal{R}_1' = \mathcal{R}_1$

From $K_1 \approx_l^p K_C$, $pc \downarrow_l^p \sqsubseteq l$, $\sigma_1^{EH}(id) \downarrow^i \sqsubseteq pc \downarrow^i$, and $\sigma_1^{EH}(id) \downarrow^c \not\sqsubseteq pc \downarrow^c$, we know that $\sigma_C^{EH}(id) \downarrow^i \sqsubseteq pc \downarrow^i$ and $\sigma_C^{EH}(id) \downarrow^c \not\sqsubseteq pc \downarrow^c$

Then, IN-D may be applied to $\mathcal{R}_C, \mathcal{S}_C; \Sigma_C; ks_C$ with input $(id.\mathsf{Ev}(v), pc)$, meaning:

$\exists K_2'$ s.t. $\mathcal{G} :: T_2 = \mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_2 \Longrightarrow^* K_C \stackrel{(id.\mathsf{Ev}(v), pc)}{\Longrightarrow} K_2'$ with
$\mathcal{S}_2' = \mathcal{S}_C$; $\Sigma_2' = \Sigma_C$; $\mathcal{P}(id.\mathsf{Ev}(v)) = pc_2'$; $E_2 = ((id.\mathsf{Ev}(v), pc'') \mid (pc \sqcup pc' \sqsubseteq pc''))$;
$\mathsf{downgrade}_{\mathcal{D}}(\mathcal{R}_2, \Sigma_2, (id.\mathsf{Ev}(v), pc), pc') = (\mathcal{R}_2', E_{d,2})$; $\Sigma_C, E_2 \rightsquigarrow ks_2''$; $pc, \mathsf{r} \vdash \Sigma_2, E_2' \rightsquigarrow ks_2'''$; and $ks_2' = ks_2'' :: ks_2'''$

From the definition of $\mathsf{downgrade}_{\mathcal{D}}$,

$\mathcal{R}_C = (\rho_C, d_C)$; $\mathcal{D}(id.\mathsf{Ev}(v), pc, \rho_C) = (\rho_2', v_2, E_{d,2}')$; $d_2' = \mathsf{update}(d_C, v_2)$; $\mathcal{R}_2' = (\rho_2', d_2')$;

$E_{d,2} = ((id.\mathsf{Ev}(v), (l_c, l_i)) \mid pc' \downarrow^c \sqsubseteq l_c \sqsubset pc \downarrow^c \wedge l_i = pc \downarrow^i \sqcup pc' \downarrow^i)$; and

$E_2' = \mathsf{robust}(\Sigma_C, E_{d,2} :: E_{d,2}', pc)$

From $K_1 \approx_l^p K_C$, $\mathcal{D}(id.\mathsf{Ev}(v), pc, \rho_C) = (\rho_2', v_2, E_{d,2}') = (\rho_1, \mathsf{none}, E_{d,1}')$ and $d_2' = \mathsf{update}(d_C, v_2)$, we know that $d_2' = d_C$ and $E_{d,1}' = E_{d,2}'$

From $E_{d,1} = (...)$ and $E_{d,2} = (...)$, we know that $E_{d,1} = E_{d,2}$

Then, from Lemma 102, $E_1' \approx_l^p E_2'$ if $p = c$ and $E_1' = E_2'$ if $p = i$

And from Lemma 96, $ks_1''' \approx_l^p ks_2'''$ if $p = c$

From Lemma 104, $ks_1''' = ks_2'''$ if $p = i$

Then, from $ks_1''' \downarrow_l^p = \cdot$ if $p = c$ and $ks_1''' = \cdot$ if $p = i$, we know that $ks_2''' \approx_l^p \cdot$ if $p = c$ and $ks_2''' = \cdot$ if $p = i$

Then, from $\mathcal{D}(id.\mathsf{Ev}(v), pc, \rho_C) = (\rho_C, \mathsf{none}, E_{d,1}')$, $d_2' = d_C$, $pc \downarrow_l^p \sqsubseteq l$, and the definition of $T \downarrow_l^p$, we know that $T_2 \downarrow_l^p = (id.\mathsf{Ev}(v), pc)$

Then, from $T_1 \downarrow_l^p = (id.\mathsf{Ev}(v), pc)$, we know that $T_1 \approx_l^p T_2$

From $\mathcal{D}(id.\mathsf{Ev}(v), pc, \rho_C) = (\rho_C, \mathsf{none}, E_{d,1}')$ and $d_2' = d_C$, we know that $\mathcal{R}_2' = \mathcal{R}_C$

From $K_1 \approx_l^p K_C$, $\mathcal{R}_1' = \mathcal{R}_1$, and $\mathcal{R}_2' = \mathcal{R}_C$, we know that $\mathcal{R}_1' = \mathcal{R}_2'$

Similarly, from $K_1 \approx_l^p K_C$, $\mathcal{S}_1' = \mathcal{S}_1$, and $\mathcal{S}_2' = \mathcal{S}_C$, we know that $\mathcal{S}_1' = \mathcal{S}_2'$

And from $K_1 \approx_l^p K_C$, $\Sigma_1' = \Sigma_1$, and $\Sigma_2' = \Sigma_C$, we know that $\Sigma_1' = \Sigma_2'$

From $E_1 = (...)$ and $E_2 = (...)$, we know that $E_1 = E_2$

From Lemma 95, $ks_1'' \approx_l^p ks_2''$

From $ks_1''' \approx_l^p ks_2'''$ if $p = c$ and $ks_1''' = ks_2'''$ if $p = i$, we know that $ks_1''' \approx_l^p ks_2'''$

From $ks_1' = ks_1'' :: ks_1'''$, $ks_2' = ks_2'' :: ks_2'''$, $ks_1'' \approx_l^p ks_2''$, and $ks_1''' \approx_l^p ks_2'''$, we know that $ks_1' \approx_l^p ks_2'$

Thus, from $\mathcal{R}_1' = \mathcal{R}_2'$, $\mathcal{S}_1' = \mathcal{S}_2'$, $\Sigma_1' = \Sigma_2'$, and $ks_1' \approx_l^p ks_2'$, we know that $K_1' \approx_l^p K_2'$

Subcase ii: $T_1 \downarrow_l^p = \mathsf{rls}(id.\mathsf{Ev}(v), \rho_1', v_1, E_1'', pc)$

By assumption and from (4), $p = i$ and $pc \downarrow^p \sqsubseteq l$

Then, from (2), we know that $\sigma_1^{EH} = \sigma_2^{EH}$

The rest of the proof for this case is similar to **Subcase i**

Case III: $\mathcal{F}$ ends in IN-E

The proof is similar to **Case II**. It uses Lemma 103 instead of Lemma 102 and Lemma 105 instead of Lemma 104.

Case IV: $\mathcal{F}$ ends in IN-DE

From (4), $T_1 \downarrow_l^p \neq \mathsf{down}(...)$, so the proof is similar to **Case II**.

Case V: $\mathcal{F}$ ends in OUT

By assumption and from (4) and (5), $\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_2 \stackrel{\tau'}{\Longrightarrow}^* K_l$ with $\mathsf{lowEH}(K_l)$ and

$\forall(\alpha, pc) \in \tau', \alpha \in \{ch(_), \bullet\} \wedge pc \downarrow^p \not\sqsubseteq l$

From $\mathsf{lowEH}(K_l)$, we know that $ks_l = (\kappa_l, pc_{src,l}, pc_l) :: ks'_l$ with $pc_l \downarrow^p \sqsubseteq l$

From Lemma 101, $(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_2 \stackrel{\tau}{\Longrightarrow}^* K_l) \downarrow^p_l = \cdot$

Then, from Lemma 86, $K_2 \approx^p_l K_l$

From (2) and $K_2 \approx^p_l K_l$, we know that $K_1 \approx^p_l K_l$

By assumption, all of the following:

$\mathcal{R}'_1 = \mathcal{R}_1$; $\mathcal{S}'_1 = \mathcal{S}_1$; $\alpha_{l,1} = (ch(v), pc_1)$; $\mathcal{P}(ch) = pc_1$; $\mathcal{R}_1 = (\rho_{d,1}, d_{d,1})$; $\mathcal{S}_1 = (\rho_{e,1}, d_{e,1})$;

$ks_1 = (\kappa_1, pc_{src,1}, pc_1) :: ks''_1$; $\mathsf{producer}(\kappa_1)$; $\exists \mathcal{F}' :: pc_{src,1}, d_{d,1}, d_{e,1} \vdash \Sigma_1, \kappa_1 \xrightarrow{ch(v)}_{pc_1} \Sigma'_1, ks'''_1$; $ks'_1 = ks'''_1 :: ks''_1$

From (1), $\alpha_{l,1} = (ch(v), pc_1)$, and the definition of $\downarrow^p_l$ for T, we know that $pc_1 \downarrow^p \sqsubseteq l$

From $K_1 \approx^p_l K_l$, $ks_1 = (\kappa_1, pc_{src,1}, pc_1) :: ks''_1$, $ks_l = (\kappa_l, pc_{src,l}, pc_l) :: ks'_l$, and the definition of $\approx^p_l$ for ks, we know that $pc_1 = pc_l$ and $(\kappa_1, pc_{src,1}, pc_1) = (\kappa_l, pc_{src,l}, pc_l)$

From $K_1 \approx^p_l K_l$, $\mathcal{R}_1 = (\rho_{d,1}, d_{d,1})$, and $\mathcal{S}_1 = (\rho_{e,1}, d_{e,1})$, we know that

$\mathcal{R}_l = (\rho_{d,l}, d_{d,l})$ and $\mathcal{S}_l = (\rho_{e,l}, d_{e,l})$ with

$d_{d,1} = d_{d,l}$ and $d_{e,1} = d_{e,l}$

From Lemma 99, $\exists \mathcal{G}' :: pc_{src,l}, d_{d,l}, d_{e,l} \vdash \Sigma_l, \kappa_l \xrightarrow{ch(v)}_{pc_l} \Sigma'_2, ks''_2$ with $\Sigma'_1 \approx^p_l \Sigma'_2$ and $ks'''_1 \approx^p_l ks''_2$

From $\mathsf{producer}(\kappa_1)$ and $(\kappa_1, pc_{src,1}, pc_1) = (\kappa_l, pc_{src,l}, pc_l)$, we know that $\mathsf{producer}(\kappa_2)$

Then, OUT may be applied to $\mathcal{R}_l, \mathcal{S}_l; \Sigma_l; ks_l$, producing output $(ch(v), pc_l)$, meaning:

$\mathcal{G} :: T_2 = \mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_2 \Longrightarrow^* K_l \stackrel{(ch(v), pc_l)}{\Longrightarrow} K'_2$ with $\mathcal{R}'_2 = \mathcal{R}_l$, $\mathcal{S}'_2 = \mathcal{S}_l$, and $ks'_2 = ks''_2 :: ks'_l$

From $pc_1 \downarrow^p_l \sqsubseteq l$, $\alpha_{l,1} = (ch(v), pc_1)$, and $\mathcal{P}(ch) = pc_1$, we know that $T_1 \downarrow^p_l = ch(v)$

Similarly, from $pc_l \downarrow^p \sqsubseteq l$, $pc_1 = pc_l$, and $(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_2 \stackrel{\tau}{\Longrightarrow}^* K_l) \downarrow^p_l = \cdot$, we know that $T_2 \downarrow^p_l = ch(v)$

Thus, $T_1 \approx^p_l T_2$

From $K_1 \approx^p_l K_l$, $ks_1 = (\kappa_1, pc_{src,1}, pc_1) :: ks''_1$, and $ks_l = (\kappa_l, pc_{src,l}, pc_l) :: ks'_l$, we know that $ks''_1 \approx^p_l ks'_l$

From $ks'_1 = ks'''_1 :: ks''_1$, $ks'_2 = ks''_2 :: ks'_l$, $ks'''_1 \approx^p_l ks''_2$, and $ks''_1 \approx^p_l ks'_l$, we know that $ks'_1 \approx^p_l ks'_2$

From $K_1 \approx^p_l K_l$, $\mathcal{R}'_1 = \mathcal{R}_1$, and $\mathcal{R}'_2 = \mathcal{R}_l$, we know that $\mathcal{R}'_1 = \mathcal{R}'_2$

Similarly, from $K_1 \approx^p_l K_l$, $\mathcal{S}'_1 = \mathcal{S}_1$, and $\mathcal{S}'_2 = \mathcal{S}_l$, we know that $\mathcal{S}'_1 = \mathcal{S}'_2$

Thus, from $\mathcal{R}'_1 = \mathcal{R}'_2$, $\mathcal{S}'_1 = \mathcal{S}'_2$, $\Sigma'_1 \approx^p_l \Sigma'_2$, and $ks'_1 \approx^p_l ks'_2$, we know that $K'_1 \approx^p_l K'_2$

Case VI: $\mathcal{F}$ ends in OUT-SKIP, OUT-SILENT, or OUT-NEXT

The proofs for these cases are similar to **Case V**

□

Lemma 99. *If* $\mathcal{F} :: pc_{src}, d_d, d_e \vdash \Sigma_1, \kappa \xrightarrow{\alpha}_{pc} \Sigma'_1, ks_1$ *with* $pc \downarrow^p \sqsubseteq l$ *and* $\Sigma_1 \approx^p_l \Sigma_2$*, then* $\exists \mathcal{G} :: pc_{src}, d_d, d_e \vdash$

$\Sigma_2, \kappa \xrightarrow{\alpha}_{pc} \Sigma_2', ks_2$ *with* $\Sigma_1' \approx_l^p \Sigma_2'$ *and* $ks_1 \approx_l^p ks_2$

Proof (sketch): We examine each case of $\mathcal{F} :: pc_{src}, d_d, d_e \vdash \Sigma_1, \kappa \xrightarrow{\alpha}_{pc} \Sigma_1', ks_1$. The proof also uses Lemmas 95 and 100 □

Lemma 100. *If* $\mathcal{F} :: pc_{src}, d_d, d_e \vdash \Sigma_1, \sigma, c \xrightarrow{\alpha}_{pc} \Sigma_1', \sigma_1, c_1, E_1$ *with* $pc \downarrow^p \sqsubseteq l$ *and* $\Sigma_1 \approx_l^p \Sigma_2$*, then* $\exists \mathcal{G} :: pc_{src}, d_d, d_e \vdash \Sigma_2, \sigma, c \xrightarrow{\alpha}_{pc} \Sigma_2', \sigma_2, c_2, E_2$ *with* $\Sigma_1' \approx_l^p \Sigma_2'$*,* $\sigma_1 = \sigma_2$*,* $c_1 = c_2$*, and* $E_1 = E_2$

Proof (sketch): By induction on the structure of $\mathcal{F} :: pc_{src}, d_d, d_e \vdash \Sigma_1, \sigma, c \xrightarrow{\alpha}_{pc} \Sigma_1', \sigma_1, c_1, E_1$ □

Lemma 101. *If* $T = \mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K \overset{\tau}{\Longrightarrow}{}^* K'$ *with* $\forall(\alpha, pc) \in \tau, \alpha \in \{ch(_), \bullet\} \wedge pc \downarrow^p \not\sqsubseteq l$*, then* $T \downarrow_l^p = \cdot$

Proof (sketch): By induction on the length of T □

Lemma 102. *If* $\Sigma_1 \approx_l^p \Sigma_2$ *with* $pc \downarrow^p \sqsubseteq l$*, with* $\mathsf{robust}(\Sigma_1, E, pc) = E_1$ *and* $\mathsf{robust}(\Sigma_2, E, pc) = E_2$*, then* $E_1 \approx_l^p E_2$ *if* $p = c$ *and* $E_1 = E_2$ *if* $p = i$

Proof (sketch): By induction on the structure of $\mathcal{F} :: \mathsf{robust}(\Sigma_1, E, pc) = E_1$ and $\mathcal{G} :: \mathsf{robust}(\Sigma_2, E, pc) = E_2$ □

Lemma 103. *If* $\Sigma_1 \approx_l^p \Sigma_2$ *and* $pc \downarrow^p \sqsubseteq l$*, with* $\mathsf{transparent}(\Sigma_1, E, pc) = E_1$ *and* $\mathsf{transparent}(\Sigma_2, E, pc) = E_2$*, then* $E_1 \approx_l^p E_2$ *if* $p = c$ *and* $E_1 = E_2$ *if* $p = i$

Proof (sketch): The proof is by induction on the structure of $\mathcal{F} :: \mathsf{transparent}(\Sigma_1, E, pc) = E_1$ and $\mathcal{G} :: \mathsf{transparent}(\Sigma_2, E, pc) = E_2$, similar to the one for Lemma 102. □

Lemma 104. *If* $\Sigma_1 \approx_l^i \Sigma_2$*, with* $pc_{\mathsf{Ev}}, \mathsf{r} \vdash \Sigma_1, E \rightsquigarrow ks_1$*,* $pc_{\mathsf{Ev}}, \mathsf{r} \vdash \Sigma_2, E \rightsquigarrow ks_2$ *and* $pc_{\mathsf{Ev}} \downarrow^i \sqsubseteq l$ $E = \mathsf{robust}(\Sigma_1, _, pc) = \mathsf{robust}(\Sigma_2, _, pc)$ *then* $ks_1 = ks_2$

Proof (sketch): By induction on the structure of $\mathcal{F} :: pc_{\mathsf{Ev}}, \mathsf{f} \vdash \Sigma_1, E \rightsquigarrow ks_1$ and $\mathcal{G} :: pc_{\mathsf{Ev}}, \mathsf{f} \vdash \Sigma_2, E \rightsquigarrow ks_2$ □

Lemma 105. *If* $\Sigma_1 \approx_l^c \Sigma_2$*, with* $pc_{\mathsf{Ev}}, \mathsf{t} \vdash \Sigma_1, E \rightsquigarrow ks_1$*,* $pc_{\mathsf{Ev}}, \mathsf{t} \vdash \Sigma_2, E \rightsquigarrow ks_2$ *and* $pc_{\mathsf{Ev}} \downarrow^c \sqsubseteq l$ $E = \mathsf{transparent}(\Sigma_1, _, pc) = \mathsf{transparent}(\Sigma_2, _, pc)$ *then* $ks_1 = ks_2$

Proof (sketch): The proof is by induction on the structure of $\mathcal{F} :: pc_{\mathsf{Ev}}, \mathsf{t} \vdash \Sigma_1, E \rightsquigarrow ks_1$ and $\mathcal{G} :: pc_{\mathsf{Ev}}, \mathsf{t} \vdash \Sigma_2, E \rightsquigarrow ks_2$, similar to Lemma 104. □

Lemma 106 (Strong One-step – Downgrade)**.** *If* $K_1 \approx_l^p K_2$*,* $T_1 = \mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_1 \overset{\alpha_{l,1}}{\Longrightarrow} K_1'$ *with* $T_1 \downarrow_l^p = \tau = \mathsf{down}(_)$ *and* $\mathsf{prog}(K_2)$*, with* $\mathsf{releaseT}(K_2, \tau, l)$ *if* $p = c$*, and* $\mathsf{sanitizeT}(K_2, \tau, l)$ *if* $p = i$*, then* $\exists K_2', T_2$ *s.t.* $T_2 = \mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_2 \Longrightarrow^* K_2'$ *with* $T_1 \approx_l^p T_2$ *and* $K_1' \approx_l^p K_2'$

Proof.

Denote $\mathcal{F} :: \mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_1 \overset{\alpha_{l,1}}{\Longrightarrow} K_1'$

Without loss of generality, assume that $p = c$ (the proof for $p = i$ is similar)

We refer to the following assumptions throughout:

(1) $K_1 \approx_l^p K_2$; (2) $T_1 \downarrow_l^p = \tau = \mathsf{down}(_)$; (3) $\mathsf{prog}(K_2)$; (4) $\mathsf{releaseT}(K_2, \tau, l)$

From (2), $\mathcal{F}$ ends in IN-DE, meaning $\exists K_C, K_2'$ s.t. $T_2 = \mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_2 \Longrightarrow^* K_C \overset{\alpha_{l,2}}{\Longrightarrow} K_2'$ with $\mathsf{consumer}(K_C)$, $(\mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_2 \Longrightarrow^* K_C) \downarrow_l^p = \cdot$, and $T_2 \downarrow_l^p = \mathsf{down}(...)$

From Lemma 86, $K_2 \approx_l^p K_C$

From (1) and $K_2 \approx_l^p K_C$, $K_1 \approx_l^p K_C$

From (2), $\mathsf{consumer}(K_C)$, and $T_2 \downarrow_l^p = \mathsf{down}(...)$, we know that $\mathcal{F}$ ends in IN-DE and all of the following:

$\alpha_{l,1} = (id.\mathsf{Ev}(v), pc)$; $\mathcal{P}(id.\mathsf{Ev}(v)) = pc'$; $\Sigma_1(pc) = (_, \sigma^{EH})$; $\sigma^{EH}(id) \downarrow^i \sqsubseteq pc \downarrow^i$; $\sigma^{EH}(id) \downarrow^c \sqsubseteq pc \downarrow^c$;

$E_1 = ((id.\mathsf{Ev}(v), pc'') \,|\, pc \sqcup pc' \sqsubseteq pc'')$; $\Sigma_1, E_1 \rightsquigarrow ks_1''$; $\mathsf{downgrade}_{\mathcal{D}}(\mathcal{R}_1, \Sigma_1, \alpha_{l,1}, pc') = (\mathcal{R}_1', E_{d,1})$;

$pc, \mathsf{r} \vdash \Sigma_1, E_{d,1} \rightsquigarrow ks_{d,1}$; $\mathsf{downgrade}_{\mathcal{E}}(\mathcal{S}_1, \Sigma_1, \alpha_{l,1}, pc') = (\mathcal{S}_1', E_{e,1})$; $pc, \mathsf{t} \vdash \Sigma_1, E_{e,1} \rightsquigarrow ks_{e,1}$;

$\mathsf{downgrade}_{\mathcal{D},\mathcal{E}}(\mathcal{R}_1, \Sigma_1, \alpha_{l,1}, pc') = E_{m,1}$; $pc, \mathsf{rt} \vdash \Sigma_1, E_{m,1} \rightsquigarrow ks_{m,1}$; $\Sigma_1' = \Sigma_1$; and $ks_1' = ks_1'' :: ks_{d,1} :: ks_{e,1} :: ks_{m,1}$

From the definition of $\mathsf{downgrade}_{\mathcal{D}}$, all of the following:

$\mathcal{R}_1 = (\rho_{d,1}, d_{d,1})$; $E_{d,1}' = ((id.\mathsf{Ev}(v), (l_c, l_i)) \,|\, pc \downarrow^c \sqsubseteq l_c \sqsubset pc' \downarrow^c \wedge l_i = pc \downarrow^i \sqcup pc' \downarrow^i)$;

$\mathcal{D}((id.\mathsf{Ev}(v), pc), pc', \rho_{d,1}) = (\rho_{d,1}', v_{d,1}, E_{d,1}'')$; $d_{d,1}' = \mathsf{update}(d_{d,1}, v_{d,1})$; $\mathcal{R}_1' = (\rho_{d,1}', d_{d,1}')$; and

$E_{d,1} = \mathsf{robust}(\Sigma_1, E_{d,1}' :: E_{d,1}'', pc)$

From the definition of $\mathsf{downgrade}_{\mathcal{E}}$, all of the following:

$\mathcal{S}_1 = (\rho_{e,1}, d_{e,1})$; $E_{e,1}' = ((id.\mathsf{Ev}(v), (l_c, l_i)) \,|\, pc \downarrow^i \sqsubseteq l_i \sqsubset pc' \downarrow^i \wedge l_c = pc \downarrow^c \sqcup pc' \downarrow^c)$;

$\mathcal{E}((id.\mathsf{Ev}(v), pc), pc', \rho_{e,1}) = (\rho_{e,1}', v_{e,1}, E_{e,1}'')$; $d_{e,1}' = \mathsf{update}(d_{e,1}, v_{e,1})$; $\mathcal{S}_1' = (\rho_{e,1}', d_{e,1}')$; and

$E_{e,1} = \mathsf{transparent}(\Sigma_1, E_{e,1}' :: E_{e,1}'', pc)$

From the definition of $\mathsf{downgrade}_{\mathcal{D},\mathcal{E}}$, we know that $E_{m,1} = \mathsf{mergeEvents}(E_{d,1}' :: E_{d,1}'', E_{e,1}' :: E_{e,1}'')$

Denote $\mathcal{G} :: \mathcal{P}, \mathcal{E}, \mathcal{E} \vdash K_C \Longrightarrow K_2'$

From (2), $\tau = \mathsf{down}(id.\mathsf{Ev}(v), \tau_{\mathsf{rls}}, \tau_{\mathsf{sntz},1}, E_{m,1}, pc)$

Then, $\mathcal{G}$ ends in IN-DE with input $\alpha_{l,2} = id.\mathsf{Ev}(v)$, producing trace $T_2 = \mathcal{P}, \mathcal{D}, \mathcal{E} \vdash K_2 \Longrightarrow^* K_2'$ with

$E_2 = ((id.\mathsf{Ev}(v), pc'') \,|\, pc \sqcup pc' \sqsubseteq pc'')$; $\Sigma_C, E_2 \rightsquigarrow ks_2''$; $\mathsf{downgrade}_{\mathcal{D}}(\mathcal{R}_C, \Sigma_C, \alpha_{l,2}, pc') = (\mathcal{R}_2', E_{d,2})$;

$pc, \mathsf{r} \vdash \Sigma_C, E_{d,2} \rightsquigarrow ks_{d,2}$; $\mathsf{downgrade}_{\mathcal{E}}(\mathcal{S}_C, \Sigma_C, \alpha_{l,2}, pc') = (\mathcal{S}_2', E_{e,2})$; $pc, \mathsf{t} \vdash \Sigma_C, E_{e,2} \rightsquigarrow ks_{e,2}$;

$\mathsf{downgrade}_{\mathcal{D},\mathcal{E}}(\mathcal{R}_C, \Sigma_C, \alpha_{l,2}, pc') = E_{m,2}$; $pc, \mathsf{rt} \vdash \Sigma_C, E_{m,2} \rightsquigarrow ks_{m,2}$; $\Sigma_2' = \Sigma_C$; and

$ks_2' = ks_2'' :: ks_{d,2} :: ks_{e,2} :: ks_{m,2}$

From the definition of $\mathsf{downgrade}_{\mathcal{D}}$, all of the following:

$\mathcal{R}_C = (\rho_{d,C}, d_{d,C})$; $E_{d,2}' = ((id.\mathsf{Ev}(v), (l_c, l_i)) \,|\, pc \downarrow^c \sqsubseteq l_c \sqsubset pc' \downarrow^c \wedge l_i = pc \downarrow^i \sqcup pc' \downarrow^i)$;

$\mathcal{D}((id.\mathsf{Ev}(v), pc), pc', \rho_{d,C}) = (\rho_{d,2}', v_{d,2}, E_{d,2}'')$; $d_{d,2}' = \mathsf{update}(d_{d,C}, v_{d,2})$; $\mathcal{R}_2' = (\rho_{d,2}', d_{d,2}')$; and

$E_{d,2} = \mathsf{robust}(\Sigma_C, E_{d,2}' :: E_{d,2}'', pc)$

From the definition of $\mathsf{downgrade}_{\mathcal{E}}$,

$\mathcal{S}_C = (\rho_{e,C}, d_{e,C})$; $E'_{e,2} = ((id.\mathsf{Ev}(v), (l_c, l_i)) \mid pc \downarrow^i \sqsubseteq l_i \sqsubset pc' \downarrow^i \wedge l_c = pc \downarrow^c \sqcup pc' \downarrow^c)$;

$\mathcal{E}((id.\mathsf{Ev}(v), pc), pc', \rho_{e,C}) = (\rho'_{e,2}, v_{e,2}, E''_{e,2})$; $d'_{e,2} = \mathsf{update}(d_{e,C}, v_{e,2})$; $\mathcal{S}'_2 = (\rho'_{e,2}, d'_{e,2})$; and

$E_{e,2} = \mathsf{transparent}(\Sigma_C, E'_{e,2} :: E''_{e,2}, pc)$

From the definition of $\mathsf{downgrade}_{\mathcal{D},\mathcal{E}}$, we know that $E_{m,2} = \mathsf{mergeEvents}(E'_{d,2} :: E''_{d,2}, E'_{e,2} :: E''_{e,2})$

From (4), $\tau = \mathsf{down}(id.\mathsf{Ev}(v), \tau_{\mathsf{rls}}, \tau_{\mathsf{sntz},1}, E_{m,1}, pc)$, and $T_2 \downarrow_l^p = \mathsf{down}(...)$, we know that

$T_2 \downarrow_l^p = \mathsf{down}(id.\mathsf{Ev}(v), \tau_{\mathsf{rls}}, _, E_{m,1}, pc)$

From the definition of trDowngrade, $pc \downarrow^p \sqsubseteq l$ and $ks_{m,1} \downarrow_l^p \neq \cdot$

From $\mathcal{R}_1 = \mathcal{R}_C$, $\mathcal{R}_1 = (\rho_{d,1}, d_{d,1})$, and $\mathcal{R}_C = (\rho_{d,C}, d_{d,C})$, we know that $(\rho_{d,1}, d_{d,1}) = (\rho_{d,C}, d_{d,C})$

Similarly, from $\mathcal{S}_1 = \mathcal{S}_C$, $\mathcal{S}_1 = (\rho_{e,1}, d_{e,1})$, and $\mathcal{S}_C = (\rho_{e,C}, d_{e,C})$, we know that $(\rho_{e,1}, d_{e,1}) = (\rho_{e,C}, d_{e,C})$

Case I: $pc \downarrow^p \sqsubseteq l$

By assumption and from $K_1 \approx_l^p K_C$, we know that $\Sigma_1(pc) = \Sigma_C(pc)$

From $(\rho_{d,1}, d_{d,1}) = (\rho_{d,C}, d_{d,C})$, $\mathcal{D}((id.\mathsf{Ev}(v), pc), pc', \rho_{d,1}) = (\rho'_{d,1}, v_{d,1}, E''_{d,1})$, and

$\mathcal{D}((id.\mathsf{Ev}(v), pc), pc', \rho_{d,C}) = (\rho'_{d,2}, v_{d,2}, E''_{d,2})$, we know that $(\rho'_{d,1}, v_{d,1}, E''_{d,1}) = (\rho'_{d,2}, v_{d,2}, E''_{d,2})$

Then from $d'_{d,1} = \mathsf{update}(d_{d,1}, v_{d,1})$ and $d'_{d,2} = \mathsf{update}(d_{d,C}, v_{d,2})$, we know that $d'_{d,1} = d'_{d,2}$

From $(\rho'_{d,1}, v_{d,1}, E''_{d,1}) = (\rho'_{d,2}, v_{d,2}, E''_{d,2})$, $d'_{d,1} = d'_{d,2}$, $\mathcal{R}'_1 = (\rho'_{d,1}, d'_{d,1})$, and $\mathcal{R}'_2 = (\rho'_{d,2}, d'_{d,2})$, we know that

$\mathcal{R}'_1 = \mathcal{R}'_2$

Similarly, we know that $(\rho'_{e,1}, v_{e,1}, E''_{e,1}) = (\rho'_{e,2}, v_{e,2}, E''_{e,2})$, which gives us $d'_{e,1} = d'_{e,,2}$ and $\mathcal{S}'_1 = \mathcal{S}'_2$

From $\Sigma_1 \approx_l^p \Sigma_C$, $\Sigma'_1 = \Sigma_1$, and $\Sigma'_2 = \Sigma_C$, we know that $\Sigma'_1 \approx_l^p \Sigma'_2$

From $E_1 = ((id.\mathsf{Ev}(v), pc'') \mid pc \sqcup pc' \sqsubseteq pc'')$ and $E_2 = ((id.\mathsf{Ev}(v), pc'') \mid pc \sqcup pc' \sqsubseteq pc'')$, we know that

$E_1 = E_2$

From Lemma 95, $ks''_1 \approx_l^p ks''_2$

From $\tau = \mathsf{down}(id.\mathsf{Ev}(v), \tau_{\mathsf{rls}}, \tau_{\mathsf{sntz},1}, E_{m,1}, pc)$ and $T_2 \downarrow_l^p = \mathsf{down}(id.\mathsf{Ev}(v), \tau_{\mathsf{rls}}, _, E_{m,1}, pc)$, we know that

either $E_{d,1} \downarrow_l^p = E_{d,2} \downarrow_l^p$ or $ks_{d,1} \downarrow_l^p = ks_{d,2} \downarrow_l^p = \cdot$

From Lemma 96, $ks_{d,1} \approx_l^p ks_{d,2}$

From $E'_{e,1} = ((id.\mathsf{Ev}(v), (l_c, l_i)) \mid pc \downarrow^i \sqsubseteq l_i \sqsubset pc' \downarrow^i \wedge l_c = pc \downarrow^c \sqcup pc' \downarrow^c)$ and

$E'_{e,2} = ((id.\mathsf{Ev}(v), (l_c, l_i)) \mid pc \downarrow^i \sqsubseteq l_i \sqsubset pc' \downarrow^i \wedge l_c = pc \downarrow^c \sqcup pc' \downarrow^c)$, we know that $E'_{e,1} = E'_{e,2}$

Then from $\Sigma_1 \approx_l^p \Sigma_C$, $(\rho'_{e,1}, v_{e,1}, E''_{e,1}) = (\rho'_{e,2}, v_{e,2}, E''_{e,2})$, $E_{e,1} = \mathsf{transparent}(\Sigma_1, E'_{e,1} :: E''_{e,1}, pc)$ and

$E_{e,2} = \mathsf{transparent}(\Sigma_C, E'_{e,2} :: E''_{e,2}, pc)$, we know that $E_{e,1} = E_{e,2}$

Then from $\Sigma_1 \approx_l^p \Sigma_C$, $pc, \mathsf{t} \vdash \Sigma_1, E_{e,1} \rightsquigarrow ks_{e,1}$, and $pc, \mathsf{t} \vdash \Sigma_C, E_{e,2} \rightsquigarrow ks_{e,2}$, we know that $ks_{e,1} = ks_{e,2}$

From $\tau = \mathsf{down}(id.\mathsf{Ev}(v), \tau_{\mathsf{rls}}, \tau_{\mathsf{sntz},1}, E_{m,1}, pc)$ and $T_2 \downarrow_l^p = \mathsf{down}(id.\mathsf{Ev}(v), \tau_{\mathsf{rls}}, _, E_{m,1}, pc)$, we know that

$E_{m,1} \downarrow_l^p = E_{m,2} \downarrow_l^p$

From Lemma 96, $ks_{m,1} \approx_l^p ks_{m,2}$

Then, from $ks'_1 = ks''_1 :: ks_{d,1} :: ks_{e,1} :: ks_{m,1}$ and $ks'_2 = ks''_2 :: ks_{d,2} :: ks_{e,2} :: ks_{m,2}$, we know that $ks'_1 \approx_l^p ks'_2$

Thus, from $\tau = \mathsf{down}(id.\mathsf{Ev}(v), \tau_{\mathsf{rls}}, \tau_{\mathsf{sntz},1}, E_{m,1}, pc)$, $T_2 \downarrow_l^p = \mathsf{down}(id.\mathsf{Ev}(v), \tau_{\mathsf{rls}}, _, E_{m,1}, pc)$, $(\rho'_{e,1}, v_{e,1}, E''_{e,1}) = (\rho'_{e,2}, v_{e,2}, E''_{e,2})$, and $ks_{e,1} = ks_{e,2}$, we know that $T_1 \approx_l^p T_2$

And, from $\mathcal{R}'_1 = \mathcal{R}'_2$, $\mathcal{S}'_1 = \mathcal{S}'_2$, $\Sigma'_1 \approx_l^p \Sigma'_2$, and $ks'_1 \approx_l^p ks'_2$, we know that $K'_1 \approx_l^p K'_2$

Case II: $pc \downarrow^p \not\sqsubseteq l$

From $(\rho_{d,1}, d_{d,1}) = (\rho_{d,C}, d_{d,C})$, $\mathcal{D}((id.\mathsf{Ev}(v), pc), pc', \rho_{d,1}) = (\rho'_{d,1}, v_{d,1}, E''_{d,1})$, and $\mathcal{D}((id.\mathsf{Ev}(v), pc), pc', \rho_{d,C}) = (\rho'_{d,2}, v_{d,2}, E''_{d,2})$, we know that $(\rho'_{d,1}, v_{d,1}, E''_{d,1}) = (\rho'_{d,2}, v_{d,2}, E''_{d,2})$

Then from $d'_{d,1} = \mathsf{update}(d_{d,1}, v_{d,1})$ and $d'_{d,2} = \mathsf{update}(d_{d,C}, v_{d,2})$, we know that $d'_{d,1} = d'_{d,2}$

From $(\rho'_{d,1}, v_{d,1}, E''_{d,1}) = (\rho'_{d,2}, v_{d,2}, E''_{d,2})$, $d'_{d,1} = d'_{d,2}$, $\mathcal{R}'_1 = (\rho'_{d,1}, d'_{d,1})$, and $\mathcal{R}'_2 = (\rho'_{d,2}, d'_{d,2})$, we know that $\mathcal{R}'_1 = \mathcal{R}'_2$

Similarly, we know that $(\rho'_{e,1}, v_{e,1}, E''_{e,1}) = (\rho'_{e,2}, v_{e,2}, E''_{e,2})$, which gives us $d'_{e,1} = d'_{e,,2}$ and $\mathcal{S}'_1 = \mathcal{S}'_2$

From $\Sigma_1 \approx_l^p \Sigma_C$, $\Sigma'_1 = \Sigma_1$, and $\Sigma'_2 = \Sigma_C$, we know that $\Sigma'_1 \approx_l^p \Sigma'_2$

From $E_1 = ((id.\mathsf{Ev}(v), pc'') \mid pc \sqcup pc' \sqsubseteq pc'')$ and $E_2 = ((id.\mathsf{Ev}(v), pc'') \mid pc \sqcup pc' \sqsubseteq pc'')$, we know that $E_1 = E_2$

From Lemma 95, $ks''_1 \approx_l^p ks''_2$

From $\tau = \mathsf{down}(id.\mathsf{Ev}(v), \tau_{\mathsf{rls}}, \tau_{\mathsf{sntz},1}, E_{m,1}, pc)$ and $T_2 \downarrow_l^p = \mathsf{down}(id.\mathsf{Ev}(v), \tau_{\mathsf{rls}}, _, E_{m,1}, pc)$, we know that either $E_{d,1} \downarrow_l^p = E_{d,2} \downarrow_l^p$ or $ks_{d,1} \downarrow_l^p = ks_{d,2} \downarrow_l^p = \cdot$

By assumption and from $p = c$, $\mathcal{E}((id.\mathsf{Ev}(v), pc), pc', \rho_{e,1}) = (\rho'_{e,1}, v_{e,1}, E''_{e,1})$, $\mathcal{E}((id.\mathsf{Ev}(v), pc), pc', \rho_{e,C}) = (\rho'_{e,2}, v_{e,2}, E''_{e,2})$, and since $\mathcal{E}$ will only change l_c to be more secret, we know that $E''_{e,1} \downarrow_l^p = E''_{e,2} \downarrow_l^p = \cdot$

From $p = c$, $E'_{e,1} = ((id.\mathsf{Ev}(v), (l_c, l_i)) \mid pc \downarrow^i \sqsubseteq l_i \sqsubset pc' \downarrow^i \wedge l_c = pc \downarrow^c \sqcup pc' \downarrow^c)$, and $E'_{e,2} = ((id.\mathsf{Ev}(v), (l_c, l_i)) \mid pc \downarrow^i \sqsubseteq l_i \sqsubset pc' \downarrow^i \wedge l_c = pc \downarrow^c \sqcup pc' \downarrow^c)$, we know that $E_{e,1'} \downarrow_l^p = E'_{e,2} \downarrow_l^p = \cdot$

Then from $E_{e,1} = \mathsf{transparent}(\Sigma_1, E'_{e,1} :: E''_{e,1}, pc)$ and $E_{e,2} = \mathsf{transparent}(\Sigma_C, E'_{e,2} :: E''_{e,2}, pc)$, we know that $E_{e,1} \downarrow_l^p = E_{e,2} \downarrow_l^p = \cdot$

From Lemma 97, $ks_{e,1} \downarrow_l^p = ks_{e,2} \downarrow_l^p = \cdot$

From $\tau = \mathsf{down}(id.\mathsf{Ev}(v), \tau_{\mathsf{rls}}, \tau_{\mathsf{sntz},1}, E_{m,1}, pc)$ and $T_2 \downarrow_l^p = \mathsf{down}(id.\mathsf{Ev}(v), \tau_{\mathsf{rls}}, _, E_{m,1}, pc)$, we know that $E_{m,1} \downarrow_l^p = E_{m,2} \downarrow_l^p$

From Lemma 96, $ks_{m,1} \approx_l^p ks_{m,2}$

Then, from $ks'_1 = ks''_1 :: ks_{d,1} :: ks_{e,1} :: ks_{m,1}$ and $ks'_2 = ks''_2 :: ks_{d,2} :: ks_{e,2} :: ks_{m,2}$, we know that $ks'_1 \approx_l^p ks'_2$

Thus, from $p = c$, $\tau = \mathsf{down}(id.\mathsf{Ev}(v), \tau_{\mathsf{rls}}, \tau_{\mathsf{sntz},1}, E_{m,1}, pc)$, $T_2 \downarrow_l^p = \mathsf{down}(id.\mathsf{Ev}(v), \tau_{\mathsf{rls}}, _, E_{m,1}, pc)$, and the definition of trTransparent, we know that $T_1 \approx_l^p T_2$

And, from $\mathcal{R}'_1 = \mathcal{R}'_2$, $\mathcal{S}'_1 = \mathcal{S}'_2$, $\Sigma'_1 \approx^p_l \Sigma'_2$, and $ks'_1 \approx^p_l ks'_2$, we know that $K'_1 \approx^p_l K'_2$

$\square$

Appendix C

Supporting Materials for Chapter 5

C.1 Additional Definitions

C.1.1 General Definitions

Value operations valOf takes a value or node and returns a standard value by removing any attached labels. It is not defined for faceted values or nodes. labOf takes a value or a node and a pc and returns the label on the value, if there is one, or the pc if the value is not labeled.

State, G projection $\sigma^G \downarrow_{EH}$ returns the event handler store from the shared storage, while $\sigma^G \downarrow_g$ returns the variable store from the shared storage. $G \downarrow_{EH}$ returns the enforcement mechanism for the event handler storage and $G \downarrow_g$ returns the enforcement mechanism for the shared variable storage.

consumer/producer The consumer predicate holds for a compositional configuration if there are no event handlers running (i.e., the configuration stack ks is empty). If there is at least one element in ks, the producer predicate holds. The consumer predicate holds for a configuration stack if the configuration on the top of the stack is in consumer state (i.e., the consumer(κ) predicate holds). Similarly, the producer predicate holds for a configuration stack if the configuration on the top of the stack is in producer state (i.e., the producer(κ) predicate holds). For $ks = \cdot$, neither predicate holds for ks.

C.1.2 Operations on Faceted Values

Updating/accessing facets Set and get facet are for creating and updating faceted values. Note for getFacet and dv: the type of the default value can be inferred from the other facet. To keep things simple, we use the same default value for all types.

$$\frac{}{\mathsf{getFacetA}(a, \cdot) = a} \qquad \frac{a \downarrow_{pc_l} \neq \cdot}{\mathsf{getFacetA}(a, pc_l) = a \downarrow_{pc_l}} \qquad \frac{a \downarrow_{pc_l} = \cdot}{\mathsf{getFacetA}(a, pc_l) = \mathsf{NULL}}$$

Optimization for faceted events We merge locally triggered events in MF so that if the same event handler is triggered in both the H and L context, it runs just once in the $\cdot$ context rather than running twice.

$$\frac{}{\mathsf{mergeEvs}((id.\mathsf{Ev}(v), H), (id.\mathsf{Ev}(v), L)) = (id.\mathsf{Ev}(v), \cdot)} \text{ MERGEEV-SAME}$$

$$\frac{id \neq id' \quad \text{or} \quad \mathsf{Ev} \neq \mathsf{Ev}' \quad \text{or} \quad v \neq v'}{\mathsf{mergeEvs}((id.\mathsf{Ev}(v), H), (id'.\mathsf{Ev}'(v'), L)) = (id.\mathsf{Ev}(v), H), (id'.\mathsf{Ev}'(v'), L)} \text{ MERGEEV-DIFF}$$

$$\frac{E_H = \cdot \text{ or } E_L = \cdot}{\mathsf{mergeEvs}(E_H, E_L) = E_H :: E_L} \text{ MERGEEV-S1}$$

Faceted value projection The L projection of a faceted value is the L facet and likewise for H. The projection of standard values is the same value. Note that we use projection rather than getFacet for determining if two stores are equivalent so if the projection produces $\cdot$ it means the value has not been initiated in that context. We return $\cdot$ to hide that value instead of returning dv so that a store with a $\cdot$ facet and a store without the value at all will still be equivalent.

C.1.3 Operations on Event Handlers

We define two projection operations for event handlers. These are used by the event handler lookup semantics. $\downarrow_{pc}$ returns all of the event handler with label visible to pc. When the $pc = \cdot$ we return all event handlers. $@_{pc}$ is the same as $\downarrow_{pc}$ for $pc \sqsubseteq L$, but when $pc = H$ it returns only the H and $\cdot$ event handlers.

$\boxed{(eh), l) \downarrow_{pc}}$

$$\frac{}{(id.\mathsf{Ev}(v), l) \downarrow_{\cdot} = (id.\mathsf{Ev}(v), l)} \qquad \frac{l \not\sqsubseteq pc_l}{(id.\mathsf{Ev}(v), l) \downarrow_{pc_l} = \cdot} \qquad \frac{l \sqsubseteq pc_l}{(id.\mathsf{Ev}(v), l) \downarrow_{pc_l} = (id.\mathsf{Ev}(v), l)}$$

$\boxed{(eh, l)@_{pc}}$

$$\frac{pc \sqsubseteq L \quad l \sqsubseteq L}{(eh, l)@pc = (eh, l)} \qquad \frac{pc \not\sqsubseteq L \quad pc = l \vee pc = \cdot}{(eh, l)@pc = (eh, pc)} \qquad \frac{pc \sqsubseteq L \quad l \not\sqsubseteq L}{(eh, l)@pc = \cdot} \qquad \frac{pc \not\sqsubseteq L \quad pc = l \vee pc = \cdot}{(eh, l)@pc = (eh, pc)}$$

C.1.4 Operations on SMS stores

getStore takes an SMS store and a *pc* and returns the copy of the store indicated by the *pc*. setStoreVar updates the SMS shared variable storage. It takes a compositional store σ^G, a *pc*, and an updated copy of the SMS store indicated by the *pc*. It updates σ^G so that the *pc* copy of the global variable store is updated to the given store. setStore does something similar for the SMS EH store.

C.1.5 Well-formed Initial State

We say that an initial state is well-formed if all of the following are true:

- Every global variable has been initialized. We assume the set of global variables is static; no new variables are created and the ones which exist in the initial state persist for the duration of the execution
- For the SMS event handler store, the event handlers in the H copy of the store only contains event handlers with label H, and likewise for the event handlers in the L copy
- For the TS event handler store, tainted nodes may only contain event handlers with label H. Untainted nodes may contain event handlers labeled L or H

C.2 Complete Semantics

C.2.1 Framework Semantics

The top-most level for our compositional framework processes user input events and outputs to channels. These rules govern how inputs trigger event handlers and how outputs are processed and use the judgement $G, \mathcal{P} \vdash K \stackrel{\alpha}{\Longrightarrow} K'$, meaning the compositional configuration K can step to K' given input α or producing output α under the compositional enforcement G and label context $\mathcal{P}$. The rules are shown in Figures 5.2 and 5.3 in Chapter 5.3.

Event handler lookup semantics The event handler lookup semantics use the judgement $G, \mathcal{V}, \sigma^G \vdash ks; \mathsf{lookupEhAPI}(...) \rightsquigarrow_{pc} ks'$ and the rules are shown in Figure C.1. They take the global store and current configuration stack, ks, use a helper function by the same name as the lookupEhAPI to look up a list of relevant event handlers and the context they should run in ($\mathcal{C}$). The rules are shown in Figure C.2. To turn this list into a configuration stack, createK looks up the enforcement mechanism, $\mathcal{V}$, and formats the configuration depending on which mechanism is used ($\mathsf{crtK}_{\mathcal{V}}(...)$). The final result is another configuration stack, ks', which is appended to the old one $ks :: ks'$.

$$\boxed{G, \mathcal{V}, \sigma^G \vdash ks; \mathsf{lookupEhAPI}(...) \rightsquigarrow_{pc} ks'}$$

$$\frac{\mathcal{C} = \mathsf{lookupEhAPI}_G(\sigma^G, id.\mathsf{Ev}(v), pc) \qquad ks' = \mathsf{createK}(\mathcal{P}, id.\mathsf{Ev}(v), \mathcal{C})}{G, \mathcal{P}, \sigma^G \vdash ks; \mathsf{lookupEhAPI}(id.\mathsf{Ev}(v)) \rightsquigarrow_{pc} ks :: ks'} \text{ LOOKUPEHAPI}$$

$$\frac{\begin{array}{c} ks = (\mathcal{V}; \kappa; pc) :: _ \\ \mathcal{C} = \mathsf{lookupEHs}_{G,\mathcal{V}}(\sigma^G, id.\mathsf{Ev}(v), pc \sqcup l) \qquad ks' = \mathsf{createK}(\mathcal{P}, id.\mathsf{Ev}(v), \mathcal{C}) \\ G, \mathcal{P}, \sigma^G \vdash ks :: ks'; \mathsf{lookupEHs}(E) \rightsquigarrow_{pc} ks'' \end{array}}{G, \mathcal{P}, \sigma^G \vdash ks; \mathsf{lookupEHs}((id.\mathsf{Ev}(v), l), E) \rightsquigarrow_{pc} ks''} \text{ LOOKUPEHS-R}$$

$$\frac{}{G, \mathcal{P}, \sigma^G \vdash ks; \mathsf{lookupEHs}(\cdot) \rightsquigarrow_{pc} ks} \text{ LOOKUPEHS-S}$$

$$\frac{\mathcal{P}(id.\mathsf{Ev}(v), eh, pc) = \mathcal{V}}{\mathsf{createK}(\mathcal{P}, id.\mathsf{Ev}(v), ((c, pc), \mathcal{C})) = \mathsf{crtK}_{\mathcal{V}}(eh, v, pc) :: \mathsf{createK}(\mathcal{P}, id.\mathsf{Ev}(v), \mathcal{C})} \qquad \frac{}{\mathsf{createK}(\mathcal{P}, id.\mathsf{Ev}(v), \cdot) = \cdot}$$

$$\frac{}{\begin{array}{c}\mathsf{crtK}_{\mathsf{SME}}(eh, v, L) = \\ (\mathsf{SME}; ((\cdot, \mathsf{skip}, C, \cdot); (\cdot, eh(v), P, \cdot)); L)\end{array}} \qquad \frac{}{\begin{array}{c}\mathsf{crtK}_{\mathsf{SME}}(eh, v, H) = \\ (\mathsf{SME}; ((\cdot, eh(v), P, \cdot); (\cdot, \mathsf{skip}, C, \cdot)); H)\end{array}}$$

$$\frac{}{\mathsf{crtK}_{\mathsf{SME}}(eh, v, \cdot) = (\mathsf{SME}; ((\cdot, eh(v), P, \cdot); (\cdot, eh(v), P, \cdot)); L)} \qquad \frac{}{\mathsf{crtK}_{\mathsf{MF}}(eh, v, l) = (\mathsf{MF}; (\cdot, eh(v), P, \cdot); l)}$$

$$\frac{l \sqsubseteq L}{\mathsf{crtK}_{\mathsf{TT}}(eh, v, l) = (\mathsf{TT}; (\cdot, eh(v), P, \cdot); L)} \qquad \frac{l \not\sqsubseteq L}{\mathsf{crtK}_{\mathsf{TT}}(eh, v, l) = (\mathsf{TT}; (\cdot, eh(v), P, \cdot); H)}$$

Figure C.1: Event handler lookup semantics

lookupEHAt performs a join because the @ operation returns everything with label $l \in \{pc, \cdot\}$, but we want to run them at pc. lookupEHAll returns all event handlers visible at that pc and runs them at the join of their label with the pc. lookupEHs returns the event handlers for a locally triggered event.

Output and output conditions $\mathsf{outCondition}_{\mathcal{V}}(...)$ determines if an output should be allowed. The output condition for SME event handlers is that the pc matches the label on the channel ($\mathcal{P}(ch)$) since SME is only allowed to output to channels matching the execution. Similarly for MF, if the pc is a standard label (i.e., L or H), the label on the channel must match the pc for the output to succeed. If the $pc = \cdot$, then we check that the value being output is visible to the channel (i.e., $v \downarrow_{\mathcal{P}(ch)} \neq \cdot$). If it is, the output succeeds, otherwise, the output fails. Finally, for TT, outputs succeed if the data is visible to the channel. This means that the data itself as well as the context the data was generated in should both be visible to the channel (i.e., $pc \sqcup l \sqsubseteq \mathcal{P}(ch)$).

$$\boxed{\mathsf{lookupEhAPI}_G(\sigma^G, pc, id.\mathsf{Ev}(v)) = \mathcal{C}}$$

$$\frac{\begin{array}{c}\mathsf{lookup}_{G\downarrow_{EH}}(\sigma, pc_l, id) = \phi \qquad \mathsf{valOf}(\phi) \neq \mathsf{NULL}\\ \mathsf{labOf}(\phi, pc_l) = l \qquad \mathcal{C} = (\phi.M(\mathsf{Ev})@pc_l) \sqcup pc_l \sqcup l\end{array}}{\mathsf{lookupEHAt}_G(\sigma, pc_l, id.\mathsf{Ev}(v)) = \mathsf{mergeC}(\mathcal{C})}\ \textsc{lookupEHAt}$$

$$\frac{\begin{array}{c}\mathsf{lookup}_{G\downarrow_{EH}}(\sigma, pc_l, id) = \phi\\ \mathsf{valOf}(\phi) = \mathsf{NULL}\end{array}}{\mathsf{lookupEHAt}_G(\sigma, pc_l, id.\mathsf{Ev}(v)) = \cdot}\ \textsc{lookupEHAt-s}$$

$$\frac{\begin{array}{c}\mathsf{lookupEHAt}_G(\sigma, H, id.\mathsf{Ev}(v)) = \mathcal{C}_H\\ \mathsf{lookupEHAt}_G(\sigma, L, id.\mathsf{Ev}(v)) = \mathcal{C}_L \qquad \mathsf{mergeC}(\mathcal{C}_H, \mathcal{C}_L) = \mathcal{C}\end{array}}{\mathsf{lookupEHAt}_G(\sigma, \cdot, id.\mathsf{Ev}(v)) = \mathcal{C}}\ \textsc{lookupEHAt-nc}$$

$$\frac{\begin{array}{c}\mathsf{lookup}_{G\downarrow_{EH}}(\sigma, pc_l, id) = \phi\\ \mathsf{valOf}(\phi) \neq \mathsf{NULL} \qquad \mathsf{labOf}(\phi, pc_l) = l \qquad \mathcal{C} = (\phi.M(\mathsf{Ev}) \downarrow_{pc_l}) \sqcup pc_l \sqcup l\end{array}}{\mathsf{lookupEHAll}_G(\sigma, pc_l, id.\mathsf{Ev}(v)) = \mathsf{mergeC}(\mathcal{C})}\ \textsc{lookupEHAll}$$

$$\frac{\mathsf{lookup}_{G\downarrow_{EH}}(\sigma, pc, id) = \phi \qquad \mathsf{valOf}(\phi) = \mathsf{NULL}}{\mathsf{lookupEHAll}_G(\sigma, pc_l, id.\mathsf{Ev}(v)) = \cdot}\ \textsc{lookupEHAll-s}$$

$$\frac{\mathsf{lookupEHAll}_G(\sigma, H, id.\mathsf{Ev}(v)) = \mathcal{C}_H \qquad \mathsf{lookupEHAll}_G(\sigma, L, id.\mathsf{Ev}(v)) = \mathcal{C}_L}{\mathsf{lookupEHAll}_{G,\mathcal{V}}(\sigma, \cdot, id.\mathsf{Ev}(v)) = \mathsf{mergeC}(\mathcal{C}_H, \mathcal{C}_L)}\ \textsc{lookupEHAll-nc-merge}$$

$$\frac{\mathcal{V} \neq \mathsf{TT} \qquad \mathcal{C} = \mathsf{lookupEHAt}_G(\sigma, pc, id.\mathsf{Ev}(v))}{\mathsf{lookupEHs}_{G,\mathcal{V}}(\sigma, pc, id.\mathsf{Ev}(v)) = \mathcal{C}} \qquad \frac{pc \sqsubseteq L \qquad \mathcal{C} = \mathsf{lookupEHAll}_G(\sigma, \cdot, id.\mathsf{Ev}(v))}{\mathsf{lookupEHs}_{G,\mathsf{TT}}(\sigma, pc, id.\mathsf{Ev}(v)) = \mathcal{C}}$$

$$\frac{pc \not\sqsubseteq L \qquad \mathcal{C} = \mathsf{lookupEHAll}_G(\sigma, H, id.\mathsf{Ev}(v))}{\mathsf{lookupEHs}_{G,\mathsf{TT}}(\sigma, pc, id.\mathsf{Ev}(v)) = \mathcal{C}}$$

$$\boxed{\mathsf{mergeC}(\mathcal{C}, \mathcal{C}') = \mathcal{C}''}$$

$$\frac{(eh, l') \notin \mathcal{C}}{\mathsf{mergeC}((eh, l), \mathcal{C}) = (eh, l), \mathsf{mergeC}(\mathcal{C})} \qquad \frac{\mathcal{C} = (\mathcal{C}', (eh, l'), \mathcal{C}'')}{\mathsf{mergeC}((eh, l), \mathcal{C}) = \mathsf{mergeC}((eh, l \wedge l'), \mathcal{C}', \mathcal{C}'')}$$

Figure C.2: Event handler lookup helper functions

$$\boxed{G, \mathcal{V}, d \vdash \sigma_1^G, \kappa^{\mathcal{V}} \xrightarrow{\alpha}_{pc} \sigma_2^G, ks}$$

$$\frac{G, \mathcal{P}, \mathsf{SME}, d \vdash \sigma_1^G, \kappa_L \xrightarrow{\alpha}_L \sigma_2^G, (\mathsf{SME}; \kappa_L'; L) :: ks \qquad \neg\mathsf{consumer}(\kappa_L')}{G, \mathcal{P}, \mathsf{SME}, d \vdash \sigma_1^G, \kappa_H; \kappa_L \xrightarrow{\alpha}_L \sigma_2^G, (\mathsf{SME}; (\kappa_H; \kappa_L'); L) :: ks} \text{ SME-L}$$

$$\frac{G, \mathcal{P}, \mathsf{SME}, d \vdash \sigma_1^G, \kappa_L \xrightarrow{\alpha}_L \sigma_2^G, (\mathsf{SME}; \kappa_L'; L) :: ks \qquad \mathsf{consumer}(\kappa_L')}{G, \mathcal{P}, \mathsf{SME}, d \vdash \sigma_1^G, \kappa_H; \kappa_L \xrightarrow{\alpha}_L \sigma_2^G, (\mathsf{SME}; (\kappa_H; \kappa_L'); H) :: ks} \text{ SME-LtoH}$$

$$\frac{G, \mathcal{P}, \mathsf{SME}, d \vdash \sigma_1^G, \kappa_H \xrightarrow{\alpha}_H \sigma_2^G, (\mathsf{SME}; \kappa_H'; H) :: ks}{G, \mathcal{P}, \mathsf{SME}, d \vdash \sigma_1^G, \kappa_H; \kappa_L \xrightarrow{\alpha}_H \sigma_2^G, (\mathsf{SME}; (\kappa_H'; \kappa_L); H) :: ks} \text{ SME-H}$$

$$\frac{G, \mathsf{MF}, d \Vdash \sigma_1^G, \sigma_1, c^{\mathsf{std}} \xrightarrow{\alpha}. \sigma_2^G, \sigma_2, \langle c_H | c_L \rangle, E_2}{G, \mathcal{P}, \mathsf{MF}, d \vdash \sigma_1^G, \sigma_1, c^{\mathsf{std}}, P, E_1 \xrightarrow{\alpha}. \sigma_2^G, (\mathsf{MF}; (\sigma_2, c_L, P, (E_1, E_2)); L) :: (\mathsf{MF}; (\sigma_2, c_H, P, (E_1, E_2)); H)} \text{ P-F}$$

Figure C.3: Additional rules for processing the event handler queue for SME and MF

C.2.2 EH queue semantics

The mid-level semantics are of the form: $G, \mathcal{P}, \mathcal{V} \vdash \sigma_1^G, \kappa \xrightarrow{\alpha}_{pc} \sigma_2^G, ks$ and run a single event handler κ with the given enforcement mechanism $\mathcal{V}$ and produce some output α. The rules are shown in Figure 5.5 in Chapter 5.3.2. There are a few additional rules for processing some enforcement mechanisms shown in Figure C.3. SME-L and SME-LtoH run the L execution. If the system is still in producer state after taking a step ($\neg\mathsf{consumer}(\kappa_L')$ as in SME-L), the pc remains L to continue running the L execution. Otherwise, the low execution is in consumer state ($\mathsf{consumer}(\kappa_L')$ as in SME-LtoH) and the pc switches to H to run the H execution (SME-H). P-F handles the case where MF produces a faceted command. In this case, we split the execution to run the L command with $pc = L$ and the H command with $pc = H$. Note about MF semantics: similar to the original faceted execution semantics, we split execution whenever we see faceted values, but unlike prior work, *we never go back to a joint execution*. Once the execution splits, it remains split until the event handler finishes execution.

C.2.3 EH semantics

The lower-level semantic rules for evaluating individual event handlers are triggered by the mid-level semantics in the "producer" state. The judgement for these rules is $G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1^{\mathcal{V}}, c_1^{\mathsf{std}} \xrightarrow{\alpha}_{pc} \sigma_2^G, \sigma_2^{\mathcal{V}}, c_2, E$. These rules are shown in Figure C.4 and are mostly standard and enforcement-independent, except for interactions with the store. Note: these rules are meant to be general enough to apply to any enforcement mechanism, which is why valOf appears in most rules.

$$\boxed{G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1^{\mathcal{V}}, c_1^{\mathsf{std}} \xrightarrow{\alpha}_{pc} \sigma_2^G, \sigma_2^{\mathcal{V}}, c_2, E}$$

$$\frac{}{G, \mathcal{V}, d \Vdash \sigma^G, \sigma^{\mathcal{V}}, \mathsf{skip}; c \xrightarrow{\bullet}_{pc} \sigma^G, \sigma^{\mathcal{V}}, c, \cdot}\ \textsc{skip} \qquad \frac{G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1^{\mathcal{V}}, c_1 \xrightarrow{\alpha}_{pc} \sigma_2^G, \sigma_2^{\mathcal{V}}, c_1', E}{G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1^{\mathcal{V}}, c_1; c_2 \xrightarrow{\alpha}_{pc} \sigma_2^G, \sigma_2^{\mathcal{V}}, c_1'; c_2, E}\ \textsc{seq}$$

$$\frac{G, \mathcal{V}, \sigma^G, \sigma^{\mathcal{V}} \vdash e \Downarrow_{pc}^{\mathcal{V}} v \qquad \mathsf{valOf}(v) = \mathsf{true}}{G, \mathcal{V}, d \Vdash \sigma^G, \sigma^{\mathcal{V}}, \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \xrightarrow{\bullet}_{pc} \sigma^G, \sigma^{\mathcal{V}}, c_1, \cdot}\ \textsc{if-true}$$

$$\frac{G, \mathcal{V}, \sigma^G, \sigma^{\mathcal{V}} \vdash e \Downarrow_{pc}^{\mathcal{V}} v \qquad \mathsf{valOf}(v) = \mathsf{false}}{G, \mathcal{V}, d \Vdash \sigma^G, \sigma^{\mathcal{V}}, \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \xrightarrow{\bullet}_{pc} \sigma^G, \sigma^{\mathcal{V}}, c_2, \cdot}\ \textsc{if-false}$$

$$\frac{G, \mathcal{V}, \sigma^G, \sigma^{\mathcal{V}} \vdash e \Downarrow_{pc}^{\mathcal{V}} v \qquad \mathsf{valOf}(v) = \mathsf{true}}{G, \mathcal{V}, d \Vdash \sigma^G, \sigma^{\mathcal{V}}, \mathsf{while}\ e\ \mathsf{do}\ c \xrightarrow{\bullet}_{pc} \sigma^G, \sigma^{\mathcal{V}}, c; \mathsf{while}\ e\ \mathsf{do}\ c, \cdot}\ \textsc{while-true}$$

$$\frac{G, \mathcal{V}, \sigma^G, \sigma^{\mathcal{V}} \vdash e \Downarrow_{pc}^{\mathcal{V}} v \qquad \mathsf{valOf}(v) = \mathsf{false}}{G, \mathcal{V}, d \Vdash \sigma^G, \sigma^{\mathcal{V}}, \mathsf{while}\ e\ \mathsf{do}\ c \xrightarrow{\bullet}_{pc} \sigma^G, \sigma^{\mathcal{V}}, \mathsf{skip}, \cdot}\ \textsc{while-false}$$

$$\frac{G, \mathcal{V}, \sigma^G, \sigma^{\mathcal{V}} \vdash e \Downarrow_{pc}^{\mathcal{V}} v}{G, \mathcal{V}, d \Vdash \sigma^G, \sigma^{\mathcal{V}}, \mathsf{output}\ ch\ e \xrightarrow{ch(v)}_{pc} \sigma^G, \sigma^{\mathcal{V}}, \mathsf{skip}, \cdot}\ \textsc{output}$$

$$\frac{\mathsf{read}(d, \iota) = v}{G, \mathcal{V}, d \Vdash \sigma^G, \sigma^{\mathcal{V}}, x := \mathsf{declassify}(\iota, e) \xrightarrow{\mathsf{declassify}(\iota, v)}_{L} \sigma^G, \sigma^{\mathcal{V}}, x := v, \cdot}\ \textsc{declassify-L}$$

$$\frac{}{G, \mathcal{V}, d \Vdash \sigma^G, \sigma^{\mathcal{V}}, x := \mathsf{declassify}(\iota, e) \xrightarrow{\bullet}_{H} \sigma^G, \sigma^{\mathcal{V}}, x := e, \cdot}\ \textsc{declassify-H}$$

Figure C.4: Event handler semantics

$$\boxed{G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1^{\mathcal{V}}, c_1 \xrightarrow{\alpha}_{pc} \sigma_2^G, \sigma_2^{\mathcal{V}}, c_2, E}$$

$$\frac{G, \mathsf{MF}, d \Vdash \sigma_1^G, \sigma_1^{\mathcal{V}}, c_1 \xrightarrow{\alpha}_{\cdot} \sigma_2^G, \sigma_2^{\mathcal{V}}, \langle c_H | c_L \rangle, E}{G, \mathsf{MF}, d \Vdash \sigma_1^G, \sigma_1^{\mathcal{V}}, c_1; c_2 \xrightarrow{\alpha}_{\cdot} \sigma_2^G, \sigma_2^{\mathcal{V}}, \mathsf{setFacetC}(c_H; c_2, c_L; c_2), E}\ \textsc{seq-f}$$

$$\frac{\begin{array}{c} G, \mathsf{MF}, \sigma^G, \sigma^{\mathcal{V}} \vdash e \Downarrow_{\cdot}^{\mathsf{MF}} v = \langle _ | _ \rangle \qquad v_H = \mathsf{getFacetV}(v, H) \qquad v_L = \mathsf{getFacetV}(v, L) \\ c_H = c_1\ \mathit{if}\ v_H = \mathsf{true} \qquad c_H = c_2\ \mathit{if}\ v_H = \mathsf{false} \\ c_L = c_1\ \mathit{if}\ v_L = \mathsf{true} \qquad c_L = c_2\ \mathit{if}\ v_L = \mathsf{false} \end{array}}{G, \mathsf{MF}, d \Vdash \sigma^G, \sigma^{\mathcal{V}}, \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \xrightarrow{\bullet}_{\cdot} \sigma^G, \sigma^{\mathcal{V}}, \mathsf{setFacetC}(c_H, c_L), \cdot}\ \textsc{if-f}$$

$$\frac{\begin{array}{c} G, \mathsf{MF}, \sigma^G, \sigma^{\mathcal{V}} \vdash e \Downarrow_{\cdot}^{\mathsf{MF}} v = \langle _ | _ \rangle \qquad v_H = \mathsf{getFacetV}(v, H) \qquad v_L = \mathsf{getFacetV}(v, L) \\ c_H = c; \mathsf{while}\ e\ \mathsf{do}\ c\ \mathit{if}\ v_H = \mathsf{true} \qquad c_H = \mathsf{skip}\ \mathit{if}\ v_H = \mathsf{false} \\ c_L = c; \mathsf{while}\ e\ \mathsf{do}\ c\ \mathit{if}\ v_L = \mathsf{true} \qquad c_L = \mathsf{skip}\ \mathit{if}\ v_L = \mathsf{false} \end{array}}{G, \mathsf{MF}, d \Vdash \sigma^G, \sigma^{\mathcal{V}}, \mathsf{while}\ e\ \mathsf{do}\ c \xrightarrow{\bullet}_{\cdot} \sigma^G, \sigma^{\mathcal{V}}, \mathsf{setFacetC}(c_H, c_L), \cdot}\ \textsc{while-f}$$

$$\frac{\mathsf{read}(d, \iota) = v}{G, \mathsf{MF}, d \Vdash \sigma^G, \sigma^{\mathcal{V}}, x := \mathsf{declassify}(\iota, e) \xrightarrow{\mathsf{declassify}(\iota, v)}_{\cdot} \sigma^G, \sigma^{\mathcal{V}}, \mathsf{setFacetC}(x := e, x := v), \cdot}\ \textsc{declassify-nc}$$

Figure C.5: Additional event handler semantics for MF

Faceted semantics There are a few additional rules shown in Figure C.5 for dealing with faceted values. SEQ-F handles the case where the first command in a sequence $c_1; c_2$ steps to a faceted command ($\langle c_H | c_L \rangle$). We use setFacetC to create a new faceted command where the H facet is the sequence $c_H; c_2$ and the L facet is the sequence $c_L; c_2$. IF-F handles branching on a faceted value and WHILE-F handles looping on a faceted value. In both cases, we evaluate the conditional for both facets and produce a faceted command. Finally, when evaluating a declassification with $pc = \cdot$, we create a faceted command where the command in the H facet performs the assignment directly, and the L facet assigns the declassified value.

Variable assignment The rules for variable assignment involve a function assign. The rules are shown in Figure C.8. For SME, configurations keep track of each copy of the store with the execution, so assignments can be made directly. On the other hand, to update a variable in an SMS store, we first pick the correct copy of the store using getStore, which is the copy that matches the given pc. We make the assignment in this store, then update the SMS store using setStoreVar to include the updated copy.

$x \in \sigma$ is a requirement for FS and TS. If the variable does not exist in the store, the assignment is skipped. Assignments when $pc = \cdot$ are straightforward. For a standard pc (i.e., L or H), we first create a faceted value which combines the old value from the store with the value being assigned. If the new value is faceted, getFacetV will get the correct facet and setFacetV will get the correct facet from the old value. The facet matching the pc comes from the new value, and the other facet comes from the old value: this is the value assigned to the store. In TS and TT, the value being assigned is labeled $l \sqcup pc$ so that neither the value nor the context it is assigned in cause any leaks.

C.2.4 Expression semantics

The expression semantics are shown in Figure C.7. Note that the sub-expressions are converted to the same types in each rule. So, for BOP, this means that types will not be mixed (e.g., (v, l) bop $\langle v_H | v_L \rangle$ is not possible). To keep top-level rules as separate as possible from mechanism-specific details (like the structure of the values), we assume that bop well-defined for the various types of values (i.e., standard, labeled, and faceted).

Type conversion rules Because the shared variable storage, EH storage, and event handlers may all be enforced differently from each other, computations involving different values from different enforcement mechanisms may need to be converted to a different format. The rules are shown in Figure C.7. We include the destination enforcement mechanism (i) in the expression semantics and convert the final result to whatever format is expected by this enforcement mechanism.

$$\boxed{G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1^{\mathcal{V}}, c_1^{\mathsf{std}} \xrightarrow{\alpha}_{pc} \sigma_2^G, \sigma_2^{\mathcal{V}}, c_2, E}$$

$$\frac{G, \mathcal{V}, \sigma_1^G, \sigma_1^{\mathcal{V}} \vdash e \Downarrow_{pc}^{\mathcal{V}} v \qquad x \notin \sigma_1^G \qquad \mathsf{assign}_{\mathcal{V}}(\sigma_1^{\mathcal{V}}, pc, x, v) = \sigma_2^{\mathcal{V}}}{G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1^{\mathcal{V}}, x := e \xrightarrow{\bullet}_{pc} \sigma_1^G, \sigma_2^{\mathcal{V}}, \mathsf{skip}, \cdot} \text{ ASSIGN-L}$$

$$\frac{G, \mathcal{V}, \sigma_1^G, \sigma_1^{\mathcal{V}} \vdash e \Downarrow_{pc}^{G\downarrow_g} v \qquad x \in \sigma_1^G \qquad \mathsf{assign}_{G\downarrow_g}(\sigma_1^G, pc, x, v) = \sigma_2^G}{G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1^{\mathcal{V}}, x := e \xrightarrow{\bullet}_{pc} \sigma_2^G, \sigma_1^{\mathcal{V}}, \mathsf{skip}, \cdot} \text{ ASSIGN-G}$$

$$\boxed{\mathsf{assign}_i(\sigma, pc, x, v) = \sigma'}$$

$$\frac{}{\mathsf{assign}_{\mathsf{SME}}(\sigma^{\mathsf{std}}, H, x, v) = \sigma^{\mathsf{std}}[x \mapsto v]} \text{ SME-ASSIGN-H} \qquad \frac{}{\mathsf{assign}_{\mathsf{SME}}(\sigma^{\mathsf{std}}, L, x, v) = \sigma^{\mathsf{std}}[x \mapsto v]} \text{ SME-ASSIGN-L}$$

$$\frac{\sigma = \mathsf{getStore}(\sigma^G \downarrow_g, pc_l) \qquad x \in \sigma \qquad \sigma' = \sigma[x \mapsto v]}{\mathsf{assign}_{\mathsf{SMS}}(\sigma^G, pc_l, x, v) = \mathsf{setStoreVar}(\sigma^G, pc_l, \sigma')} \text{ SMS-ASSIGN}$$

$$\frac{\sigma = \mathsf{getStore}(\sigma^G \downarrow_g, pc_l) \qquad x \notin \sigma}{\mathsf{assign}_{\mathsf{SMS}}(\sigma^G, pc_l, x, v) = \sigma^G} \text{ SMS-ASSIGN-S}$$

$$\frac{\sigma' = \sigma[x \mapsto v]}{\mathsf{assign}_{\mathsf{MF}}(\sigma, \cdot, x, v) = \sigma'} \text{ MF-ASSIGN}$$

$$\frac{v_L = \mathsf{var}_{\mathsf{MF}}(\sigma, L, x) \qquad v' = \mathsf{setFacetV}(\mathsf{getFacetV}(v, H), v_L)}{\mathsf{assign}_{\mathsf{MF}}(\sigma, H, x, v) = \sigma[x \mapsto v']} \text{ MF-ASSIGN-H}$$

$$\frac{v_H = \mathsf{var}_{\mathsf{MF}}(\sigma, H, x) \qquad v' = \mathsf{setFacetV}(v_H, \mathsf{getFacetV}(v, L))}{\mathsf{assign}_{\mathsf{MF}}(\sigma, L, x, v) = \sigma[x \mapsto v']} \text{ MF-ASSIGN-L}$$

$$\frac{x \in \sigma \qquad \sigma' = \sigma[x \mapsto v]}{\mathsf{assign}_{\mathsf{FS}}(\sigma, \cdot, x, v) = \sigma'} \text{ FS-ASSIGN} \qquad \frac{\begin{array}{c} x \in \sigma \qquad v_L = \mathsf{var}_{\mathsf{FS}}(\sigma, L, x) \\ v' = \mathsf{setFacetV}(\mathsf{getFacetV}(v, H), v_L) \end{array}}{\mathsf{assign}_{\mathsf{FS}}(\sigma, H, x, v) = \sigma[x \mapsto v']} \text{ FS-ASSIGN-H}$$

$$\frac{\begin{array}{c} x \in \sigma \qquad v_H = \mathsf{var}_{\mathsf{FS}}(\sigma, H, x) \\ v' = \mathsf{setFacetV}(v_H, \mathsf{getFacetV}(v, L)) \end{array}}{\mathsf{assign}_{\mathsf{FS}}(\sigma, L, x, v) = \sigma[x \mapsto v']} \text{ FS-ASSIGN-L} \qquad \frac{x \notin \sigma}{\mathsf{assign}_{\mathsf{FS}}(\sigma, pc, x, v) = \sigma} \text{ FS-ASSIGN-S}$$

$$\frac{}{\mathsf{assign}_{\mathsf{TT}}(\sigma, pc, x, (v, l)) = \sigma[x \mapsto (v, l \sqcup pc)]} \text{ TT-ASSIGN}$$

$$\frac{x \in \sigma}{\mathsf{assign}_{\mathsf{TS}}(\sigma, pc, x, (v, l)) = \sigma[x \mapsto (v, l \sqcup pc)]} \text{ TS-ASSIGN} \qquad \frac{x \notin \sigma}{\mathsf{assign}_{\mathsf{TS}}(\sigma, pc, x, (v, l)) = \sigma} \text{ TS-ASSIGN-S}$$

Figure C.6: Variable assignment semantics

$$\boxed{G, \mathcal{V}, \sigma^G, \sigma^{\mathcal{V}} \vdash e \Downarrow^i_{pc} v}$$

$$\frac{\text{if } x \in \sigma^G, \text{then } v = \mathsf{var}_{G\downarrow_g}(\sigma^G, pc, x) \qquad \text{otherwise}, v = \mathsf{var}_{\mathcal{V}}(\sigma^{\mathcal{V}}, pc, x) \qquad \mathsf{toDst}(v, pc, i) = v'}{G, \mathcal{V}, \sigma^G, \sigma^{\mathcal{V}} \vdash x \Downarrow^i_{pc} v'} \text{ VAR}$$

$$\frac{G, \mathcal{V}, \sigma^G, \sigma^{\mathcal{V}} \vdash e_1 \Downarrow^i_{pc} v_1 \qquad G, \mathcal{V}, \sigma^G, \sigma^{\mathcal{V}} \vdash e_2 \Downarrow^i_{pc} v_2 \qquad v = v_1 \text{ bop } v_2}{G, \mathcal{V}, \sigma^G, \sigma^{\mathcal{V}} \vdash e_1 \text{ bop } e_2 \Downarrow^i_{pc} v} \text{ BOP}$$

$$\boxed{\mathsf{toDst}(v, pc, i) = v'}$$

$$\frac{dst \in \{\mathsf{SME}, \mathsf{SMS}, \mathsf{MF}, \mathsf{FS}\}}{\mathsf{toDst}(v^{\mathsf{std}}, pc, dst) = v^{\mathsf{std}}} \text{ MS} \qquad \frac{dst \in \{\mathsf{TT}, \mathsf{TS}\} \qquad pc \sqsubseteq L}{\mathsf{toDst}(v^{\mathsf{std}}, pc, dst) = (v^{\mathsf{std}}, L)} \text{ MSTOT-L}$$

$$\frac{dst \in \{\mathsf{TT}, \mathsf{TS}\} \qquad pc \not\sqsubseteq L}{\mathsf{toDst}(v^{\mathsf{std}}, pc, dst) = (v^{\mathsf{std}}, pc)} \text{ MSTOT-H} \qquad \frac{dst \in \{\mathsf{TT}, \mathsf{TS}\}}{\mathsf{toDst}((v^{\mathsf{std}}, l), pc_l, dst) = (v^{\mathsf{std}}, l \sqcup pc_l)} \text{ T}$$

$$\frac{dst \notin \{\mathsf{TT}, \mathsf{TS}\} \qquad l \sqsubseteq pc_l}{\mathsf{toDst}((v^{\mathsf{std}}, l), pc_l, dst) = v^{\mathsf{std}}} \text{ TTOMS} \qquad \frac{dst \notin \{\mathsf{TT}, \mathsf{TS}\} \qquad l \not\sqsubseteq pc_l}{\mathsf{toDst}((v^{\mathsf{std}}, l), pc_l, dst) = \mathsf{dv}} \text{ TTOMS-DV}$$

$$\frac{}{\mathsf{toDst}((v^{\mathsf{std}}, L), \cdot, dst) = v^{\mathsf{std}}} \text{ TTOMS-NC-L} \qquad \frac{}{\mathsf{toDst}((v^{\mathsf{std}}, H), \cdot, dst) = \langle v^{\mathsf{std}} | \mathsf{dv} \rangle} \text{ TTOMS-NC-H}$$

$$\frac{v = \langle _ | _ \rangle}{\mathsf{toDst}(v, \cdot, dst) = v} \text{ NC}$$

$$\boxed{\mathsf{ehAPIe}(\mathcal{G}, \sigma, pc, id, v_1, \cdots, v_n) = v}$$

$$\frac{\begin{array}{c}\mathsf{ehAPIe}_{\mathcal{G}}(\sigma, H, id, \mathsf{getFacetV}(v_1, H), \cdots, \mathsf{getFacetV}(v_n, H)) = v_H \\ \mathsf{ehAPIe}_{\mathcal{G}}(\sigma, L, id, \mathsf{getFacetV}(v_1, L), \cdots, \mathsf{getFacetV}(v_n, L)) = v_L\end{array}}{\mathsf{ehAPIe}(\mathcal{G}, \sigma, \cdot, id, v_1, \cdots, v_n) = \mathsf{createFct}(v_H, v_L)} \text{ EHAPI-NC}$$

$$\frac{\mathsf{ehAPIe}_{\mathcal{G}}(\sigma, pc_l, id, v_1, \cdots, v_n) = v}{\mathsf{ehAPIe}(\mathcal{G}, \sigma, pc_l, id, v_1, \cdots, v_n) = v} \text{ EHAPI}$$

$$\boxed{v_1^{\mathsf{MF}} \text{ bop } v_2^{\mathsf{MF}}}$$

$$\frac{\begin{array}{c}v_1 = \langle _ | _ \rangle \quad \text{or} \quad v_2 = \langle _ | _ \rangle \qquad v_H = (\mathsf{getFacetV}(v_1, H)) \text{ bop } (\mathsf{getFacetV}(v_2, H)) \\ v_L = (\mathsf{getFacetV}(v_1, L)) \text{ bop } (\mathsf{getFacetV}(v_2, L))\end{array}}{v_1 \text{ bop } v_2 = \mathsf{setFacetV}(v_H, v_L)} \text{ BOP-FACET}$$

$$\boxed{v_1^{\mathsf{TT}} \text{ bop } v_2^{\mathsf{TT}}}$$

$$\frac{}{(v, l) \text{ bop } (v', l') = (v \text{ bop } v', l \sqcup l')} \text{ BOP-LABEL}$$

Figure C.7: Expression semantics

EH expression APIs, binary operations The ehAPIe rules shown in Figure C.7 handle any special cases at the interface between different enforcement mechanisms. Since the expression semantics convert the types of the attributes to the format expected by $\mathcal{G}$, we only need to consider the case where the no-context *pc* may be used. We use EHAPI-NC to split the context and compute both the H and L cases to make the result as accurate as possible. We combine the results using createFct. Otherwise, the EH storage functions are called directly. The rules for the tree-structured EH storage are similar except that the EH nodes are passed by reference rather than passing an *id*.

Binary operations are straightforward, except when they involve facets or tainted values. The relevant rules are shown in Figure C.7. Recall from our expression semantics that values in sub-expressions will be converted to the same format (i.e., binary operations will not mix faceted values and labeled values). To perform a binary operation involving a faceted value (rule BOP-FACET), split the faceted value(s) on the H and L facets, separately, then combine the results into a faceted value. To perform a binary operation on two labeled (tainted) values (rule BOP-LABEL), perform the operation on the values and assign the result the join of the labels on the original values.

Variable lookup Variable lookup rules are shown in Figure C.8. Variable lookups for MF and FS stores are the same. If the lookup happens under a "standard" (i.e., L or H) *pc*, then we use getFacetV to get the appropriate facet from that value in the store (rules MF-VAR and FS-VAR). Uninitialized L or H facets will be $\cdot$. getFacetV returns dv in this case. If the *pc* is $\cdot$, we perform the lookup in both the L and H context and create a faceted value with the results using setFacetV (rules MF-VAR-F and FS-VAR-F).

The rules for TT and TS are mostly straightforward. If the variable is in the store, we simply return the value (rules TT-VAR and TS-VAR). If not, we always return a default value with label H, regardless of the *pc* (rules TT-VAR-DV and TS-VAR-DV). If we labeled the default value with the *pc*, we would leak something to the attacker if the variable did exist in the store and was tainted (because (dv, L) is distinguishable from (v, H)). To hide the possibility of the variable holding a secret, we always taint the default value.

C.2.5 EH Storage semantics

Unstructured EH Storage command semantics Here we describe the commands for interacting with the event handler storage. The rules are shown in Figure C.9 Note: these rules are meant to be general enough to apply to any enforcement mechanism, which is why labOf appears in most rules. ASSIGN-D updates the attribute for a node with id *id*. CREATEELEM adds an empty node with id *id* and assigns the value determined by expression e. ADDEH registers a new event handler (*eh*) to a node given by *id*. TRIGGER triggers the event handlers for Ev associated with a node *id* with parameter e. Each of these rules uses a helper function with the type of enforcement ($G \downarrow_{EH}$) as one of the parameters.

$$\boxed{\mathsf{var}_{\mathcal{V}}(\sigma^{\mathcal{V}}, pc, x) = v}$$

$$\frac{}{\mathsf{var}_{\mathsf{SME}}(\sigma^{\mathsf{std}}, pc_l, x) = \sigma^{\mathsf{std}}(x)}\ \textsc{sme-var} \qquad \frac{x \notin \sigma^{\mathsf{std}}}{\mathsf{var}_{\mathsf{SME}}(\sigma^{\mathsf{std}}, pc_l, x) = \mathsf{dv}}\ \textsc{sme-var-dv}$$

$$\frac{\sigma_l = \mathsf{getStore}(\sigma, pc_l)}{\mathsf{var}_{\mathsf{SMS}}(\sigma, pc_l, x) = \sigma_l(x)}\ \textsc{sms-var} \qquad \frac{\sigma_l = \mathsf{getStore}(\sigma, pc_l) \qquad x \notin \sigma_l}{\mathsf{var}_{\mathsf{SMS}}(\sigma, pc_l, x) = \mathsf{dv}}\ \textsc{sms-var-dv}$$

$$\frac{v = \mathsf{getFacetV}(\sigma(x), pc_l)}{\mathsf{var}_{\mathsf{MF}}(\sigma, pc_l, x) = v}\ \textsc{mf-var} \qquad \frac{\begin{array}{c} x \in \sigma \\ v_H = \mathsf{var}_{\mathsf{MF}}(\sigma, H, x) \qquad v_L = \mathsf{var}_{\mathsf{MF}}(\sigma, L, x) \end{array}}{\mathsf{var}_{\mathsf{MF}}(\sigma, \cdot, x) = \mathsf{setFacetV}(v_H, v_L)}\ \textsc{mf-var-f}$$

$$\frac{x \notin \sigma}{\mathsf{var}_{\mathsf{MF}}(\sigma, pc, x) = \mathsf{dv}}\ \textsc{mf-var-dv}$$

$$\frac{v = \mathsf{getFacetV}(\sigma(x), pc_l)}{\mathsf{var}_{\mathsf{FS}}(\sigma, pc_l, x) = v}\ \textsc{fs-var} \qquad \frac{x \in \sigma \qquad v_H = \mathsf{var}_{\mathsf{FS}}(\sigma, H, x) \qquad v_L = \mathsf{var}_{\mathsf{FS}}(\sigma, L, x)}{\mathsf{var}_{\mathsf{FS}}(\sigma, \cdot, x) = \mathsf{setFacetV}(v_H, v_L)}\ \textsc{fs-var-f}$$

$$\frac{x \notin \sigma}{\mathsf{var}_{\mathsf{FS}}(\sigma, pc, x) = \mathsf{dv}}\ \textsc{fs-var-dv}$$

$$\frac{}{\mathsf{var}_{\mathsf{TT}}(\sigma, pc_l, x) = \sigma(x)}\ \textsc{tt-var} \qquad \frac{x \notin \sigma}{\mathsf{var}_{\mathsf{TT}}(\sigma, pc_l, x) = (\mathsf{dv}, H)}\ \textsc{tt-var-dv}$$

$$\frac{}{\mathsf{var}_{\mathsf{TS}}(\sigma, pc, x) = \sigma(x)}\ \textsc{ts-var} \qquad \frac{x \notin \sigma}{\mathsf{var}_{\mathsf{TS}}(\sigma, pc, x) = (\mathsf{dv}, H)}\ \textsc{ts-var-dv}$$

Figure C.8: Variable lookup rules

Tree-structured EH Storage command semantics The rules are mostly similar to the unstructured EH storage semantics. The exceptions are shown in Figure C.10. CREATECHILD adds an empty node as the left-most child of the node at location a_p. The new node has id *id* and value given by expression e. The parent of the new node is located at a_p. If a_p is NULL, the store is unchanged. CREATESIBLING adds an empty node as the right-hand sibling of the node at location a_s. The new node has id *id* and value given by expression e. The parent of the new node is the parent of a_s. If a_s or the parent of a_s is NULL, the store is unchanged.

Unstructured EH Storage command semantics (helper functions) The rules in this section connect the framework to the specific enforcement mechanisms protecting the shared EH storage. For each helper function, there is a set of rules for each enforcement mechanism (SMS, FS, TS).

$$\boxed{G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1^{\mathcal{V}}, c_1^{\mathsf{std}} \xrightarrow{\alpha}_{pc} \sigma_2^G, \sigma_2^{\mathcal{V}}, c_2, E}$$

$$\frac{G, \mathcal{V}, \sigma_1^G, \sigma_1^{\mathcal{V}} \vdash e \Downarrow_{pc}^{G\downarrow EH} v \qquad \mathsf{assign}_{G\downarrow_{EH}}(\sigma_1^G, pc, id, v) = \sigma_2^G}{G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1^{\mathcal{V}}, id := e \xrightarrow{\bullet}_{pc} \sigma_2^G, \sigma_1^{\mathcal{V}}, \mathsf{skip}, \cdot} \textsc{assign-D}$$

$$\frac{G, \mathcal{V}, \sigma_1^G, \sigma_1^{\mathcal{V}} \vdash e \Downarrow_{pc}^{G\downarrow EH} v \qquad \sigma_2^G = \mathsf{createElem}_{G\downarrow_{EH}}(\sigma_1^G, pc, id, v)}{G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1^{\mathcal{V}}, \mathsf{create}(id, e) \xrightarrow{\bullet}_{pc} \sigma_2^G, \sigma_1^{\mathcal{V}}, \mathsf{skip}, \cdot} \textsc{createElem}$$

$$\frac{\sigma_2^G = \mathsf{registerEH}_{G\downarrow_{EH}}(\sigma_1^G, pc, id, eh)}{G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1^{\mathcal{V}}, \mathsf{register}(id, eh) \xrightarrow{\bullet}_{pc} \sigma_2^G, \sigma_1^{\mathcal{V}}, \mathsf{skip}, \cdot} \textsc{addEH}$$

$$\frac{G, \mathcal{V}, \sigma^G, \sigma^{\mathcal{V}} \vdash e \Downarrow_{pc}^{G\downarrow EH} v \qquad E = \mathsf{triggerEH}_{G\downarrow_{EH}}(\sigma^G, pc, id, \mathsf{Ev}, v)}{G, \mathcal{V}, d \Vdash \sigma^G, \sigma^{\mathcal{V}}, \mathsf{trigger}(id, \mathsf{Ev}, e) \xrightarrow{\bullet}_{pc} \sigma^G, \sigma^{\mathcal{V}}, \mathsf{skip}, E} \textsc{trigger}$$

Figure C.9: Shared Unstructured EH storage command semantics

$$\frac{G, \mathcal{V}, \sigma_1^G, \sigma_1^{\mathcal{V}} \vdash e \Downarrow_{pc}^{G\downarrow EH} v \qquad \sigma_2^G = \mathsf{createChild}_{G\downarrow_{EH}}(\sigma_1^G, pc, id, a_p, v)}{G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1^{\mathcal{V}}, \mathsf{createChild}(id, a_p, e) \xrightarrow{\bullet}_{pc} \sigma_2^G, \sigma_1^{\mathcal{V}}, \mathsf{skip}, \cdot} \textsc{createChild}$$

$$\frac{G, \mathcal{V}, \sigma_1^G, \sigma_1^{\mathcal{V}} \vdash e \Downarrow_{pc}^{G\downarrow EH} v \qquad \sigma_2^G = \mathsf{createSibling}_{G\downarrow_{EH}}(\sigma_1^G, pc, id, a_s, v)}{G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1^{\mathcal{V}}, \mathsf{createSibling}(id, a_s, e) \xrightarrow{\bullet}_{pc} \sigma_2^G, \sigma_1^{\mathcal{V}}, \mathsf{skip}, \cdot} \textsc{createSibling}$$

Figure C.10: Shared Tree-structured EH storage command semantics

Node lookup. Figure C.11 shows the rules for looking up a node in the unstructured EH storage. For the SMS store, this is straightforward. Recall that for the faceted store, nodes may contain faceted values if they depend on a secret. The FS store also checks whether the node is initialized in the given context by checking if the attribute in the node is initialized (i.e., whether $\phi.v \downarrow_{pc} \neq \cdot$). If a node with matching *id* does not exist in the store, or if the node has not been initialized in the given context, rule FS-LOOKUP-S returns NULL. If the $pc = \cdot$, we split the execution and perform the lookup in both the L and H contexts. We create a facet with the results using setFacetN. The rules for looking up a node in the TS store are mostly straightforward. We always label NULL as secret in this case, regardless of the pc because we want it to be indistinguishable from a secret node (in case one exists in an equivalent store).

Node attribute update. Figure C.12 shows the rules for updating the attribute of a node. The SMS rules are straightforward. SMS-ASSIGNEH-NC handles the case where the $pc = \cdot$. Here, we split the context and make assignments to the L and H stores separately. The rules for FS attribute updates directly replace the

$\boxed{\mathsf{lookup}_G(\sigma, pc, id) = \phi}$

$$\frac{\sigma' = \mathsf{getStore}_{\mathsf{SMS}}(\sigma, pc_l) \qquad \sigma'(id) = \phi}{\mathsf{lookup}_{\mathsf{SMS}}(\sigma, pc_l, id) = \phi}\ \textsc{sms-lookup} \qquad \frac{\sigma' = \mathsf{getStore}_{\mathsf{SMS}}(\sigma, pc_l) \qquad id \notin \sigma'}{\mathsf{lookup}_{\mathsf{SMS}}(\sigma, pc_l, id) = \mathsf{NULL}}\ \textsc{sms-lookup-s}$$

$$\frac{\sigma(id) = \phi \qquad \phi.v \downarrow_{pc_l} \neq \cdot}{\mathsf{lookup}_{\mathsf{FS}}(\sigma, pc_l, id) = \phi}\ \textsc{fs-lookup} \qquad \frac{id \notin \sigma \qquad \text{or} \qquad \sigma(id).v \downarrow_{pc_l} = \cdot}{\mathsf{lookup}_{\mathsf{FS}}(\sigma, pc_l, id) = \mathsf{NULL}}\ \textsc{fs-lookup-s}$$

$$\frac{\phi_H = \mathsf{lookup}_{\mathsf{FS}}(\sigma, H, id) \qquad \phi_L = \mathsf{lookup}_{\mathsf{FS}}(\sigma, L, id)}{\mathsf{lookup}_{\mathsf{FS}}(\sigma, \cdot, id) = \mathsf{setFacetN}(\phi_H, \phi_L)}\ \textsc{fs-lookup-f}$$

$$\frac{\sigma(id) = \phi}{\mathsf{lookup}_{\mathsf{TS}}(\sigma, pc, id) = \phi}\ \textsc{ts-lookup} \qquad \frac{id \notin \sigma}{\mathsf{lookup}_{\mathsf{TS}}(\sigma, pc, id) = (\mathsf{NULL}, H)}\ \textsc{ts-lookup-s}$$

Figure C.11: Rules for looking up a node in the unstructured EH storage

attribute in the store if the $pc = \cdot$ and a node with matching id is found in the store (rule FS-ASSIGNEH) and do not change the store if no matching node is found or if the node is not initialized in the given context (rule FS-ASSIGNEH-S). For a standard pc (i.e., it is L or H), we update the attribute by using updateFacet (rule FS-ASSIGNEH-UPD). If the node is faceted (i.e., $\mathsf{lookup}_{\mathsf{FS}}$ returns a faceted value), then the execution splits and performs the assignment in both the L and H context (rule FS-ASSIGNEH-NC). The rules for the TS store are straightforward.

Triggering an event handler. Figure C.13 contains rules for triggering event handlers. For SMS, SMS-TRIGGEREH triggers the event in the given context (either L or H). If a node with matching id does not exist, SMS-TRIGGEREH-S does not trigger any events. When the $pc = \cdot$, SMS-TRIGGEREH-NC splits the context to look up the event in both copies of the store. If multiple events are generated, they are merged using mergeEvs (Section C.1.2). The rules for FS and TS are similar, except that TS runs the new event in the context given by the pc joined with the label on the node joined with the label on the argument to the event (rather than just pc like the others).

Creating a new node. Figure C.14 contains rules for creating a new EH node. To create a new node, we first check if a node with id exists. If it does not, we create an empty node with the given id. If it does exist, we update the node with the given attribute using assign. The rules for SMS and FS are straightforward.

Updating a node in TS is more complex: First, we check the current label on the node (given by l').

$\boxed{\mathsf{assign}_G(\sigma, pc, id, v) = \sigma'}$

$$\frac{\sigma = \mathsf{getStore}_{\mathsf{SMS}}(\sigma^G \downarrow_{EH}, pc_l) \qquad (v', M) = \sigma(id) \qquad \sigma' = \sigma[id \mapsto (v, M)]}{\mathsf{assign}_{\mathsf{SMS}}(\sigma^G, pc_l, id, v) = \mathsf{setStore}_{\mathsf{SMS}}(\sigma^G, pc_l, \sigma')} \text{ SMS-ASSIGNEH}$$

$$\frac{\sigma = \mathsf{getStore}_{\mathsf{SMS}}(\sigma^G \downarrow_{EH}, pc_l) \qquad id \notin \sigma}{\mathsf{assign}_{\mathsf{SMS}}(\sigma^G, pc_l, id, v) = \sigma^G} \text{ SMS-ASSIGNEH-S}$$

$$\frac{\sigma_1^G = \mathsf{assign}_{\mathsf{SMS}}(\sigma^G, H, id, \mathsf{getFacet}(v, H)) \qquad \sigma_2^G = \mathsf{assign}_{\mathsf{SMS}}(\sigma_1^G, L, id, \mathsf{getFacet}(v, L))}{\mathsf{assign}_{\mathsf{SMS}}(\sigma^G, \cdot, id, v) = \sigma_2^G} \text{ SMS-ASSIGNEH-NC}$$

$$\frac{(v', M) = \mathsf{lookup}_{\mathsf{FS}}(\sigma, \cdot, id)}{\mathsf{assign}_{\mathsf{FS}}(\sigma, \cdot, id, v) = \sigma[id \mapsto (v, M)]} \text{ FS-ASSIGNEH} \qquad \frac{\mathsf{NULL} = \mathsf{lookup}_{\mathsf{FS}}(\sigma, pc, id)}{\mathsf{assign}_{\mathsf{FS}}(\sigma, pc, id, v) = \sigma} \text{ FS-ASSIGNEH-S}$$

$$\frac{\begin{array}{c}(v', M) = \mathsf{lookup}_{\mathsf{FS}}(\sigma, pc_l, id)\\ v'' = \mathsf{updateFacet}(v', v, pc_l)\end{array}}{\mathsf{assign}_{\mathsf{FS}}(\sigma, pc_l, id, v) = \sigma[id \mapsto (v'', M)]} \text{ FS-ASSIGNEH-UPD}$$

$$\frac{\begin{array}{c}\phi = \mathsf{lookup}_{\mathsf{FS}}(\sigma, \cdot, id) = \langle _ | _ \rangle\\ \sigma' = \mathsf{assign}_{\mathsf{FS}}(\sigma, H, id, \mathsf{getFacet}(v, H)) \qquad \sigma'' = \mathsf{assign}_{\mathsf{FS}}(\sigma', L, id, \mathsf{getFacet}(v, L))\end{array}}{\mathsf{assign}_{\mathsf{FS}}(\sigma, \cdot, id, v) = \sigma''} \text{ FS-ASSIGNEH-NC}$$

$$\frac{(v', M, l') = \sigma(id)}{\mathsf{assign}_{\mathsf{TS}}(\sigma, pc, id, (v, l)) = \sigma[id \mapsto (id, (v, l \sqcup pc \sqcup l'), M, l')]} \text{ TS-ASSIGNEH}$$

$$\frac{id \notin \sigma}{\mathsf{assign}_{\mathsf{TS}}(\sigma, pc, id, (v, l)) = \sigma} \text{ TS-ASSIGNEH-S}$$

Figure C.12: Rules for updating the attribute of a node in the unstructured EH storage

If the node is visible in the current context (i.e., $l' \sqsubseteq pc$), TS-CREATE-U1 updates the attribute. The new attribute has the label given by joining the label on the node, pc, and the label on the attribute. If the node is not visible in the current context (i.e., $l' \not\sqsubseteq pc$), TS-CREATE-U2 updates the attribute and the label on the node. The new attribute has the label given by joining the pc and the label on the attribute. The new label on the nodes is the pc. We update the label on the node because leaving the old label on the node would mean the node is still not visible in the current context, despite the node being created. This would leak to the attacker that a secret node with the given id existed. Finally, if the $pc = \cdot$ TS-CREATE-NC creates the node in the L context. This is different from the other stores (which split the context and create the node twice) because the given attribute is already formatted as a labeled value. If we were to create the node twice, it would create the node, then overwrite the attribute with the same value anyway. Creating the node just once in the L context is more efficient and does not leak anything because the existence of the node is not secret (since the $pc = \cdot$).

$\boxed{\mathsf{triggerEH}_{\mathcal{G}}(\sigma, pc, id, \mathsf{Ev}, v) = E}$

$$\frac{\mathsf{lookup}_{\mathsf{SMS}}(\sigma, pc_l, id) = \phi \neq \mathsf{NULL}}{\mathsf{triggerEH}_{\mathsf{SMS}}(\sigma, pc_l id, \mathsf{Ev}, v) = (id.\mathsf{Ev}(v), pc_l)} \text{ SMS-TRIGGEREH}$$

$$\frac{\phi = \mathsf{lookup}_{\mathsf{SMS}}(\sigma, pc_l, id) = \mathsf{NULL}}{\mathsf{triggerEH}_{\mathsf{SMS}}(\sigma, pc_l, id, \mathsf{Ev}, v) = \cdot} \text{ SMS-TRIGGEREH-S}$$

$$\frac{\begin{array}{c} E_H = \mathsf{triggerEH}_{\mathsf{SMS}}(\sigma, H, id, \mathsf{Ev}, \mathsf{getFacet}(v, H)) \\ E_L = \mathsf{triggerEH}_{\mathsf{SMS}}(\sigma, L, id, \mathsf{Ev}, \mathsf{getFacet}(v, L)) \end{array}}{\mathsf{triggerEH}_{\mathsf{SMS}}(\sigma, \cdot, id, \mathsf{Ev}, v) = \mathsf{mergeEvs}(E_H, E_L)} \text{ SMS-TRIGGEREH-NC}$$

$$\frac{\phi = \mathsf{lookup}_{\mathsf{FS}}(\sigma, pc, id) \neq \langle _|_ \rangle \neq \mathsf{NULL}}{\mathsf{triggerEH}_{\mathsf{FS}}(\sigma, pc, id, \mathsf{Ev}, v^{\mathsf{std}}) = (id.\mathsf{Ev}(v), pc)} \text{ FS-TRIGGEREH}$$

$$\frac{\begin{array}{c} \langle _|_ \rangle = \mathsf{lookup}_{\mathsf{FS}}(\sigma, pc, id) \vee v = \langle _|_ \rangle \\ E_H = \mathsf{triggerEH}_{\mathsf{FS}}(\sigma, H, id, \mathsf{Ev}, \mathsf{getFacet}(v, H)) \qquad E_L = \mathsf{triggerEH}_{\mathsf{FS}}(\sigma, L, id, \mathsf{Ev}, \mathsf{getFacet}(v, L)) \end{array}}{\mathsf{triggerEH}_{\mathsf{FS}}(\sigma, \cdot, id, \mathsf{Ev}, v) = \mathsf{mergeEvs}(E_H, E_L)} \text{ FS-TRIGGEREH-NC}$$

$$\frac{\mathsf{lookup}_{\mathsf{FS}}(\sigma, pc, id) = \mathsf{NULL}}{\mathsf{triggerEH}_{\mathsf{FS}}(\sigma, pc, id, \mathsf{Ev}, v^{\mathsf{std}}) = \cdot} \text{ FS-TRIGGEREH-S}$$

$$\frac{\mathsf{lookup}_{\mathcal{G}}(\sigma, pc_l, id) = (_, _, l_\phi) \qquad l = \mathsf{labOf}(v, pc_l)}{\mathsf{triggerEH}_{\mathcal{G}}(\sigma, pc_l, id, \mathsf{Ev}, v) = (id.\mathsf{Ev}(v), pc_l \sqcup l_\phi \sqcup l)} \text{ TS-TRIGGEREH}$$

$$\frac{E_H = \mathsf{triggerEH}_{\mathcal{G}}(\sigma, H, id, \mathsf{Ev}, \mathsf{getFacetV}(v, H)) \qquad E_L = \mathsf{triggerEH}_{\mathcal{G}}(\sigma, L, id, \mathsf{Ev}, \mathsf{getFacetV}(v, L))}{\mathsf{triggerEH}_{\mathcal{G}}(\sigma, \cdot, id, \mathsf{Ev}, v) = \mathsf{mergeEvs}(E_H, E_L)} \text{ TS-TRIGGEREH-NC}$$

$$\frac{\phi = \mathsf{lookup}_{\mathcal{G}}(\sigma, pc_l, id) = (\mathsf{NULL}, _)}{\mathsf{triggerEH}_{\mathcal{G}}(\sigma, pc_l, id, \mathsf{Ev}, v) = \cdot} \text{ TS-TRIGGEREH-S}$$

Figure C.13: Rules for triggering an event in the unstructured EH storage

Registering a new event handler. Figure C.15 contains rules for registering a new event handler. To register a new event handler, we first look up the node with the given *id*. If the node exists, we add the event handler, otherwise we do not change the store. If the $pc = \cdot$ we split the context and add the event handler in both the L and H context. The rules for the SMS and TS stores are straightforward. For the FS store, if the node with matching *id* is faceted, REGISTEREH-NC splits the execution to add the event handler to both nodes in the facet.

Tree-structured EH Storage command semantics (helper functions) The rules in this section connect the framework to the specific enforcement mechanisms protecting the shared EH storage. For each helper function, there is a set of rules for each enforcement mechanism (SMS, FS). Note that because we prove the unstructured TS store only satisfies weak secrecy, we do not formalize the tree-structured TS store.

$\boxed{\mathsf{createElem}_{\mathcal{G}}(\sigma_1^G, pc, id, v) = \sigma_2^G}$

$$\frac{\mathsf{lookup}_{\mathsf{SMS}}(\sigma, pc_l, id) = \mathsf{NULL} \qquad \phi = (v^{\mathsf{std}}, \cdot) \qquad \sigma' = \mathsf{getStore}(\sigma, pc_l)}{\mathsf{createElem}_{\mathsf{SMS}}(\sigma, pc_l, id, v) = \mathsf{setStore}(\sigma, pc_l, \sigma'[id \mapsto \phi])} \text{ SMS-CREATE}$$

$$\frac{\mathsf{lookup}_{\mathsf{SMS}}(\sigma, pc, id) = \phi \neq \mathsf{NULL} \qquad \sigma' = \mathsf{assign}_{\mathsf{SMS}}(\sigma, pc, id, v)}{\mathsf{createElem}_{\mathsf{SMS}}(\sigma, pc_l, id, v) = \sigma'} \text{ SMS-CREATE-U}$$

$$\frac{\sigma' = \mathsf{createElem}_{\mathsf{SMS}}(\sigma, H, id, v) \qquad \sigma'' = \mathsf{createElem}_{\mathsf{SMS}}(\sigma', L, id, v)}{\mathsf{createElem}_{\mathsf{SMS}}(\sigma, \cdot, id, v) = \sigma''} \text{ SMS-CREATE-NC}$$

$$\frac{\mathsf{lookup}_{\mathsf{FS}}(\sigma, \cdot, id) = \mathsf{NULL} \qquad v' = \mathsf{getFacetV}(v, pc) \qquad \phi = (\mathsf{createFacet}(v', pc), \cdot)}{\mathsf{createElem}_{\mathsf{FS}}(\sigma, pc, id, v) = \sigma[id \mapsto \phi]} \text{ FS-CREATE}$$

$$\frac{\mathsf{lookup}_{\mathsf{FS}}(\sigma, \cdot, id) = \phi \neq \mathsf{NULL} \qquad \sigma' = \mathsf{assign}_{\mathsf{FS}}(\sigma, pc, id, v)}{\mathsf{createElem}_{\mathsf{FS}}(\sigma, pc, id, v) = \sigma'} \text{ FS-CREATE-U}$$

$$\frac{\mathsf{lookup}_{\mathsf{TS}}(\sigma, pc_l, id) = (\mathsf{NULL}, _) \qquad \phi = (v, \cdot, pc_l)}{\mathsf{createElem}_{\mathsf{TS}}(\sigma, pc_l, id, v) = \sigma[id \mapsto \phi]} \text{ TS-CREATE}$$

$$\frac{\mathsf{lookup}_{\mathsf{TS}}(\sigma, pc, id) = (v', M, l') \qquad l' \sqsubseteq pc \qquad \sigma' = \sigma[id \mapsto ((v, l \sqcup pc \sqcup l'), M, l')]}{\mathsf{createElem}_{\mathsf{TS}}(\sigma, pc, id, (v, l)) = \sigma'} \text{ TS-CREATE-U1}$$

$$\frac{\mathsf{lookup}_{\mathsf{TS}}(\sigma, pc, id) = (v', M, l') \qquad l' \not\sqsubseteq pc \qquad \sigma' = \sigma[id \mapsto ((v, l \sqcup pc), M, pc)]}{\mathsf{createElem}_{\mathsf{TS}}(\sigma, pc, id, (v, l)) = \sigma'} \text{ TS-CREATE-U2}$$

$$\frac{\sigma' = \mathsf{createElem}_{\mathsf{TS}}(\sigma, L, id, v)}{\mathsf{createElem}_{\mathsf{TS}}(\sigma, \cdot, id, v) = \sigma'} \text{ TS-CREATE-NC}$$

Figure C.14: Rules for creating a new node in the unstructured EH storage

Node address lookup. Figure C.16 contains rules for looking up the address of a node. lookupA looks up the address of a node by traversing the EH tree store recursively. We show only the rules for FS. The biggest difference between SMS and FS is that the structure of the nodes is different. FS-LOOKUPA handles the case where the current node matches the given *id*. We also have to check that the node is visible in the given context (i.e., $\phi.v \downarrow_{pc_l} \neq \cdot$). If the node we're currently checking does not match, but it is visible in the given context, FS-LOOKUPA-R1 adds the node's children (that are visible in the given context, $A \downarrow_{pc_l}$) to the list of nodes to check. If the current node does not match the given *id*, and is not visible, FS-LOOKUPA-R2 checks the rest of the nodes in the list but does not add the children of the current node. Traversing the SMS tree is a little easier because we know all of the nodes are visible as long as we are interacting with

$\boxed{\mathsf{registerEH}_{\mathcal{G}}(\sigma_1^{\mathcal{G}}, pc, id, eh) = \sigma_2^{\mathcal{G}}}$

$$\frac{\begin{array}{c}(v, M) = \mathsf{lookup}_{\mathsf{SMS}}(\sigma, pc_l, id)\\ eh = \mathsf{onEv}(x)\{c\} \qquad M' = M[\mathsf{Ev} \mapsto M(\mathsf{Ev}) \cup \{(eh, pc_l)\}] \qquad \sigma' = \mathsf{getStore}(\sigma, pc_l)\end{array}}{\mathsf{registerEH}_{\mathsf{SMS}}(\sigma, pc_l, id, eh) = \mathsf{setStore}(\sigma, pc_l, \sigma'[id \mapsto (v, M')])}\ \textsc{sms-register}\mathrm{EH}$$

$$\frac{\mathsf{NULL} = \mathsf{lookup}_{\mathsf{SMS}}(\sigma, pc_l, id)}{\mathsf{registerEH}_{\mathsf{SMS}}(\sigma, pc_l, id, eh) = \sigma}\ \textsc{sms-register}\mathrm{EH}\textsc{-s}$$

$$\frac{\sigma' = \mathsf{registerEH}_{\mathsf{SMS}}(\sigma, H, id, eh) \qquad \sigma'' = \mathsf{registerEH}_{\mathsf{SMS}}(\sigma', L, id, eh)}{\mathsf{registerEH}_{\mathsf{SMS}}(\sigma, \cdot, id, eh) = \sigma''}\ \textsc{sms-register}\mathrm{EH}\textsc{-nc}$$

$$\frac{(v, M) = \mathsf{lookup}_{\mathsf{FS}}(\sigma, pc, id) \qquad eh = \mathsf{onEv}(x)\{c\} \qquad M' = M[\mathsf{Ev} \mapsto M(\mathsf{Ev}) \cup \{(eh, pc)\}]}{\mathsf{registerEH}_{\mathsf{FS}}(\sigma, pc, id, eh) = \sigma[id \mapsto (v, M')]}\ \textsc{fs-register}\mathrm{EH}$$

$$\frac{\begin{array}{c}\langle \phi_H | \phi_L \rangle = \mathsf{lookup}_{\mathsf{FS}}(\sigma, pc, id)\\ \sigma' = \mathsf{registerEH}_{\mathsf{FS}}(\sigma, H, id, eh) \qquad \sigma'' = \mathsf{registerEH}_{\mathsf{FS}}(\sigma', L, id, eh)\end{array}}{\mathsf{registerEH}_{\mathsf{FS}}(\sigma, \cdot, id, eh) = \sigma''}\ \textsc{fs-register}\mathrm{EH}\textsc{-nc}$$

$$\frac{\mathsf{NULL} = \mathsf{lookup}_{\mathsf{FS}}(\sigma, pc, id)}{\mathsf{registerEH}_{\mathsf{FS}}(\sigma, pc, id, eh) = \sigma}\ \textsc{fs-register}\mathrm{EH}\textsc{-s}$$

$$\frac{\begin{array}{c}(v, M, l) = \mathsf{lookup}_{\mathsf{TS}}(\sigma, pc_l, id)\\ eh = \mathsf{onEv}(x)\{c\} \qquad M' = M[\mathsf{Ev} \mapsto M(\mathsf{Ev}) \cup \{(eh, pc_l \sqcup l)\}]\end{array}}{\mathsf{registerEH}_{\mathsf{TS}}(\sigma, pc_l, id, eh) = \sigma[id \mapsto (v, M', l)]}\ \textsc{ts-register}\mathrm{EH}$$

$$\frac{\sigma' = \mathsf{registerEH}_{\mathsf{TS}}(\sigma, H, id, eh) \qquad \sigma'' = \mathsf{registerEH}_{\mathsf{TS}}(\sigma', L, id, eh)}{\mathsf{registerEH}_{\mathsf{TS}}(\sigma, \cdot, id, eh) = \sigma''}\ \textsc{ts-register}\mathrm{EH}\textsc{-nc}$$

$$\frac{\phi = \mathsf{lookup}_{\mathsf{TS}}(\sigma, pc_l, id) \qquad \mathsf{valOf}(\phi) = \mathsf{NULL}}{\mathsf{registerEH}_{\mathsf{TS}}(\sigma, pc_l, id, eh) = \sigma}\ \textsc{ts-register}\mathrm{EH}\textsc{-s}$$

Figure C.15: Rules for registering a new event handler in the unstructured EH storage

$\boxed{\mathsf{lookupA}_{\mathcal{G}}(\sigma^{\mathcal{G}}, pc, id, A) = a}$

$$\frac{\sigma(a).id = id \qquad \phi.v \downarrow_{pc_l} \neq \cdot}{\mathsf{lookupA}_{\mathsf{FS}}(\sigma, pc_l, id, (a :: A)) = a}\ \textsc{fs-lookup}\mathrm{A}$$

$$\frac{\begin{array}{c}\sigma(a).id \neq id \qquad \sigma(a).v \downarrow_{pc_l} \neq \cdot\\ \mathsf{lookupA}_{\mathsf{FS}}(\sigma, pc_l, id, (A :: \sigma(a).A \downarrow_{pc_l})) = a'\end{array}}{\mathsf{lookupA}_{\mathsf{FS}}(\sigma, pc_l, id, (a :: A)) = a'}\ \textsc{fs-lkup}\mathrm{A}\textsc{-r1}$$

$$\frac{\begin{array}{c}\sigma(a).v \downarrow_{pc_l} = \cdot\\ \mathsf{lookupA}_{\mathsf{FS}}(\sigma, pc_l, id, A) = a'\end{array}}{\mathsf{lookupA}_{\mathsf{FS}}(\sigma, pc_l, id, (a :: A)) = a'}\ \textsc{fs-lookup}\mathrm{A}\textsc{-r2}$$

$$\frac{a_H = \mathsf{lookupA}_{\mathsf{FS}}(\sigma, H, id, A \downarrow_H) \qquad a_L = \mathsf{lookupA}_{\mathsf{FS}}(\sigma, L, id, A \downarrow_L)}{\mathsf{lookupA}_{\mathsf{FS}}(\sigma, \cdot, id, A) = \mathsf{setFacetA}(a_H, a_L)}\ \textsc{fs-lookup}\mathrm{A}\textsc{-f}$$

$$\frac{}{\mathsf{lookupA}_{\mathsf{FS}}(\sigma, pc_l, id, \cdot) = \mathsf{NULL}}\ \textsc{fs-lookup}\mathrm{A}\textsc{-s}$$

Figure C.16: Rules for looking up the address of a node in the tree-structured EH storage

$\boxed{\mathsf{lookup}_{\mathcal{G}}(\sigma^{\mathcal{G}}, pc_l, id, A) = \phi}$

$$\frac{\sigma' = \mathsf{getStore}(\sigma, pc_l) \qquad \mathsf{lookupA}_{\mathsf{SMS}}(\sigma, pc_l, id, a^{\mathsf{rt}}) = a \neq \mathsf{NULL} \qquad \sigma'(a) = \phi}{\mathsf{lookup}_{\mathsf{SMS}}(\sigma, pc_l, id) = \phi} \text{ SMS-LOOKUP}$$

$$\frac{\mathsf{lookupA}_{\mathsf{SMS}}(\sigma, pc_l, id, a^{\mathsf{rt}}) = \mathsf{NULL}}{\mathsf{lookup}_{\mathsf{SMS}}(\sigma, pc_l, id) = \mathsf{NULL}} \text{ SMS-LOOKUP-S}$$

Figure C.17: Rules for looking up a node in the tree-structured EH storage

$\boxed{\mathsf{assign}_{\mathcal{G}}(\sigma_1^{\mathcal{G}}, pc, a, v) = \sigma_2^{\mathcal{G}}}$

$$\frac{(id, v', M, a_p, A) = \sigma(a)}{\mathsf{assign}_{\mathsf{FS}}(\sigma, \cdot, a, v) = \sigma[a \mapsto (id, v, M, a_p, A)]} \text{ FS-ASSIGNEH} \qquad \frac{}{\mathsf{assign}_{\mathsf{FS}}(\sigma, pc_l, \mathsf{NULL}, v) = \sigma} \text{ FS-ASSIGNEH-S}$$

$$\frac{\mathsf{assign}_{\mathsf{FS}}(\sigma, H, a_H, \mathsf{getFacetV}(v, H)) = \sigma' \qquad \mathsf{assign}_{\mathsf{FS}}(\sigma, L, a_L, \mathsf{getFacetV}(v, L)) = \sigma''}{\mathsf{assign}_{\mathsf{FS}}(\sigma, \cdot, \langle a_H | a_L \rangle, v) = \sigma''} \text{ FS-ASSIGNEH-NC}$$

$$\frac{(id, v', M, a_p, A) = \sigma(a) \qquad v'' = \mathsf{updateFacet}(v', v, pc_l)}{\mathsf{assign}_{\mathsf{FS}}(\sigma, pc_l, a, v) = \sigma[a \mapsto (id, v'', M, a_p, A)]} \text{ FS-ASSIGNEH-UPD}$$

Figure C.18: Rules for updating the attribute of a node in the tree-structured EH storage

the appropriate copy of the store. If $pc = \cdot$, FS-LOOKUPA-F splits the execution to look for the address of the node in both the L and H contexts. The result is made into a faceted address using setFacetA. Finally, if a node with matching id does not exist, FS-LOOKUPA-S returns NULL.

Node lookup. Figure C.17 contains rules for looking up nodes by id. This function works by looking up the address of the node with matching id and then returning the node stored at that address. For the SMS store, SMS-LOOKUP uses $\mathsf{lookupA}_{\mathsf{SMS}}$ to get the address of the node. If the address is not NULL, we look up the node using the address in the copy of the store returned by getStore. If the address is NULL, SMS-LOOKUP-S returns NULL. The rules for FS are similar except they don't use getStore.

Node attribute update. Figure C.18 contains rules for updating the attribute of a node. One of the arguments to this function is the address of the node being updated, so the node does not need to be looked up. Note for FS we assume that we only have the address of valid nodes, visible at this context. To update the attribute of an FS node, when $pc = \cdot$ FS-ASSIGNEH updates the attribute directly. If the $pc \neq \cdot$ (i.e., it is L or H), FS-ASSIGNEH-UPD updates the attribute by using updateFacet which only changes the appropriate facet of the attribute. If the address is NULL, FS-ASSIGNEH-S leaves the store unchanged. If the address is

$\boxed{\mathsf{triggerEH}_{\mathcal{G}}(\sigma^{\mathcal{G}}, pc, id, \mathsf{Ev}, v) = E}$

$$\frac{\mathsf{lookupA}_{\mathsf{SMS}}(\sigma, pc_l, id, a^{\mathsf{rt}}) \neq \mathsf{NULL}}{\mathsf{triggerEH}_{\mathsf{SMS}}(\sigma, pc_l, id, \mathsf{Ev}, v) = (id.\mathsf{Ev}(v), pc)} \text{ SMS-TRIGGEREH}$$

$$\frac{\mathsf{lookupA}_{\mathsf{SMS}}(\sigma, pc_l, id, a^{\mathsf{rt}}) = \mathsf{NULL}}{\mathsf{triggerEH}_{\mathsf{SMS}}(\sigma, pc_l, id, \mathsf{Ev}, v) = \cdot} \text{ SMS-TRIGGEREH-S}$$

$$\frac{\begin{array}{c} E_H = \mathsf{triggerEH}_{\mathsf{SMS}}(\sigma, H, id, \mathsf{Ev}, \mathsf{getFacet}(v, H)) \\ E_L = \mathsf{triggerEH}_{\mathsf{SMS}}(\sigma, L, id, \mathsf{Ev}, \mathsf{getFacet}(v, L)) \end{array}}{\mathsf{triggerEH}_{\mathsf{SMS}}(\sigma, \cdot, id, \mathsf{Ev}, v) = \mathsf{mergeEvs}(E_H, E_L)} \text{ SMS-TRIGGEREH-NC}$$

Figure C.19: Rules for triggering the event handler in the tree-structured EH storage

faceted, FS-ASSIGNEH-NC splits the execution and performs the assignment to both addresses. The rules for SMS are similar except they use getStore and setStore to ensure we are interacting with the appropriate copy of the storage.

Triggering an event handler. Figure C.19 contains rules for triggering an event handler. Unlike some of the other functions, this one takes the *id* of the node, so it does need to be looked up. We trigger events in SMS and FS stores the same way, so we show only the rules for SMS. We first look up the node with matching *id*. If one exists (i.e., the address of the node is not NULL), then SMS-TRIGGEREH returns an event triggered in the given context. If a node with matching *id* does not exist (i.e., the address of the node is NULL), then SMS-TRIGGEREH-S does not trigger any events. If the $pc = \cdot$, then SMS-TRIGGEREH-NC splits the execution and we trigger the event in both the L and H copy of the store. Finally, if more than one event is triggered, we merge them using mergeEvs (Section C.1.2).

Adding a child to a node. Figure C.20 contains rules for adding a child to an existing node. One of the arguments to this function is the address of the parent node, so the node does not need to be looked up. Adding a child to a node in the SMS store is straightforward, so we focus on the rules for FS. FS-CREATEC looks up the given *id* to see if the given node already exists. Note that we use $pc = \cdot$ for the lookup to see if the node exists in any context, not just the given context. If the node does not exist, we also look up the parent node. We create a new node with given (faceted) attribute and (faceted) parent, add it to the store (at a fresh location), and also add a faceted pointer to the new node to the list of children of the given parent node. We create facets using createFacet (Section C.1.2). The new node will have a faceted attribute and parent because it only exists in the given context, if a node with the same *id* is added later in a different context, we will update the other facet of the attribute/parent appropriately. If a node with

$\boxed{\mathsf{createChild}_{\mathcal{G}}(\sigma_1^{\mathcal{G}}, pc, id, a_p, v) = \sigma_2^{\mathcal{G}}}$

$$\frac{\begin{array}{c}\mathsf{lookupA_{FS}}(\sigma, \cdot, id, a^{\mathsf{rt}}) = \mathsf{NULL} \qquad a \notin \sigma \qquad \sigma(a_p) = (id_p, v_p, M, a'_p, A) \qquad v_p \downarrow_{pc_l} \neq \cdot \\ \sigma' = \sigma[a_p \mapsto (id_p, v_p, M, a'_p, (\mathsf{createFacet}(a, pc_l) :: A))] \\ \sigma'' = \sigma'[a \mapsto (id, \mathsf{createFacet}(v, pc_l), \cdot, \mathsf{createFacet}(a_p, pc_l), \cdot)]\end{array}}{\mathsf{createChild_{FS}}(\sigma, pc_l, id, a_p, v) = \sigma''} \text{ FS-CREATEC}$$

$$\frac{\begin{array}{c}\mathsf{lookupA_{FS}}(\sigma, \cdot, id, a^{\mathsf{rt}}) = \langle a | \mathsf{NULL} \rangle \\ \sigma(a) = (id, v', M, a'_p, A) \qquad \sigma(a_p) = (id_p, v_p, M_p, a''_p, A_p) \qquad v_p \downarrow_L \neq \cdot \\ \sigma' = \sigma[a_p \mapsto (id_p, v_p, M_p, a'_p, (\mathsf{createFacet}(a, L) :: A_p))] \\ \sigma'' = \sigma'[a \mapsto (id, \mathsf{updateFacet}(v', v, L), M, \mathsf{updateFacet}(a'_p, a_p, L), A)]\end{array}}{\mathsf{createChild_{FS}}(\sigma, L, id, a_p, v) = \sigma''} \text{ FS-CREATEC-UL}$$

$$\frac{\begin{array}{c}\mathsf{lookupA_{FS}}(\sigma, \cdot, id, a^{\mathsf{rt}}) = \langle \mathsf{NULL} | a \rangle \\ \sigma(a) = (id, v', M, a'_p, A) \qquad \sigma(a_p) = (id_p, v_p, M_p, a''_p, A_p) \qquad v_p \downarrow_H \neq \cdot \\ \sigma' = \sigma[a_p \mapsto (id_p, v_p, M_p, a'_p, (\mathsf{createFacet}(a, H) :: A_p))] \\ \sigma'' = \sigma'[a \mapsto (id, \mathsf{updateFacet}(v', v, H), M, \mathsf{updateFacet}(a'_p, a_p, H), A)]\end{array}}{\mathsf{createChild_{FS}}(\sigma, H, id, a_p, v) = \sigma''} \text{ FS-CREATEC-UH}$$

$$\frac{\begin{array}{c}\sigma' = \mathsf{createChild_{FS}}(\sigma, H, id, \mathsf{getFacetA}(a_p, H), \mathsf{getFacetV}(v, H)) \\ \sigma'' = \mathsf{createChild_{FS}}(\sigma, L, id, \mathsf{getFacetA}(a_p, L), \mathsf{getFacetV}(v, L))\end{array}}{\mathsf{createChild_{FS}}(\sigma, \cdot, id, a_p, v) = \sigma''} \text{ FS-CREATEC-NC}$$

$$\frac{\mathsf{lookupA_{FS}}(\sigma, \cdot, id, a^{\mathsf{rt}}) = a \qquad a \downarrow_{pc_l} \neq \mathsf{NULL} \vee \sigma(a_p).v \downarrow_{pc_l} = \cdot}{\mathsf{createChild_{FS}}(\sigma, pc_l, id, a_p, v) = \sigma} \text{ FS-CREATEC-S1}$$

$$\frac{}{\mathsf{createChild_{FS}}(\sigma, pc_l, id, \mathsf{NULL}, v) = \sigma} \text{ FS-CREATEC-S2}$$

Figure C.20: Rules for adding a child to a node in the tree-structured EH storage

the given *id* already exists in the given context, the parent node is not visible in the given context, or if the pointer to the parent node is NULL, FS-CREATEC-S1 or FS-CREATEC-S2 (respectively) leaves the store unchanged. If $pc = \cdot$, FS-CREATEC-NC splits the execution to add the node in both contexts.

Recall that we use the $\cdot$ context to see if a node with the given *id* already exists in the store. We do this to see if the node already exists in the other context so that we can initialize the node in the new context. FS-CREATEC-UL handles the case where the node already exists in the H context and is being added in the L context (and respectively for FS-CREATEC-UH to add the node in the H context). In these cases, we get a faceted node when we look up the node by *id* and the facet for the context we want to add the node in is NULL (meaning that the node does not exist in that context yet). Instead of adding a new node, we update the existing node. We use updateFacet to update the attribute and pointer to the parent.

$$\boxed{\mathsf{createSibling}_{\mathcal{G}}(\sigma_1^{\mathcal{G}}, pc, id, a_s, v) = \sigma_2^{\mathcal{G}}}$$

$$\frac{\begin{array}{c}\mathsf{lookupA}_{\mathsf{FS}}(\sigma, \cdot, id, a^{\mathsf{rt}}) = \mathsf{NULL} \quad a \notin \sigma \quad \sigma(a_s).v \downarrow_{pc_l} \neq \cdot \\ \sigma(a_s).a_p \downarrow_{pc_l} = a_p \quad \sigma(a_p) = (id_p, v', M, a'_p, (A :: a'_s :: A')) \quad a'_s \downarrow_{pc_l} = a_s \quad v' \downarrow_{pc_l} \neq \cdot \\ \sigma' = \sigma[a_p \mapsto (id_p, v', M, a'_p, (A :: a'_s :: \mathsf{createFacet}(a, pc_l) :: A'))] \\ \sigma'' = \sigma'[a \mapsto (id, \mathsf{createFacet}(v, pc_l), \cdot, \mathsf{createFacet}(a_p, pc_l), \cdot)]\end{array}}{\mathsf{createSibling}_{\mathsf{FS}}(\sigma, pc_l, id, a_s, v) = \sigma''} \text{ FS-CREATES}$$

$$\frac{\begin{array}{c}\mathsf{lookupA}_{\mathsf{FS}}(\sigma, \cdot, id, a^{\mathsf{rt}}) = \langle a | \mathsf{NULL} \rangle \\ \sigma(a) = (id, v', M, a'_p, A) \quad \sigma(a_s).v \downarrow_L \neq \cdot \quad \sigma(a_s).a_p \downarrow_L = a_p \\ \sigma(a_p) = (id_p, v_p, M_p, a'_p, (A_p :: a'_s :: A'_p)) \quad a'_s \downarrow_L = a_s \quad v_p \downarrow_L \neq \cdot \\ \sigma' = \sigma[a_p \mapsto (id_p, v', M, a'_p, (A :: a'_s :: \mathsf{createFacet}(a, L) :: A'))] \\ \sigma'' = \sigma'[a \mapsto (id, \mathsf{updateFacet}(v', v, L), \cdot, \mathsf{updateFacet}(a'_p, a_p, L), \cdot)]\end{array}}{\mathsf{createSibling}_{\mathsf{FS}}(\sigma, L, id, a_s, v) = \sigma''} \text{ FS-CREATES-UL}$$

$$\frac{\begin{array}{c}\mathsf{lookupA}_{\mathsf{FS}}(\sigma, \cdot, id, a^{\mathsf{rt}}) = \langle \mathsf{NULL} | a \rangle \\ \sigma(a) = (id, v', M, a'_p, A) \quad \sigma(a_s).v \downarrow_H \neq \cdot \quad \sigma(a_s).a_p \downarrow_H = a_p \\ \sigma(a_p) = (id_p, v_p, M_p, a'_p, (A_p :: a'_s :: A'_p)) \quad a'_s \downarrow_H = a_s \quad v_p \downarrow_H \neq \cdot \\ \sigma' = \sigma[a_p \mapsto (id_p, v', M, a'_p, (A :: a'_s :: \mathsf{createFacet}(a, H) :: A'))] \\ \sigma'' = \sigma'[a \mapsto (id, \mathsf{updateFacet}(v', v, H), \cdot, \mathsf{updateFacet}(a'_p, a_p, H), \cdot)]\end{array}}{\mathsf{createSibling}_{\mathsf{FS}}(\sigma, H, id, a_s, v) = \sigma''} \text{ FS-CREATES-UH}$$

$$\frac{\begin{array}{c}\sigma' = \mathsf{createSibling}_{\mathsf{FS}}(\sigma, H, id, \mathsf{getFacetA}(a_s, H), \mathsf{getFacetV}(v, H)) \\ \sigma'' = \mathsf{createSibling}_{\mathsf{FS}}(\sigma, L, id, \mathsf{getFacetA}(a_s, L), \mathsf{getFacetV}(v, L))\end{array}}{\mathsf{createSibling}_{\mathsf{FS}}(\sigma, \cdot, id, a_s, v) = \sigma''} \text{ FS-CREATES-NC}$$

$$\frac{\mathsf{lookupA}_{\mathsf{FS}}(\sigma, \cdot, id, a^{\mathsf{rt}}) = a \quad a \downarrow_{pc_l} \neq \cdot}{\mathsf{createSibling}_{\mathsf{FS}}(\sigma, pc_l, id, a_s, v) = \sigma} \text{ FS-CREATES-S1} \qquad \frac{a_s = \mathsf{NULL} \vee \sigma(a_s).a_p \downarrow_{pc_l} = \mathsf{NULL}}{\mathsf{createSibling}_{\mathsf{FS}}(\sigma, pc_l, id, a_s, v) = \sigma} \text{ FS-CREATES-S2}$$

$$\frac{\sigma(a_s).v \downarrow_{pc_l} = \cdot \vee \sigma(a_s).a_p \downarrow_{pc_l} = \cdot \vee \sigma(\sigma(a_s).a_p \downarrow_{pc_l}).v \downarrow_{pc_l} = \cdot}{\mathsf{createSibling}_{\mathsf{FS}}(\sigma, pc_l, id, a_s, v) = \sigma} \text{ FS-CREATES-S3}$$

Figure C.21: Rules for adding a sibling to a node in the tree-structured EH storage

$$\boxed{\mathsf{registerEH}_{\mathcal{G}}(\sigma_1^{\mathcal{G}}, pc, a, eh) = \sigma_2^{\mathcal{G}}}$$

$$\frac{\begin{array}{c}\sigma' = \mathsf{getStore}(\sigma, pc_l) \\ \sigma'(a) = (id, v, M, a_p, A) \quad eh = \mathsf{onEv}(x)\{c\} \quad M' = M[\mathsf{Ev} \mapsto M(\mathsf{Ev}) \cup \{(eh, pc_l)\}]\end{array}}{\mathsf{registerEH}_{\mathsf{SMS}}(\sigma, pc_l, a, eh) = \mathsf{setStore}(\sigma, pc_l, \sigma'[a \mapsto (id, v, M', a_p, A)])} \text{ SMS-REGISTEREH}$$

$$\frac{\sigma' = \mathsf{registerEH}_{\mathsf{SMS}}(\sigma, H, \mathsf{getFacet}(a, H), eh) \quad \sigma'' = \mathsf{registerEH}_{\mathsf{SMS}}(\sigma, L, \mathsf{getFacet}(a, L), eh)}{\mathsf{registerEH}_{\mathsf{SMS}}(\sigma, \cdot, a, eh) = \sigma''} \text{ SMS-REGISTEREH-NC}$$

$$\frac{}{\mathsf{registerEH}_{\mathsf{SMS}}(\sigma, pc_l, \mathsf{NULL}, eh) = \sigma} \text{ SMS-REGISTEREH-S}$$

Figure C.22: Rules for registering an event handler to a node in the tree-structured EH storage

Adding a sibling to a node. Figure C.21 contains rules for adding a sibling to an existing node. One of the arguments to this function is the address of the sibling node, so the node does not need to be looked up. Again, adding a sibling to a node in the SMS store is straightforward, so we show only the FS rules, which are similar to adding a child.

Registering a new event handler. Figure C.22 contains rules for registering an event handler to a node. One of the arguments to this function is the address node being updated, so the node does not need to be looked up. To register a new event handler in the SMS store, SMS-REGISTEREH looks up the node and adds the new event handler to the event handler map with the given context as the label. getStore and setStore are used to ensure that we interact with the correct copy of the SMS store. If the $pc = \cdot$, SMS-REGISTEREH-NC splits the execution and registers the event handler in both copies of the store. If the node's address is NULL, SMS-REGISTEREH-S leaves the store unchanged. Registering a new event handler in the FS store is similar.

Unstructured EH Storage expression semantics For the unstructured EH store, we only have one expression which is used to look up the attribute of a node: $\mathsf{getVal}_{\mathcal{G}}$. These rules are straightforward. If a matching node is found, we return the value, otherwise we return dv. For the TS store, we always taint dv so the attacker cannot distinguish between a tainted node and one which does not exist.

$\boxed{G, \mathcal{V}, \sigma^G, \sigma^{\mathcal{V}} \vdash e \Downarrow^i_{pc} v}$

$$\frac{\forall i \in [1, n], G, \mathcal{V}, \sigma^G, \sigma^{\mathcal{V}} \vdash e_i \Downarrow^{G \downarrow_{EH}}_{pc} v_i \qquad \mathsf{ehAPIe}(G \downarrow_{EH}, \sigma^G, pc, a, v_1, \cdots, v_n) = v \qquad \mathsf{toDst}(v, pc, i) = v'}{G, \mathcal{V}, \sigma^G, \sigma^{\mathcal{V}} \vdash \mathsf{ehAPIe}(a, e_1, \cdots, e_n) \Downarrow^i_{pc} v'} \text{ EHAPI}$$

Tree-structured EH Storage expression semantics For the tree-structured EH store, we have several expressions for navigating the tree (moveX) and looking up the attribute (getVal) and children of a node (getChildren). We evaluate EH expressions using a helper function. Using EHAPI, we first evaluate sub-expressions, then pass those results to a helper function, and finally use toDst to ensure the results are appropriately formatted for the mechanism i.

Traversing the tree. We have several APIs for navigating the EH storage trees. moveRoot returns the root node (i.e., the top-most node in the tree). moveUp takes the address of a node and returns the node above

$\boxed{\mathsf{moveX_{SMS}}(\sigma^{\mathsf{SMS}}, pc, \cdots) = a}$

$$\frac{}{\mathsf{moveRoot_{SMS}}(\sigma, pc_l) = a^{\mathsf{rt}}} \text{ SMS-MOVEROOT}$$

$$\frac{\sigma' = \mathsf{getStore}(\sigma, pc_l)}{\mathsf{moveUp_{SMS}}(\sigma, pc_l, a) = \sigma'(a).a_p} \text{ SMS-MOVEU} \qquad \frac{}{\mathsf{moveUp_{SMS}}(\sigma, pc_l, \mathsf{NULL}) = \mathsf{NULL}} \text{ SMS-MOVEU-S}$$

$$\frac{\sigma' = \mathsf{getStore}(\sigma, pc_l) \quad \sigma'(a).A = a' :: A}{\mathsf{moveDown_{SMS}}(\sigma, pc_l, a) = a'} \text{ SMS-MOVED} \qquad \frac{\sigma' = \mathsf{getStore}(\sigma, pc_l) \quad \sigma'(a).A = \cdot}{\mathsf{moveDown_{SMS}}(\sigma, pc_l, a) = \mathsf{NULL}} \text{ SMS-MOVED-S1}$$

$$\frac{}{\mathsf{moveDown_{SMS}}(\sigma, pc_l, \mathsf{NULL}) = \mathsf{NULL}} \text{ SMS-MOVED-S2}$$

$$\frac{\sigma' = \mathsf{getStore}(\sigma, pc_l) \qquad \sigma'(a).a_p = a_p \qquad \sigma'(a_p).A = A :: a :: a' :: A'}{\mathsf{moveRight_{SMS}}(\sigma, pc_l, a) = a'} \text{ SMS-MOVER}$$

$$\frac{\sigma' = \mathsf{getStore}(\sigma, pc_l) \qquad \sigma'(a).a_p = a_p \qquad \sigma'(a_p).A = A'_p :: a}{\mathsf{moveRight_{SMS}}(\sigma, pc_l, a) = \mathsf{NULL}} \text{ SMS-MOVER-S1}$$

$$\frac{\sigma' = \mathsf{getStore}(\sigma, pc_l) \qquad \sigma'(a).a_p = \mathsf{NULL}}{\mathsf{moveRight_{SMS}}(\sigma, pc_l, a) = \mathsf{NULL}} \text{ SMS-MOVER-S2}$$

$$\frac{}{\mathsf{moveRight_{SMS}}(\sigma, pc_l, \mathsf{NULL}) = \mathsf{NULL}} \text{ SMS-MOVER-S3}$$

Figure C.23: Rules for navigating the tree-structured SMS EH storage

$\boxed{\mathsf{getVal}_{\mathcal{G}}(\sigma^{\mathcal{G}}, pc, a) = v}$

$$\frac{\sigma' = \mathsf{getStore}(\sigma, pc_l) \qquad \phi = \sigma'(a)}{\mathsf{getVal_{SMS}}(\sigma, pc_l, a) = \phi.v} \text{ SMS-GETVAL} \qquad \frac{}{\mathsf{getVal_{SMS}}(\sigma, pc_l, \mathsf{NULL}) = \mathsf{dv}} \text{ SMS-GETVAL-S}$$

$$\frac{\sigma(a) = \phi}{\mathsf{getVal_{FS}}(\sigma, pc_l, a) = \mathsf{getFacetV}(\phi.v, pc_l)} \text{ FS-GETVAL} \qquad \frac{}{\mathsf{getVal_{FS}}(\sigma, pc_l, \mathsf{NULL}) = \mathsf{dv}} \text{ FS-GETVAL-S}$$

Figure C.24: Rules for accessing the attribute of node in the tree-structured EH storage

it (i.e., its parent). moveDown takes the address of a node and returns the first node below it (i.e., its first child). moveRight takes the address of a node and returns the node to its right (i.e., its righthand sibling).

Figure C.23 shows the rules for navigating the SMS tree. SMS-MOVEROOT returns the address of the root node in the tree. Recall that the root node is always given by a^{rt}. SMS-MOVEU takes the address of a node and returns its parent and SMS-MOVEU-S handles the case where the given address is NULL (in which

$\boxed{\mathsf{getChildren}_{\mathcal{G}}(\sigma^{\mathcal{G}}, pc, a) = v}$

$$\frac{\sigma' = \mathsf{getStore}(\sigma, pc_l) \qquad \phi = \sigma'(a)}{\mathsf{getChildren}_{\mathsf{SMS}}(\sigma, pc_l, a) = \mathsf{len}(\phi.A)}\ \textsc{sms-getChildren} \qquad \frac{}{\mathsf{getChildren}_{\mathsf{SMS}}(\sigma, pc_l, \mathsf{NULL}) = \mathsf{dv}}\ \textsc{sms-getChildren-s}$$

$$\frac{\sigma(a) = \phi \qquad \phi.v \downarrow_{pc_l} \neq \cdot}{\mathsf{getChildren}_{\mathsf{FS}}(\sigma, pc_l, a) = \mathsf{len}(\phi.A \downarrow_{pc_l})}\ \textsc{fs-getChildren} \qquad \frac{\sigma(a).v \downarrow_{pc_l} = \cdot \vee a = \mathsf{NULL}}{\mathsf{getChildren}_{\mathsf{FS}}(\sigma, pc_l, a) = \mathsf{dv}}\ \textsc{fs-getChildren-s}$$

Figure C.25: Rules for returning the number children a node in the tree-structured EH storage has

case we return NULL). SMS-MOVED takes the address of a node and returns its first child. SMS-MOVED-S1 and SMS-MOVED-S2 handle the case where the given node does not have any children, or the given node is NULL (respectively). In both cases we return NULL. SMS-MOVER takes the address of a node (a), navigates to its parent (a_p), and returns a_p's child following a. If there is no child following a, SMS-MOVER-S1 returns NULL. If a_p is NULL, SMS-MOVER-S2 returns NULL. If a is NULL, SMS-MOVER-S3 returns NULL. The rules for navigating the FS tree are similar.

Looking up attributes. Figure C.24 shows the rules for looking up the attribute of a given node. To look up the attribute of an SMS node, SMS-GETVAL looks up the given address in the relevant copy of the store and returns the node's attribute. If the given address is NULL, SMS-GETVAL-S returns a default value. To look up the attribute of an FS node, FS-GETVAL looks up the given address and returns the appropriate facet of the node. If the given address is NULL, FS-GETVAL-S returns a default value.

Number of children. Figure C.25 shows the rules for looking up the number of children a node has. Note that we return a default value instead of 0 for invalid or NULL nodes. This is to ensure that the attacker cannot tell the difference between a node not existing and a node being hidden from the current context. For an SMS node, SMS-GETCHILDREN first looks up the node in the relevant copy of the store. Then, it computes the length of the node's list of children. If the given address is NULL, SMS-GETCHILDREN-S returns a default value. For an FS node, FS-GETCHILDREN looks up the node and, if it is visible in the current context (i.e., $v \downarrow_{pc_l} \neq \cdot$), then it returns the length of the node's list of children. If the given address is NULL or if the node is not visible in the current context (i.e., $v \downarrow_{pc_l} = \cdot$), then FS-GETCHILDREN-S returns the default value.

C.2.6 Weak secrecy semantics

We differentiate weak from standard semantics by using $\Vdash_w$ instead of $\Vdash$, $\vdash_w$ instead of $\vdash$, assignW instead of assign, etc., respectively. For commands involving conditionals: IF-FALSE, WHILE-TRUE, and WHILE-FALSE are similar to IF-TRUE. For $\mathcal{G} \neq$ TS, all assignment functions return $\bullet$ for α. The rules are shown in Figure C.26. Some event handler APIs are also modified for TS. The rules are shown in Figure C.27. All other mechanisms return $\bullet$ for α. $\mathsf{gw}(id)$ is emitted whenever a node is created in the H context or the value of the node is upgraded from L to H.

C.3 Security Definitions

C.3.1 Configuration equivalence

Configurations are equivalent if (1) their release modules $\mathcal{R}$ are in the same state, (2) the same values are on their release channels d, (3) their global stores are equivalent σ^G, and (4) their current configuration stacks ks are equivalent. The rules are shown in Figure C.28.

Definition 107 (Configuration equivalence)**.** *Given two compositional configurations K_1 and K_2 where $K_1 = \mathcal{R}_1, d_1; \sigma_1^G; ks_1$ and $K_2 = \mathcal{R}_2, d_2; \sigma_2^G; ks_2$, $K_1 \approx_L K_2$ iff $\mathcal{R}_1 = \mathcal{R}_2$, $d_1 = d_2$, $\sigma_1^G \approx_L \sigma_2^G$, and $ks_1 \approx_L ks_2$*

Configuration stack equivalence Equivalence is defined inductively from the top down; the publicly visible ($pc \sqsubseteq L$) configurations should be equivalent and the private configurations ($pc \not\sqsubseteq L$) are ignored. Equivalence for single configurations, κ. Two configurations are low equivalent if (1) they have equivalent stores σ, (2) they are executing the same command c, (3) they are in the same execution state s, and (4) their local events E are low equivalent. (Note: the rule for SME configurations helps with proofs). The rules are shown in Figure C.28.

Local event queues are low equivalent iff their low projections are the same.

Definition 108 (Event queue equivalence)**.** *Given two local event queues E_1 and E_2, $E_1 \approx_L E_2$ iff $E_1 \downarrow_L = E_2 \downarrow_L$*

The low projection of an event queue keeps only the publicly visible events ($l \sqsubseteq L$); the secret events ($l \not\sqsubseteq L$) are ignored. Note that in our semantics, tainted arguments to local events also taints the event itself. Also, facets never appear as event arguments: instead, they are split into separate events. So only the label on the event needs to be considered for the $\downarrow_L$ definition. The rules are shown in Figure C.28.

C.3.2 Store equivalence

Local store equivalence Local stores $\sigma^{\mathcal{V}}$ are low equivalent if their low projections are the same.

$$\boxed{G, \mathsf{TT}, d \Vdash_w \sigma_1^G, \sigma_1^{\mathsf{TT}}, c_1 \xrightarrow{\alpha}_{pc} \sigma_2^G, \sigma_2^{\mathsf{TT}}, c_2, E}$$

$$\frac{G, \mathsf{TT}, \sigma^G, \sigma \vdash e \Downarrow_{pc}^{\mathsf{TT}} (\mathsf{true}, l) \qquad pc \sqsubseteq L \qquad l \not\sqsubseteq L}{G, \mathsf{TT}, d \Vdash_w \sigma^G, \sigma, \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \xrightarrow{\mathsf{br(true)}}_{pc} \sigma^G, \sigma, c_1, \cdot} \text{ IF-TRUE-BR}$$

$$\frac{G, \mathsf{TT}, \sigma_1^G, \sigma \vdash e \Downarrow_{pc}^{G\downarrow_g} v \qquad x \in \sigma^G \qquad pc \not\sqsubseteq L \qquad \mathsf{assignW}_{G\downarrow_g}(\sigma_1^G, pc, x, v) = (\sigma_2^G, \alpha)}{G, \mathsf{TT}, d \Vdash_w \sigma_1^G, \sigma_1, x := e \xrightarrow{\alpha}_{pc} \sigma_2^G, \sigma_1, \mathsf{skip}, \cdot} \text{ ASSIGN-G-H}$$

$$\frac{G, \mathsf{TT}, \sigma_1^G, \sigma \vdash e \Downarrow_{pc}^{G\downarrow_{EH}} v \qquad pc \not\sqsubseteq L \qquad \mathsf{assignW}_{G\downarrow_{EH}}(\sigma_1^G, pc, id, v) = (\sigma_2^G, \alpha)}{G, \mathsf{TT}, d \Vdash_w \sigma_1^G, \sigma_1, id := e \xrightarrow{\alpha}_{pc} \sigma_2^G, \sigma_1, \mathsf{skip}, \cdot} \text{ ASSIGN-D-H}$$

$$\frac{G, \mathsf{TT}, \sigma_1^G, \sigma \vdash e \Downarrow_{pc}^{G\downarrow_{EH}} v \qquad (\sigma_2^G, \alpha) = \mathsf{createElemW}_{G\downarrow_{EH}}(\sigma_1^G, pc, id, v)}{G, \mathsf{TT}, d \Vdash_w \sigma_1^G, \sigma_1, \mathsf{create}(id, e) \xrightarrow{\alpha}_{pc} \sigma_2^G, \sigma_1, \mathsf{skip}, \cdot} \text{ CREATEELEM-H}$$

$$\boxed{\mathsf{assignW}_{\mathsf{TS}}(\sigma, pc, x, v) = \sigma'}$$

$$\frac{x \in \sigma \qquad l \sqcup pc \sqsubseteq \mathsf{labOf}(\sigma(x), pc)}{\mathsf{assignW}_{\mathsf{TS}}(\sigma, pc, x, (v, l)) = (\sigma[x \mapsto (v, l \sqcup pc)], \bullet)} \text{ TS-ASSIGN}$$

$$\frac{x \in \sigma \qquad l \sqcup pc \not\sqsubseteq \mathsf{labOf}(\sigma(x), pc)}{\mathsf{assignW}_{\mathsf{TS}}(\sigma, pc, x, (v, l)) = (\sigma[x \mapsto (v, l \sqcup pc)], \mathsf{gw}(x))} \text{ TS-ASSIGN-GW}$$

$$\frac{x \notin \sigma}{\mathsf{assignW}_{\mathsf{TS}}(\sigma, pc, x, (v, l)) = (\sigma, \bullet)} \text{ TS-ASSIGN-S}$$

$$\frac{(id, (v', l''), M, l') = \sigma(id) \qquad l \sqcup pc \sqsubseteq l'' \vee l' \not\sqsubseteq L}{\mathsf{assignW}_{\mathsf{TS}}(\sigma, pc, id, (v, l)) = (\sigma[id \mapsto (id, (v, l \sqcup pc \sqcup l'), M, l')], \bullet)} \text{ TS-ASSIGNEH}$$

$$\frac{(id, (v', l''), M, l') = \sigma(id) \qquad l \sqcup pc \not\sqsubseteq l'' \qquad l' \sqsubseteq L}{\mathsf{assignW}_{\mathsf{TS}}(\sigma, pc, id, (v, l)) = (\sigma[id \mapsto (id, (v, l \sqcup pc \sqcup l'), M, l')], \mathsf{gw}(id))} \text{ TS-ASSIGNEH-GW}$$

$$\frac{id \notin \sigma}{\mathsf{assignW}_{\mathsf{TS}}(\sigma, pc, id, (v, l)) = (\sigma, \bullet)} \text{ TS-ASSIGNEH-S}$$

Figure C.26: Modified EH semantics for weak secrecy

$$\frac{\mathsf{lookup}_{\mathsf{TS}}(\sigma, pc_l, id) = (\mathsf{NULL}, _) \qquad \phi = (id, v, \cdot, pc_l)}{\mathsf{createElemW}_{\mathsf{TS}}(\sigma, pc_l, id, v) = (\sigma[id \mapsto \phi], \bullet)} \text{ TS-CREATE}$$

$$\frac{(\sigma', \alpha) = \mathsf{createElemW}_{\mathsf{TS}}(\sigma, L, id, v)}{\mathsf{createElemW}_{\mathsf{TS}}(\sigma, \cdot, id, v) = (\sigma', \alpha)} \text{ TS-CREATE-NC}$$

$$\frac{\begin{array}{c}\mathsf{lookup}_{\mathsf{TS}}(\sigma, pc_l, id) = (id, (v', l'), M, l'') \\ l'' \sqsubseteq pc_l \qquad l \sqcup pc_l \sqsubseteq l' \vee l'' \not\sqsubseteq L \qquad \sigma' = \sigma[id \mapsto (id, (v, l \sqcup pc_l \sqcup l''), M, l'')]\end{array}}{\mathsf{createElemW}_{\mathsf{TS}}(\sigma, pc_l, id, (v, l)) = (\sigma', \bullet)} \text{ TS-CREATE-U1}$$

$$\frac{\begin{array}{c}\mathsf{lookup}_{\mathsf{TS}}(\sigma, pc_l, id) = (id, (v', l'), M, l'') \\ l'' \sqsubseteq pc_l \qquad l'' \sqsubseteq L \qquad l \sqcup pc_l \not\sqsubseteq l' \qquad \sigma' = \sigma[id \mapsto (id, (v, l \sqcup pc_l \sqcup l''), M, l'')]\end{array}}{\mathsf{createElemW}_{\mathsf{TS}}(\sigma, pc_l, id, (v, l)) = (\sigma', \mathsf{gw}(id))} \text{ TS-CREATE-U1-GW}$$

$$\frac{\mathsf{lookup}_{\mathsf{TS}}(\sigma, pc_l, id) = (id, (v', l'), M, l'') \qquad l'' \not\sqsubseteq pc_l \qquad \sigma' = \sigma[id \mapsto (id, (v, l \sqcup pc_l), M, pc_l)]}{\mathsf{createElemW}_{\mathsf{TS}}(\sigma, pc, id, (v, l)) = (\sigma', \bullet)} \text{ TS-CREATE-U2}$$

Figure C.27: Updated event handler API semantics for weak secrecy

$\boxed{ks \approx_L ks}$

$$\frac{}{\cdot \approx_L \cdot} \qquad \frac{pc_1 \sqsubseteq L \wedge pc_2 \sqsubseteq L \qquad \mathcal{V}_1 = \mathcal{V}_2 \qquad \kappa_1 \approx_L \kappa_2 \qquad ks_1 \approx_L ks_2}{((\mathcal{V}_1; \kappa_1; pc_1) :: ks_1) \approx_L ((\mathcal{V}_2; \kappa_2; pc_2) :: ks_2)}$$

$$\frac{pc_1 \not\sqsubseteq L \qquad ks_1 \approx_L ((\mathcal{V}_2; \kappa_2; pc_2) :: ks_2)}{((\mathcal{V}_1; \kappa_1; pc_1) :: ks_1) \approx_L ((\mathcal{V}_2; \kappa_2; pc_2) :: ks_2)} \qquad \frac{pc_2 \not\sqsubseteq L \qquad ((\mathcal{V}_1; \kappa_1; pc_1) :: ks_1) \approx_L ks_2}{((\mathcal{V}_1; \kappa_1; pc_1) :: ks_1) \approx_L ((\mathcal{V}_2; \kappa_2; pc_2) :: ks_2)}$$

$\boxed{\kappa_1^{\mathcal{V}} \approx_L \kappa_2^{\mathcal{V}}}$

$$\frac{\kappa_L = \kappa'_L}{\kappa_H; \kappa_L \approx_L \kappa'_H; \kappa'_L} \qquad \frac{\begin{array}{c}\kappa_1 = \sigma_1, c_1, s_1, E_1 \qquad \kappa_2 = \sigma_2, c_2, s_2, E_2 \\ \sigma_1 \approx_L \sigma_2 \qquad c_1 = c_2 \qquad s_1 = s_2 \qquad E_1 \approx_L E_2\end{array}}{\kappa_1 \approx_L \kappa_2}$$

$\boxed{E \downarrow_L = E'}$

$$\frac{pc \sqsubseteq L}{((id.\mathsf{Ev}(v), pc), E) \downarrow_L = id.\mathsf{Ev}(v), E \downarrow_L} \qquad \frac{pc \not\sqsubseteq L}{((id.\mathsf{Ev}(v), pc), E) \downarrow_L = E \downarrow_L} \qquad \frac{}{\cdot \downarrow_L = \cdot}$$

Figure C.28: Rules for configuration stack and event queue equivalence

$\boxed{\sigma^{\mathcal{V}} \downarrow_L = \sigma^{\mathsf{std}}}$

$$\frac{}{\cdot \downarrow_L = \cdot} \qquad \frac{\sigma_1^{\mathcal{V}} = \sigma_2^{\mathcal{V}}, x \mapsto \langle _ | v \rangle \quad \text{or} \quad \sigma_1^{\mathcal{V}} = \sigma_2^{\mathcal{V}}, x \mapsto (v, L)}{\sigma_1^{\mathcal{V}} \downarrow_L = x \mapsto v, \sigma_2^{\mathcal{V}} \downarrow_L}$$

$$\frac{\sigma_1^{\mathcal{V}} = \sigma_2^{\mathcal{V}}, x \mapsto (v, H) \quad \text{or} \quad \sigma_1^{\mathcal{V}} = \sigma_2^{\mathcal{V}}, x \mapsto \langle v | \cdot \rangle}{\sigma_1^{\mathcal{V}} \downarrow_L = \sigma_2^{\mathcal{V}} \downarrow_L} \qquad \frac{\sigma_1^{\mathsf{MF}} = \sigma_2^{\mathsf{MF}}, x \mapsto v^{\mathsf{std}}}{\sigma_1^{\mathsf{MF}} \downarrow_L = x \mapsto v^{\mathsf{std}}, \sigma_2^{\mathsf{MF}} \downarrow_L} \qquad \frac{\sigma^{\mathsf{SME}} = (\sigma_H, \sigma_L)}{\sigma^{\mathsf{SME}} \downarrow_L = \sigma_L}$$

Figure C.29: Low projection of a local variable store. The rules for the global variable store are the same.

Definition 109 (Local store equivalence)**.** *Given two local stores $\sigma_1^{\mathcal{V}}$ and $\sigma_2^{\mathcal{V}}$, $\sigma_1^{\mathcal{V}} \approx_L \sigma_2^{\mathcal{V}}$ iff $\sigma_1^{\mathcal{V}} \downarrow_L = \sigma_2^{\mathcal{V}} \downarrow_L$*

The low projection of a local store $\sigma^{\mathcal{V}}$ keeps the publicly visible variables and ignores the secret variables. *Publicly visible* and *secret* depend on the enforcement mechanism, $\mathcal{V}$. For MF, unfaceted v^{std} values are publicly visible, as well as the low facet v_L of faceted values $\langle v_H | v_L \rangle$. If a faceted value does not have a low facet, as in $\langle v | \cdot \rangle$, the value is not publicly visible. For , L-labeled values are publicly visible (i.e., v is visible in (v, L)) and H-labeled values are secret. For SME, the L copy of the store is visible while the H copy is secret. The rules are shown in Figure C.29.

Global store equivalence Global stores (σ_g, σ_{EH}) are low equivalent if both their global variable stores σ_g and event handler stores σ_{EH} are equivalent.

Definition 110. *Given two global stores σ_1^G and σ_2^G where $\sigma_1^G = (\sigma_{g,1}^{\mathcal{G}}, \sigma_{EH,1}^{\mathcal{G}'})$ and $\sigma_2^G = (\sigma_{g,2}^{\mathcal{G}}, \sigma_{EH,2}^{\mathcal{G}'})$, $\sigma_1^G \approx_L \sigma_2^G$ iff $\sigma_{g,1}^{\mathcal{G}} \approx_L \sigma_{g,2}^{\mathcal{G}}$ and $\sigma_{EH,1}^{\mathcal{G}'} \approx_L \sigma_{EH,2}^{\mathcal{G}'}$*

Global variable store equivalence is defined the same as for local variable stores. Two global variable stores are equivalent if their low projections are the same. Global variable store projection is defined the same as for the local variable store (see Figure C.29).

Definition 111 (Global variable store equivalence)**.** *Given two global variable stores $\sigma_{g,1}^{\mathcal{G}}$ and $\sigma_{g,2}^{\mathcal{G}}$, $\sigma_{g,1}^{\mathcal{G}} \approx_L \sigma_{g,2}^{\mathcal{G}}$ iff $\sigma_{g,1}^{\mathcal{G}} \downarrow_L = \sigma_{g,2}^{\mathcal{G}} \downarrow_L$*

Unstructured EH store equivalence Event handler storage σ_{EH} low equivalence depends on the structure of the event handler storage. We consider an unstructured EH storage and tree structured EH storage. Two unstructured EH storages σ_{EH} are low equivalent if their low projections are the same.

Definition 112 (Unstructured event handler store equivalence)**.** *Given two unstructured event handler stores $\sigma_{EH,1}^{\mathcal{G}}$ and $\sigma_{EH,2}^{\mathcal{G}}$, $\sigma_{EH,1}^{\mathcal{G}} \approx_L \sigma_{EH,2}^{\mathcal{G}}$ iff $\sigma_{EH,1}^{\mathcal{G}} \downarrow_L = \sigma_{EH,2}^{\mathcal{G}} \downarrow_L$*

$\boxed{\sigma_{EH}^{\mathcal{G}} \downarrow_L = \sigma_{EH}^{\mathsf{std}}}$

$$\frac{\sigma_1^{\mathsf{FS}} = \sigma_2^{\mathsf{FS}}, id \mapsto \phi \qquad \phi = (v, M) \qquad v \downarrow_L \neq \cdot}{\sigma_1^{\mathsf{FS}} \downarrow_L = id \mapsto \phi \downarrow_L, \sigma_2^{\mathsf{FS}} \downarrow_L} \qquad \frac{\sigma_1^{\mathsf{FS}} = \sigma_2^{\mathsf{FS}}, id \mapsto \phi \qquad \phi = (v, M) \qquad v \downarrow_L = \cdot}{\sigma_1^{\mathsf{FS}} \downarrow_L = \sigma_2^{\mathsf{FS}} \downarrow_L}$$

$$\frac{\sigma_1^{\mathsf{TS}} = \sigma_2^{\mathsf{TS}}, id \mapsto \phi \qquad \phi = (v, M, l) \qquad l \sqsubseteq L}{\sigma_1^{\mathsf{TS}} \downarrow_L = id \mapsto \phi \downarrow_L, \sigma_2^{\mathsf{TS}} \downarrow_L} \qquad \frac{\sigma_1^{\mathsf{TS}} = \sigma_2^{\mathsf{TS}}, id \mapsto (v, M, l) \qquad l \not\sqsubseteq L}{\sigma_1^{\mathsf{TS}} \downarrow_L = \sigma_2^{\mathsf{TS}} \downarrow_L}$$

$$\frac{}{(\sigma_H, \sigma_L) \downarrow_L = \sigma_L} \qquad \frac{}{\cdot \downarrow_L = \cdot}$$

Figure C.30: Event handler store projection for unstructured event handler stores

The low projection of an unstructured EH storage is the publicly observable parts of publicly observable nodes (secret nodes are ignored). The meaning of *publicly observable* and *secret* depends on the enforcement mechanism, $\mathcal{G}$. The rules are shown in Figure C.30. For SMS, the entire L copy of the EH storage is publicly observable, while the H copy is secret. In FS, a node is visible if the value stored in the node is visible (i.e., $v \downarrow_L \neq \cdot$). If the value is not visible (i.e., $v \downarrow_L = \cdot$), the node is considered secret. For TS, a node labeled L is publicly observable, while a node labeled H is secret. The definitions for *publicly observable parts* of a node are shown in Section C.3.4.

Tree structured EH store equivalence Two tree structured EH storages σ_{EH} are low equivalent if the public view of the tree is the same in both EH stores. The meaning of the *public view* of a tree depends on the enforcement mechanism, $\mathcal{G}$. For SMS, the entire L copy of the EH storage is the public view, so they are equivalent only if their L copies of the EH storages are the same. For FS, the low copy of the EH storage is defined as the low projection of the tree, starting at the root node, located at a^{rt}. The low projection of the tree is defined below in Section C.3.4.

Definition 113 (Tree-structured event handler store equivalence)**.** *Given two tree-structured event handler stores $\sigma_{EH,1}^{\mathcal{G}}$ and $\sigma_{EH,2}^{\mathcal{G}}$ where a_1^{rt} is the root node of $\sigma_{EH,1}^{\mathcal{G}}$ and a_2^{rt} is the root node of $\sigma_{EH,2}^{\mathcal{G}}$, $\sigma_{EH,1}^{\mathcal{G}} \approx_L \sigma_{EH,2}^{\mathcal{G}}$ iff*

- *If $\mathcal{G} = \mathsf{SMS}$: $\sigma_{L,1}(a_1^{\mathsf{rt}}) \downarrow_L^{\sigma_{L,1}} = \sigma_{L,2}(a_2^{\mathsf{rt}}) \downarrow_L^{\sigma_{L,2}}$ for $\sigma_{EH,1}^{\mathcal{G}} = (\sigma_{H,1}, \sigma_{L,1})$ and $\sigma_{EH,2}^{\mathcal{G}} = (\sigma_{H,2}, \sigma_{L,2})$*
- *If $\mathcal{G} = \mathsf{FS}$: $\sigma_1^{\mathsf{FS}}(a_1^{\mathsf{rt}}) \downarrow_L^{\sigma_1^{\mathsf{FS}}} = \sigma_2^{\mathsf{FS}}(a_2^{\mathsf{rt}}) \downarrow_L^{\sigma_2^{\mathsf{FS}}}$*

C.3.3 Value equivalence and strong equivalence

We need a different definition for unstructured and tree-structured EH storages because for the tree-structured EH storage values include locations in the EH store, while the unstructured EH storage can use a simpler equivalence definition because it does not include references.

$\boxed{v \downarrow_L = v'}$

$$\frac{}{v^{\mathsf{std}} \downarrow_L = v^{\mathsf{std}}} \qquad \frac{}{\langle _|v_L\rangle \downarrow_L = v_L} \qquad \frac{}{\langle _|\cdot\rangle \downarrow_L = \cdot} \qquad \frac{}{(v, L) \downarrow_L = v} \qquad \frac{}{(v, H) \downarrow_L = \cdot}$$

Figure C.31: Value projection for unstructured EH stores

$\boxed{v \downarrow_L^\sigma = v'}$

$$\frac{v \in \{n, b, \mathsf{dv}, \mathsf{NULL}\}}{v^{\mathsf{std}} \downarrow_L^\sigma = v^{\mathsf{std}}} \qquad \frac{\phi = \sigma(a)}{a \downarrow_L^\sigma = \phi \downarrow_L^\sigma} \qquad \frac{v_L \in \{n, b, \mathsf{dv}, \mathsf{NULL}\}}{\langle _|v_L\rangle \downarrow_L^\sigma = v_L} \qquad \frac{\phi = \sigma(a_L)}{\langle _|a_L\rangle \downarrow_L^\sigma = \phi \downarrow_L^\sigma} \qquad \frac{}{\langle _|\cdot\rangle \downarrow_L^\sigma = \cdot}$$

$$\frac{v \in \{n, b, \mathsf{dv}, \mathsf{NULL}\}}{(v, L) \downarrow_L^\sigma = v} \qquad \frac{\phi = \sigma(a)}{(a, L) \downarrow_L^\sigma = \phi \downarrow_L^\sigma} \qquad \frac{}{(v, H) \downarrow_L^\sigma = \cdot}$$

Figure C.32: Value projection for tree-structured EH stores

Unstructured EH storage Two values are equivalent if their low projections are the same.

Definition 114 (Value equiv.–Unstructured EH store)**.** *Given two values v_1 and v_2, $v_1 \approx_L v_2$ iff $v_1 \downarrow_L = v_2 \downarrow_L$*

The low projection rules (when the unstructured EH store is used) are shown in Figure C.31. The low projection of a standard value v^{std} is the same value v^{std}. The low projection of a faceted value $\langle v_H|v_L\rangle$ is the value in the low facet v_L, or nothing (denoted $\cdot$) when the low facet is empty, as in $\langle v|\cdot\rangle$. The low projection of a labeled value (v, l) is the value v when the label is L, or nothing, otherwise.

Tree-structured EH storage Two values are equivalent if their low projections are the same.

Definition 115 (Value equivalence–Tree-structured EH store)**.** *Given two values v_1 and v_2, $v_1 \approx_L^{\sigma_1,\sigma_2} v_2$ iff $v_1 \downarrow_L^{\sigma_1} = v_2 \downarrow_L^{\sigma_2}$*

The low projection rules (when the tree-structured EH store is used) are shown in Figure C.32. The low projection of a standard value v^{std} is the same value v^{std} when the value is not a (non-NULL) location. If the value is a location, the low projection is the public view of the node at that location. The low projection of a faceted value $\langle v_H|v_L\rangle$ is the value in the low facet v_L, if it is not a location, or nothing (denoted $\cdot$) when the low facet is empty, as in $\langle v|\cdot\rangle$. When the value in the low facet is a location, the low projection is the public view of the node at that location. The low projection of a labeled value (v, l) is the value v when the label is L, or nothing, otherwise.

No-context projection is useful for proofs. The low projection of a value is the same value: $v \downarrow_{\cdot} = v$

Value (strong) equivalence Two values are (strong) low-equivalent when they are both low-equivalent and have publicly observable interpretations. Having *publicly observable interpretations* has different meanings, depending on the format of the value. Standard values and faceted values always have public interpretations: standard values are, themselves, public, while faceted values have a public facet (or the default value). Tainted values only have public interpretations if they themselves are publicly observable (i.e., their label is at or below L). This distinction is important for the proofs, where tainted values might introduce secrets to the public context and lead to implicit leaks, whereas standard and faceted values will not.

Definition 116 (Value strong equiv.–Unstructured EH store)**.** *Given two values* $v_1^{\mathcal{I}}$ *and* $v_2^{\mathcal{I}}$*,* $v_1^{\mathcal{I}} \simeq_L v_2^{\mathcal{I}}$ *iff*

- *If* $\mathcal{I} \in \{\mathsf{std}, \mathsf{MF}, \mathsf{FS}\}$*:* $v_1^{\mathcal{I}} \approx_L v_2^{\mathcal{I}}$
- *If* $\mathcal{I} \in \{\mathsf{TT}, \mathsf{TS}\}$*:* $(v_1', l_1) \approx_L (v_2', l_2)$ *and* $l_1 \sqsubseteq L$*,* $l_2 \sqsubseteq L$ *for* $v_1^{\mathcal{I}} = (v_1', l_1)$ *and* $v_2^{\mathcal{I}} = (v_2', l_2)$

Definition 117 (Value strong equiv.–Tree-structured EH store)**.** *Given two values* $v_1^{\mathcal{I}}$ *and* $v_2^{\mathcal{I}}$*,* $v_1^{\mathcal{I}} \simeq_L^{\sigma_1,\sigma_2} v_2^{\mathcal{I}}$ *iff*

- *If* $\mathcal{I} \in \{\mathsf{std}, \mathsf{MF}, \mathsf{FS}\}$*:* $v_1^{\mathcal{I}} \approx_L^{\sigma_1,\sigma_2} v_2^{\mathcal{I}}$
- *If* $\mathcal{I} \in \{\mathsf{TT}, \mathsf{TS}\}$*:* $(v_1', l_1) \approx_L^{\sigma_1,\sigma_2} (v_2', l_2)$ *and* $l_1 \sqsubseteq L$*,* $l_2 \sqsubseteq L$ *for* $v_1^{\mathcal{I}} = (v_1', l_1)$ *and* $v_2^{\mathcal{I}} = (v_2', l_2)$

C.3.4 Node equivalence

Unstructured EH storage Publicly observable nodes ϕ are low-equivalent in an unstructured EH storage if their low projections (their publicly observable parts) are the same. The low projection of a secret EH storage node is denoted $\cdot$ since none of it is publicly observable.

Definition 118 (Node equiv.–Unstructured EH store)**.** *Given two nodes* ϕ_1 *and* ϕ_2*,* $\phi_1 \approx_L \phi_2$ *iff* $\phi_1 \downarrow_L = \phi_2 \downarrow_L$

The low projection of a node ϕ in an unstructured EH storage is defined differently for different enforcement mechanisms, $\mathcal{G}$. The rules are shown in Figure C.33. The publicly observable parts of a standard node ϕ^{std} are its value and publicly observable event handlers $M \downarrow_L$. For a faceted node ϕ^{FS}, the publicly observable parts are the low projection of the value in the node $v \downarrow_L$, and the publicly observable event handlers $M \downarrow_L$. If the value in the node is not publicly observable (i.e., $v \downarrow_L = \cdot$), then the entire node is considered secret. Finally, the public view of a tainted node ϕ^{TS} is the public view of its value $(v, l) \downarrow_L$ and event handler map $M \downarrow_L$. If the label on the node itself is H, the entire node is considered secret. If the value in a public node is considered secret, it is replaced with a default value dv in the public view.

$\boxed{\phi^{\mathcal{G}} \downarrow_L = \phi^{\mathsf{std}}}$

$$\frac{\phi = (v, M)}{\phi^{\mathsf{std}} \downarrow_L = (v, M \downarrow_L)} \qquad \frac{\phi = (v, M) \qquad v \downarrow_L \neq \cdot}{\phi^{\mathsf{FS}} \downarrow_L = (v \downarrow_L, M \downarrow_L)} \qquad \frac{\phi = (v, M) \qquad v \downarrow_L = \cdot}{\phi^{\mathsf{FS}} \downarrow_L = \cdot}$$

$$\frac{\phi = ((v, l), M, l') \qquad l \sqsubseteq L \qquad l' \sqsubseteq L}{\phi^{\mathsf{TS}} \downarrow_L = (v, M \downarrow_L)} \qquad \frac{\phi = ((v, l), M, l') \qquad l \not\sqsubseteq L \qquad l' \sqsubseteq L}{\phi^{\mathsf{TS}} \downarrow_L = (\mathsf{dv}, M \downarrow_L)}$$

$$\frac{\phi = (v, M, l) \qquad l \not\sqsubseteq L}{\phi^{\mathsf{TS}} \downarrow_L = \cdot} \qquad \frac{}{\mathsf{NULL} \downarrow_L = \mathsf{NULL}} \qquad \frac{l \sqsubseteq L}{(\mathsf{NULL}, l) \downarrow_L = \mathsf{NULL}} \qquad \frac{l \not\sqsubseteq L}{(\mathsf{NULL}, l) \downarrow_L = \cdot}$$

Figure C.33: Node projection for an unstructured EH store

Tree structured EH storage

Definition 119 (Node equivalence–Tree-structured EH store)**.** *Given two nodes ϕ_1 and ϕ_2, $\phi_1 \approx_L^{\sigma_1,\sigma_2} \phi_2$ iff $\phi_1 \downarrow_L^{\sigma_1} = \phi_2 \downarrow_L^{\sigma_2}$*

The public view of a tree-structured node given store σ, $\phi \downarrow_L^{\sigma}$, is defined inductively over the node's children. Denote $N = (id, v^{\mathsf{std}}, M, ID, Ns)$ or $\cdot$ where ID is an id or $\cdot$, and Ns is an ordered list of N's. The rules are shown in Figure C.34. For a standard node ϕ^{std}, the public view includes (1) the node's *id*, (2) the value stored in the node v, (3) the public event handlers $M \downarrow_L$, (4) the *id* of the parent, and (5) an ordered list of the public view of each of its children $A \downarrow_L^{\sigma}$. If the node has no parent ($a_p = \mathsf{NULL}$), then ID in the public view is $\cdot$. The public view of a faceted node ϕ^{FS} includes (1) the node's *id*, (2) the public view of the value stored in the node $v \downarrow_L$, (3) the public event handlers $M \downarrow_L$, (4) the *id* of the publicly observable parent, and (5) and ordered list of the public view of each of its children $A \downarrow_L^{\sigma}$. If the publicly observable parent is NULL ($a_p \downarrow_L = \mathsf{NULL}$), then ID in the public view is $\cdot$. If there is no publicly observable value ($v \downarrow_L = \cdot$) or parent ($a_p \downarrow_L = \cdot$), then the node itself is not publicly observable.

The public view of a list of nodes $A \downarrow_L^{\sigma}$ from store σ is defined inductively on the structure of the list and produces an ordered list of publicly observable nodes Ns. If the first element in the list is a standard address a (i.e., not faceted), then the public view of the list is the public view of the node at that address $\phi \downarrow_L^{\sigma}$, followed by the public view of the rest of the list. If the first address in the list is faceted a^{FS}, then the public view of the list is public view of the node $\phi \downarrow_L^{\sigma}$ at the public view of the address $a \downarrow_L$ followed by the public view of the rest of the list. If the address is not publicly visible (i.e., $a \downarrow_L = \cdot$), then that address is ignored.

$\boxed{\phi^{\mathcal{G}} \downarrow_L^\sigma = N}$

$$\frac{\phi = (id, v, M, a_p, A) \qquad id_p = \sigma(a_p).id}{\phi^{\mathsf{std}} \downarrow_L^\sigma = (id, v \downarrow_L^\sigma, M \downarrow_L, id_p, A \downarrow_L^\sigma)} \qquad \frac{\phi = (id, v, M, \mathsf{NULL}, A)}{\phi^{\mathsf{std}} \downarrow_L^\sigma = (id, v \downarrow_L^\sigma, M \downarrow_L, \cdot, A \downarrow_L^\sigma)}$$

$$\frac{\phi = (id, v, M, a_p, A) \qquad a_p \downarrow_L \neq \cdot \qquad id_p = \sigma(a_p \downarrow_L).id \qquad v \downarrow_L \neq \cdot}{\phi^{\mathsf{FS}} \downarrow_L^\sigma = (id, v \downarrow_L^\sigma, M \downarrow_L, id_p, A \downarrow_L^\sigma)}$$

$$\frac{\phi = (id, v, M, a_p, A) \qquad a_p = \mathsf{NULL} \vee a_p \downarrow_L = \mathsf{NULL} \qquad v \downarrow_L \neq \cdot}{\phi^{\mathsf{FS}} \downarrow_L^\sigma = (id, v \downarrow_L^\sigma, M \downarrow_L, \cdot, A \downarrow_L^\sigma)}$$

$$\frac{\phi = (id, v, M, a_p, A) \qquad v \downarrow_L = \cdot \vee a_p \downarrow_L = \cdot}{\phi^{\mathsf{FS}} \downarrow_L^\sigma = \cdot} \qquad \frac{}{\mathsf{NULL} \downarrow_L^\sigma = \mathsf{NULL}}$$

$\boxed{A \downarrow_L^\sigma = Ns}$

$$\frac{\phi = \sigma(a)}{(a :: A) \downarrow_L^\sigma = \phi \downarrow_L^\sigma :: A \downarrow_L^\sigma} \qquad \frac{a \downarrow_L \neq \cdot \qquad \phi = \sigma(a \downarrow_L)}{(a^{\mathsf{FS}} :: A^{\mathsf{FS}}) \downarrow_L^\sigma = \phi^{\mathsf{FS}} \downarrow_L^\sigma :: A^{\mathsf{FS}} \downarrow_L^\sigma} \qquad \frac{a \downarrow_L = \cdot}{(a^{\mathsf{FS}} :: A^{\mathsf{FS}}) \downarrow_L^\sigma = A^{\mathsf{FS}} \downarrow_L^\sigma} \qquad \frac{}{(\cdot) \downarrow_L^\sigma = \cdot}$$

Figure C.34: Node projection for a tree-structured EH store

$$\frac{EH \downarrow_L = EH' \neq \emptyset}{((\mathsf{Ev} \mapsto EH), M) \downarrow_L = \mathsf{Ev} \mapsto EH', M \downarrow_L} \qquad \frac{EH \downarrow_L = \emptyset}{((\mathsf{Ev} \mapsto EH), M) \downarrow_L = M \downarrow_L} \qquad \frac{}{\cdot \downarrow_L = \cdot}$$

$$\frac{pc \sqsubseteq L}{(\{eh, pc\} \cup EH) \downarrow_L = \{eh, L\} \cup EH \downarrow_L} \qquad \frac{pc \not\sqsubseteq L}{(\{eh, pc\} \cup EH) \downarrow_L = EH \downarrow_L} \qquad \frac{}{\emptyset \downarrow_L = \emptyset}$$

Figure C.35: Event handler map projection

Event handler map projection

The low projection of an event handler map M is defined inductively over the structure of the map. The rules are shown in Figure C.35. For one event $\mathsf{Ev} \mapsto EH$, it is defined as the low projection of the event handler sets EH for each event Ev. Events which do not have publicly observable event handlers (i.e., $EH \downarrow_L = \emptyset$) are ignored.

The low projection of an event handler set EH is the set of publicly observable event handlers in EH. An event handler is public if it was registered under a public pc (i.e., $pc \sqsubseteq L$). In the projected set, all publicly observable event handlers have $pc = L$. Secret event handlers (i.e., $pc \not\sqsubseteq L$) are ignored.

$\boxed{c \downarrow_L = c}$

$$\frac{}{c^{\mathsf{std}} \downarrow_L = c^{\mathsf{std}}} \qquad \frac{}{\langle c_H | c_L \rangle \downarrow_L = c_L}$$

$\boxed{\mathcal{C} \downarrow_L = \mathcal{C}'}$

$$\frac{pc \sqsubseteq L}{((eh, pc), \mathcal{C}) \downarrow_L = eh, \mathcal{C} \downarrow_L} \qquad \frac{pc \not\sqsubseteq L}{((eh, pc), \mathcal{C}) \downarrow_L = \mathcal{C} \downarrow_L} \qquad \frac{}{\cdot \downarrow_L = \cdot}$$

Figure C.36: Command and EH queue projection rules

C.3.5 Additional equivalence definitions

Command and EH queue equivalence Commands and EH queues are low equivalent if their low projections are the same.

Definition 120 (Command equivalence)**.** *Given two commands c_1 and c_2, $c_1 \approx_L c_2$ iff $c_1 \downarrow_L = c_2 \downarrow_L$*

Definition 121 (EH queue equivalence)**.** *Given two EH queues EH_1 and EH_2, $EH_1 \approx_L EH_2$ iff $EH_1 \downarrow_L = EH_2 \downarrow_L$*

The low projection of most commands c is the same command, c. The only exception is when the command is faceted $\langle c_H | c_L \rangle$. In that case, the low projection is the command in the low facet, c_L. Rules are shown in Figure C.36. The EH queue is a runtime construct for building the configuration stack. EH queues $\mathcal{C}$ are low equivalent if their low projections are the same. The low projection keeps the publicly visible event handlers ($pc \sqsubseteq L$). The secret event handlers ($pc \not\sqsubseteq L$) are ignored. Note that command and EH queue equivalence is not used for configuration equivalence, but it is useful for MF proofs.

C.3.6 Trace Equivalence

Two traces are equivalent if their low projections are the same.

Definition 122 (Trace equivalence)**.** *Given two traces T_1 and T_2, $T_1 \approx_L T_2$ iff $T_1 \downarrow_L = T_2 \downarrow_L$*

The rules for the low projection of a trace are shown in Figure C.37. They are similar to the rules from Figure 5.10 in Section 5.4 except they also handle faceted actions. Note that $\mathcal{P}(\alpha) = \cdot$ has a different interpretation than the $\cdot$ we have previously used. Here, it means "it has no label".

$\boxed{T \downarrow_L = \tau}$

$$\frac{}{G, \mathcal{P} \vdash K \downarrow_L = \cdot}\ \text{TP-BASE} \qquad \frac{\mathcal{P}(id.\mathsf{Ev}(v)) = L}{(G, \mathcal{P} \vdash K \overset{id.\mathsf{Ev}(v)}{\Longrightarrow} T') \downarrow_L = id.\mathsf{Ev}(v) :: T' \downarrow_L}\ \text{TP-LI}$$

$$\frac{\mathcal{P}(\alpha) = L \quad \text{or} \quad l \sqsubseteq L}{(G, \mathcal{P} \vdash K \overset{(\alpha,l)}{\Longrightarrow} T') \downarrow_L = \alpha :: T' \downarrow_L}\ \text{TP-L} \qquad \frac{\alpha_I = \langle _|_ \rangle}{(G, \mathcal{P} \vdash K \overset{\alpha_I}{\Longrightarrow} T') \downarrow_L = \mathsf{getFacet}(\alpha_I, L) :: T' \downarrow_L}\ \text{TP-F}$$

$$\frac{\mathsf{rel}(K) = \mathcal{R} \qquad \mathcal{P}(id.ev(v)) = H \qquad \mathcal{R}(\mathcal{P}, id.ev(v)) = \alpha \neq \bullet}{(G, \mathcal{P} \vdash K \overset{id.ev(v)}{\Longrightarrow} T') \downarrow_L = \mathsf{rls}(\alpha) :: T' \downarrow_L}\ \text{TP-HI-R}$$

$$\frac{\mathcal{P}(id.ev(v)) = H_\Delta}{(G, \mathcal{P} \vdash K \overset{id.ev(v)}{\Longrightarrow} T') \downarrow_L = T' \downarrow_L}\ \text{TP-HI-NR1}$$

$$\frac{\mathsf{rel}(K) = \mathcal{R} \qquad \mathcal{P}(id.ev(v)) = H \wedge \mathcal{R}(\mathcal{P}, id.ev(v)) = \bullet}{(G, \mathcal{P} \vdash K \overset{id.ev(v)}{\Longrightarrow} T') \downarrow_L = T' \downarrow_L}\ \text{TP-HI-NR2} \qquad \frac{\mathcal{P}(\alpha) = H \quad \text{or} \quad \alpha = \bullet}{(G, \mathcal{P} \vdash K \overset{(\alpha,H)}{\Longrightarrow} T') \downarrow_L = T' \downarrow_L}\ \text{TP-H}$$

$\boxed{\mathcal{P}(\alpha) = l}$

$$\frac{}{\mathcal{P}(ch(v)) = \mathcal{P}(ch)}\ \text{LAB-O} \qquad \frac{}{\mathcal{P}(\mathsf{declassify}(\iota, v)) = L}\ \text{LAB-D} \qquad \frac{}{\mathcal{P}(\mathsf{gw}(x)) = L}\ \text{LAB-GW}$$

$$\frac{}{\mathcal{P}(\mathsf{br}(b)) = L}\ \text{LAB-BR} \qquad \frac{\alpha \notin \{id.\mathsf{Ev}(v), ch(v), \mathsf{declassify}(\iota, v), \mathsf{gw}(x), \mathsf{br}(b)\}}{\mathcal{P}(\alpha) = \cdot}\ \text{LAB-S}$$

$\boxed{\mathcal{R}(\mathcal{P}, id.ev(v)) = \alpha}$

$$\frac{\mathcal{R} = (\rho, \mathcal{D}) \qquad \mathcal{D}(\rho, id.\mathsf{Ev}(v)) = (\rho', r, v', \mathcal{V})}{\mathcal{R}(\mathcal{P}, id.\mathsf{Ev}(v)) = (\rho', r, id.\mathsf{Ev}(v'))} \qquad \frac{\mathcal{R} = (\rho, \mathcal{D}) \qquad \mathcal{D}(\rho, id.\mathsf{Ev}(v)) = (\rho', r, \mathsf{emp}, \mathcal{V})}{\mathcal{R}(\mathcal{P}, id.\mathsf{Ev}(v)) = (\rho', r, \bullet)}$$

$$\frac{\mathcal{R} = (\rho, \mathcal{D}) \qquad \mathcal{D}(\rho, id.\mathsf{Ev}(v)) = (\rho, \mathsf{none}, \bullet, \mathcal{V})}{\mathcal{R}(\mathcal{P}, id.\mathsf{Ev}(v)) = \bullet}$$

Figure C.37: Projection of Traces to L Observation

C.3.7 Knowledge Definitions

Definition 21 (Attacker Knowledge). *Formally, $\mathcal{K}(T, \sigma_0^G, \mathcal{R}, \mathcal{P})$ is defined as* $\{\tau_i \mid \exists T' \in \mathsf{runs}(\sigma_0^G, \mathcal{R}, \mathcal{P}), T \approx_L T', \tau_i = \mathsf{in}(T')\}$

Definition 22 (Progress Knowledge). *Formally, $\mathcal{K}_p(T, \sigma_0^G, \mathcal{R}, \mathcal{P})$ is defined as* $\{\tau_i \mid \exists T' \in \mathsf{runs}(\sigma_0^G, \mathcal{R}, \mathcal{P}), T \approx_L T', \tau_i = \mathsf{in}(T'), \mathsf{prog}(T')\}$

Where $\mathsf{prog}(T)$ iff $T = G, \mathcal{P} \vdash K_0 \Longrightarrow^* K$ and $\exists K_C$ s.t. $G, \mathcal{P} \vdash K \Longrightarrow^* K_C$ and $\mathsf{consumer}(K_C)$

Definition 23 (Release Knowledge). *Formally, $\mathcal{K}_{rp}(T, \sigma_0^G, \mathcal{R}, \mathcal{P}, \alpha)$ is defined as* $\{\tau_i \mid \exists T' \in \mathsf{runs}(\sigma_0^G, \mathcal{R}, \mathcal{P}), T \approx_L T', \tau_i = \mathsf{in}(T'), \mathsf{prog}(T'), \alpha' = (\mathsf{last}(T) \overset{\alpha}{\Longrightarrow} K) \downarrow_L, \mathsf{releaseT}(T', \alpha')\}$

$$\mathsf{releaseT}(T,\alpha) = \begin{cases} T = G,\mathcal{P} \vdash K_0 \Longrightarrow^* K \wedge \exists \alpha', K' s.t., G,\mathcal{P} \vdash K \stackrel{\alpha'}{\Longrightarrow} K' & \\ \quad \wedge (G,\mathcal{P} \vdash K \stackrel{\alpha'}{\Longrightarrow} K') \downarrow_L = \alpha & \alpha = \mathsf{rls}(_) \\ T = G,\mathcal{P} \vdash K_0 \Longrightarrow^* K, \exists K' s.t., G,\mathcal{P} \vdash K \stackrel{\alpha}{\Longrightarrow} K' & \alpha = \mathsf{declassify}(\iota, v) \end{cases}$$

Definition 26 (Weak Knowledge). *Formally,* $\mathcal{K}_{wp}(T, \sigma_0^G, \mathcal{R}, \mathcal{P}, \alpha)$ *is defined as* $\{\tau_i \mid \exists T' \in \mathsf{runs}(\sigma_0^G, \mathcal{R}, \mathcal{P}), T \approx_L T', \tau_i = \mathsf{in}(T'), \mathsf{prog}(T'), \alpha' = (G,\mathcal{P} \vdash \mathsf{last}(T) \stackrel{\alpha}{\Longrightarrow} K)) \downarrow_L, \mathsf{wkT}(T', \alpha'))\}$

$$\mathsf{wkT}(T,\alpha) = \begin{cases} T = G,\mathcal{P} \vdash K_0 \Longrightarrow^* K \wedge \exists K' \text{ s.t. } K \Longrightarrow K' \wedge (G,\mathcal{P} \vdash K \Longrightarrow K') \downarrow_L = \alpha & \alpha = \mathsf{br}(b) \\ T = G,\mathcal{P} \vdash K_0 \Longrightarrow^* K \wedge \exists K', T' \text{ s.t. } T' = G,\mathcal{P} \vdash K \Longrightarrow^* K' \wedge T' \downarrow_L = \cdot & \alpha = \mathsf{gw}(x) \\ \quad \wedge \exists K'' \text{ s.t. } G,\mathcal{P} \vdash K' \Longrightarrow^* K'' \wedge (G,\mathcal{P} \vdash K' \Longrightarrow^* K'') \downarrow_L = \alpha & \\ T = G,\mathcal{P} \vdash K_0 \Longrightarrow^* K & \alpha = id.\mathsf{Ev}(v) \\ \quad \wedge \exists T', K' \text{ s.t. } T' = G,\mathcal{P} \vdash K \Longrightarrow^* K_C \wedge \mathsf{consumer}(K_C) \wedge T' \downarrow_L = \cdot & \\ T = G,\mathcal{P} \vdash K_0 \Longrightarrow^* K & \alpha \in \{\bullet, ch(v)\} \\ \quad \wedge \exists T', K' \text{ s.t. } T' = G,\mathcal{P} \vdash K \Longrightarrow^* K_{lp} \wedge \mathsf{lowProducer}(K_{lp}) \wedge T' \downarrow_L = \cdot & \end{cases}$$

$$\frac{\neg\mathsf{consumer}(K) \quad K = \mathcal{R}, d, \sigma^G, (\mathcal{V};\kappa;pc) :: ks \quad pc \sqsubseteq L}{\mathsf{lowProducer}(K)} \qquad \frac{\neg\mathsf{consumer}(K) \quad K = \mathcal{R}, d, \sigma^G, (\mathcal{V};\kappa;pc) :: ks \quad pc \not\sqsubseteq L}{\mathsf{highProducer}(K)}$$

$$\frac{pc \sqsubseteq L}{\mathsf{lowContext}((\mathcal{V};\kappa;pc) :: ks) = \mathsf{true}} \qquad \frac{pc \not\sqsubseteq L}{\mathsf{lowContext}((\mathcal{V};\kappa;pc) :: ks) = \mathsf{false}}$$

$$\frac{pc \not\sqsubseteq L}{\mathsf{highContext}((\mathcal{V};\kappa;pc) :: ks) = \mathsf{true}} \qquad \frac{pc \sqsubseteq L}{\mathsf{highContext}((\mathcal{V};\kappa;pc) :: ks) = \mathsf{false}}$$

$$
\begin{array}{lll}
\mathsf{rlsA}(G,\mathcal{P}\vdash K \overset{\alpha}{\Longrightarrow} K') & \text{iff} & (G,\mathcal{P}\vdash K \overset{\alpha}{\Longrightarrow} K')\downarrow_L = \mathsf{rls}(\alpha') \text{ or} \\
 & & (G,\mathcal{P}\vdash K \overset{\alpha}{\Longrightarrow} K')\downarrow_L = \mathsf{declassify}(\iota, v) \\
\mathsf{wkA}(G,\mathcal{P}\vdash K \overset{\alpha}{\Longrightarrow} K') & \text{iff} & (G,\mathcal{P}\vdash K \overset{\alpha}{\Longrightarrow} K')\downarrow_{L,w} = \mathsf{br}(b) \text{ or} \\
 & & (G,\mathcal{P}\vdash K \overset{\alpha}{\Longrightarrow} K')\downarrow_{L,w} = \mathsf{gw}(x) \text{ or} \\
 & & \alpha \in \mathsf{in}(G,\mathcal{P}\vdash K \overset{\alpha}{\Longrightarrow} K') \wedge \mathcal{P}(\alpha) = L \text{ or} \\
 & & (G,\mathcal{P}\vdash K \overset{\alpha}{\Longrightarrow} K')\downarrow_L \in \{\bullet, ch(v)\}
\end{array}
$$

C.3.8 Progress-Insensitive Security and Weak Secrecy

We say σ_0^G is well-formed if the initial EH store is defined and the global variables are initialized. The event handlers in the EH store should agree on the names of the global variables.

Definition 24 (Progress-Insensitive Security). *The compositional framework is progress-insensitive secure iff given any initial global store σ_0^G and release policy $\mathcal{R}, \mathcal{P}$, it is the case that for all traces T, actions α, and configurations K s.t. $(G,\mathcal{P}\vdash T \overset{\alpha}{\Longrightarrow} K) \in \mathsf{runs}(\sigma_0^G, \mathcal{P})$, then, the following holds*

- *If* $\mathsf{rlsA}(G,\mathcal{P}\vdash \mathsf{last}(T) \overset{\alpha}{\Longrightarrow} K)$*:*
  $\mathcal{K}(G,\mathcal{P}\vdash T \overset{\alpha}{\Longrightarrow} K, \sigma_0^G, \mathcal{R}) \supseteq_{\preceq} \mathcal{K}_{rp}(T, \sigma_0^G, \mathcal{R}, \mathcal{P}, \alpha)$
- *Otherwise:*
  $\mathcal{K}(G,\mathcal{P}\vdash T \overset{\alpha}{\Longrightarrow} K, \sigma_0^G, \mathcal{R}, \mathcal{P}) \supseteq_{\preceq} \mathcal{K}_{p}(T, \sigma_0^G, \mathcal{R}, \mathcal{P})$

Definition 27 (Progress-Insensitive Weak Security). *The compositional framework satisfies progress-insensitive weak secrecy in our framework iff given any initial global store, σ_0^G, and release policy $\mathcal{R}, \mathcal{P}$ it is the case that for all traces T, actions α, and configurations K s.t. $(G,\mathcal{P}\vdash T \overset{\alpha}{\Longrightarrow} K) \in \mathsf{runs}(\sigma_0^G, \mathcal{P})$, the following holds*

- *If* $\mathsf{rlsA}(G,\mathcal{P}\vdash \mathsf{last}(T) \overset{\alpha}{\Longrightarrow} K)$*:*
  $\mathcal{K}(G,\mathcal{P}\vdash T \overset{\alpha}{\Longrightarrow} K, \sigma_0^G, \mathcal{R}, \mathcal{P}) \supseteq_{\preceq} \mathcal{K}_{rp}(T, \sigma_0^G, \mathcal{R}, \mathcal{P}, \alpha)$
- *If* $\mathsf{wkA}(G,\mathcal{P}\vdash \mathsf{last}(T) \overset{\alpha}{\Longrightarrow} K)$*:*
  $\mathcal{K}(G,\mathcal{P}\vdash T \overset{\alpha}{\Longrightarrow} K, \sigma_0^G, \mathcal{R}, \mathcal{P}) \supseteq_{\preceq} \mathcal{K}_{wp}(T, \sigma_0^G, \mathcal{R}, \mathcal{P}, \alpha)$
- *Otherwise:*
  $\mathcal{K}(G,\mathcal{P}\vdash T \overset{\alpha}{\Longrightarrow} K, \sigma_0^G, \mathcal{R}, \mathcal{P}) \supseteq_{\preceq} \mathcal{K}_{p}(T, \sigma_0^G, \mathcal{R}, \mathcal{P})$

C.4 Proofs

C.4.1 Progress-Insensitive Noninterference implies Weak Secrecy

Theorem 29 (Progress-Insensitive Noninterference implies Weak Secrecy). *If the composition of event handlers and global storage enforcement satisfies progress-insensitive noninterference, then they also satisfy progress-insensitive weak secrecy.*

Proof.

We want to show that the conditions for weak secrecy hold for any trace satisfying progress-insensitive noninterference.

Let $\mathcal{V}, G$ be progress-insensitive secure and σ_0^G be well-formed.

Let T, α, K be s.t. $(G, \mathcal{P} \vdash T \stackrel{\alpha}{\Longrightarrow} K) \in \mathsf{runs}(\sigma_0^G, \mathcal{R}, \mathcal{P})$. Then,

(1) If $\mathsf{rlsA}(G, \mathcal{P} \vdash \mathsf{last}(T) \stackrel{\alpha}{\Longrightarrow} K)$:

$\mathcal{K}(G, \mathcal{P} \vdash T \stackrel{\alpha}{\Longrightarrow} K, \sigma_0^G, \mathcal{R}, \mathcal{P}) \supseteq_{\preceq} \mathcal{K}_{rp}(T, \sigma_0^G, \mathcal{R}, \mathcal{P}, \alpha)$

(2) Otherwise:

$\mathcal{K}(G, \mathcal{P} \vdash T \stackrel{\alpha}{\Longrightarrow} K, \sigma_0^G, \mathcal{R}, \mathcal{P}) \supseteq_{\preceq} \mathcal{K}_{p}(T, \sigma_0^G, \mathcal{R}, \mathcal{P})$

We examine each case of $G, \mathcal{P} \vdash T \stackrel{\alpha}{\Longrightarrow} K$. For each case, we want to show that the corresponding condition for weak secrecy holds.

Case I: $\mathsf{rlsA}(\mathsf{last}(T) \stackrel{\alpha}{\Longrightarrow} K)$

This case follows from (1) since the corresponding case for release events is the same for weak secrecy as it is for standard security.

Case II: $\mathsf{wkA}(\mathsf{last}(T) \stackrel{\alpha}{\Longrightarrow} K)$

From the assumption, we want to show

$\mathcal{K}(G, \mathcal{P} \vdash T \stackrel{\alpha}{\Longrightarrow} K, \sigma_0^G, \mathcal{R}, \mathcal{P}) \supseteq_{\preceq} \mathcal{K}_{wp}(T, \sigma_0^G, \mathcal{R}, \mathcal{P}, \alpha)$

By assumption and since visible, non-release events fall into the "other" category from (2),

$\mathcal{K}(G, \mathcal{P} \vdash T \stackrel{\alpha}{\Longrightarrow} K, \sigma_0^G, \mathcal{R}, \mathcal{P}) \supseteq_{\preceq} \mathcal{K}_{p}(T, \sigma_0^G, \mathcal{R}, \mathcal{P})$

Then, it is sufficient to show that:

$\mathcal{K}_p(T, \sigma_0^G, \mathcal{R}, \mathcal{P}) \supseteq_{\preceq} \mathcal{K}_{wp}(T, \sigma_0^G, \mathcal{R}, \mathcal{P}, \alpha)$ i.e., for any $\tau \in \mathcal{K}_{wp}(T, \sigma_0^G, \mathcal{R}, \mathcal{P}, \alpha)$,

$\exists \tau' \in \mathcal{K}_p(T, \sigma_0^G, \mathcal{R}, \mathcal{P})$ s.t. $\tau \preceq \tau'$

Let $\tau \in \mathcal{K}_{wp}(T, \sigma_0^G, \mathcal{R}, \mathcal{P}, \alpha)$

Then, from the definition of $\mathcal{K}_{wp}()$, there is a trace for which τ is the input and all of the following:

$\exists T' \in \mathsf{runs}(\sigma_0^G, \mathcal{R}, \mathcal{P})$ s.t. $T \approx_L T'$, $\tau = \mathsf{in}(T')$, and $\mathsf{prog}(T')$

Then from the definition of $\mathcal{K}_p()$, $\tau \in \mathcal{K}_p(T, \sigma_0^G, \mathcal{R}, \mathcal{P})$

If we let $\tau' = \tau$ then $\tau' \in \mathcal{K}_p(T, \sigma_0^G, \mathcal{R}, \mathcal{P})$ and from the definition of $\preceq$, $\tau \preceq \tau'$

Case III: $\neg\mathsf{rlsA}(\mathsf{last}(T) \stackrel{\alpha}{\Longrightarrow} K)$ and $\neg\mathsf{wkA}(\mathsf{last}(T) \stackrel{\alpha}{\Longrightarrow} K)$

This case follows from (2) since the corresponding case for non-release events is the same for standard security as it is for the non-release, non-visible events for weak secrecy.

□

C.4.2 Top-Level Soundness Theorems

Theorem 25 (Soundness–Progress-Insensitive Noninterference). *If* $\forall id.\mathsf{Ev}(v), eh, pc : \mathcal{P}(id.\mathsf{Ev}(v), eh, pc) \in \{\mathsf{SME}, \mathsf{MF}\}$ *and* $\mathcal{G}_g, \mathcal{G}_{EH} \in \{\mathsf{SMS}, \mathsf{FS}\}$ *and* $G = (\mathcal{G}_g, \mathcal{G}_{EH})$*, then* $\forall \mathcal{R}, \mathcal{P}, \sigma_0, T, K, \alpha_l$ *s.t.* $G, \mathcal{P} \vdash T \stackrel{\alpha_l}{\Longrightarrow} K \in \mathsf{runs}(\sigma_0^G, \mathcal{R}, \mathcal{P}, \mathcal{I})$*, and* σ_0^G *is well-formed,*

- *If* $\mathsf{rlsA}(\mathsf{last}(T) \stackrel{\alpha_l}{\Longrightarrow} K)$*:* $\mathcal{K}(G, \mathcal{P} \vdash T \stackrel{\alpha_l}{\Longrightarrow} K, \sigma_0^G, \mathcal{R}, \mathcal{P}) \supseteq_{\preceq} \mathcal{K}_{rp}(T, \sigma_0^G, \mathcal{R}, \mathcal{P}, \alpha_l)$
- *Otherwise:* $\mathcal{K}(G, \mathcal{P} \vdash T \stackrel{\alpha_l}{\Longrightarrow} K, \sigma_0^G, \mathcal{R}, \mathcal{P}) \supseteq_{\preceq} \mathcal{K}_p(T, \sigma_0^G, \mathcal{R}, \mathcal{P})$

Proof.

The proof is split between two cases depending on the action, shown below. In either case, we want to show that $\exists \tau' \in \mathcal{K}(G, \mathcal{P} \vdash T \stackrel{\alpha_l}{\Longrightarrow} K, \sigma_0^G, \mathcal{R}, \mathcal{P})$ s.t. $\tau \preceq \tau'$ for τ defined below

Case I: $\mathsf{rlsA}(G, \mathcal{P} \vdash \mathsf{last}(T) \stackrel{\alpha_l}{\Longrightarrow} K)$

Let $\tau \in \mathcal{K}_{rp}(T, \sigma_0^G, \mathcal{R}, \mathcal{P}, \alpha_l)$ and $\alpha' = (G, \mathcal{P} \vdash \mathsf{last}(T) \stackrel{\alpha_l}{\Longrightarrow} K) \downarrow_L$

Then from the definition of $\mathcal{K}_{rp}()$, there is a trace for which τ is the input $T_1 = G, \mathcal{P} \vdash K_0 \Longrightarrow^* K_1$ and $\tau = \mathsf{in}(T_1)$ and all of the following:

$T_1 \approx_L T$, $\mathsf{prog}(T_1)$, and $\mathsf{release}(T_1, \alpha')$

From $\mathsf{release}(T_1, \alpha')$, $\exists K_1', \alpha_{l,1}$ s.t. $G, \mathcal{P} \vdash T_1 \stackrel{\alpha_{l,1}}{\Longrightarrow} K_1'$ with $(G, \mathcal{P} \vdash K_1 \stackrel{\alpha_{l,1}}{\Longrightarrow} K_1') \downarrow_L = \alpha'$

Then from $T_1 \approx_L T$, we know that $(G, \mathcal{P} \vdash T \stackrel{\alpha_l}{\Longrightarrow} K) \approx_L (G, \mathcal{P} \vdash T_1 \stackrel{\alpha_{l,1}}{\Longrightarrow} K_1')$

From this the definition of $\mathcal{K}()$, $\mathsf{in}(T_1) :: \mathsf{in}(G, \mathcal{P} \vdash K_1 \Longrightarrow K_1') \in \mathcal{K}(G, \mathcal{P} \vdash T \stackrel{\alpha_l}{\Longrightarrow} K, \sigma_0^G, \mathcal{R}, \mathcal{P})$

Let $\tau' = \mathsf{in}(T_1) :: \mathsf{in}(G, \mathcal{P} \vdash K_1 \Longrightarrow K_1')$, then we have $\tau \preceq \tau'$

Case II: $\neg\mathsf{rlsA}(G, \mathcal{P} \vdash \mathsf{last}(T) \stackrel{\alpha_l}{\Longrightarrow} K)$

Let $\tau \in \mathcal{K}_p(T, \sigma_0^G, \mathcal{R}, \mathcal{P}, \alpha_l)$

Then from the definition of $\mathcal{K}_p()$, there is a trace for which τ is the input $T_1 = G, \mathcal{P} \vdash K_0 \Longrightarrow^* K_1$ with $\tau = \mathsf{in}(T_1)$ and both of the following:

$T_1 \approx_L T$ and $\mathsf{prog}(T_1)$

We also know from $\mathcal{K}_{rp}()$ there is a trace $T = G, \mathcal{P} \vdash K_0 \Longrightarrow^* K_2$ and $G, \mathcal{P} \vdash K_2 \stackrel{\alpha_l}{\Longrightarrow} K$

Subcase i: $(G, \mathcal{P} \vdash \mathsf{last}(T) \stackrel{\alpha_l}{\Longrightarrow} K) \downarrow_L = \cdot$

By assumption, and from $T \approx_L T_1$, we know $T \approx_L T_1 \approx_L (G, \mathcal{P} \vdash T \xRightarrow{\alpha_I} K)$

Let $\tau' = \mathsf{in}(T_1)$, then from the definition of $\mathcal{K}()$, $\tau' \in \mathcal{K}(G, \mathcal{P} \vdash T \xRightarrow{\alpha_I} K, \sigma_0^G, \mathcal{R}, \mathcal{P})$ and $\tau \preceq \tau'$

Subcase ii: $(G, \mathcal{P} \vdash \mathsf{last}(T) \xRightarrow{\alpha_I} K) \downarrow_L \neq \cdot$

From $T_1 \approx_L T$ and Lemma 123 (Requirement **(T1)**), $K_1 \approx_L K_2$

By assumption and from $\mathsf{prog}(T_1)$, $K_1 \approx_L K_2$, and Lemma 133 (Requirement **(T4)**),

$\exists K_1', \tau''$ s.t. $G, \mathcal{P} \vdash K_1 \xRightarrow{\tau''}^* K_1'$ and $(G, \mathcal{P} \vdash K_1 \xRightarrow{\tau''}^* K_1') \approx_L (G, \mathcal{P} \vdash K_2 \xRightarrow{\alpha_I} K)$

From this and $T \approx_L T_1$, $(G, \mathcal{P} \vdash T_1 \xRightarrow{\tau''}^* K_1') \approx_L (G, \mathcal{P} \vdash T \xRightarrow{\alpha_I} K)$

Then from the definition of $\mathcal{K}()$, we know that

$\mathsf{in}(T_1) :: \mathsf{in}(G, \mathcal{P} \vdash K_1 \xRightarrow{\tau''}^* K_1') \in \mathcal{K}(G, \mathcal{P} \vdash T \xRightarrow{\alpha_I} K, \sigma_0^G, \mathcal{R}, \mathcal{P})$

Let $\tau' = \mathsf{in}(T_1) :: \mathsf{in}(G, \mathcal{P} \vdash K_1 \xRightarrow{\tau''}^* K_1')$ then

$\tau' \in \mathcal{K}(G, \mathcal{P} \vdash T \xRightarrow{\alpha_I} K, \sigma_0^G, \mathcal{R}, \mathcal{P})$ and $\tau \preceq \tau'$

□

Theorem 28 (Soundness - Weak Secrecy). *If* $\forall id.\mathsf{Ev}(v), eh, pc : \mathcal{P}(id.\mathsf{Ev}(v), eh, pc) \in \{\mathsf{SME}, \mathsf{MF}, \mathsf{TT}\}$ *and* $\mathcal{G}_g, \mathcal{G}_{EH} \in \{\mathsf{SMS}, \mathsf{FS}, \mathsf{TS}\}$ *and* $G = (\mathcal{G}_g, \mathcal{G}_{EH})$*, then* $\forall \mathcal{R}, \mathcal{P}, \sigma_0, T, K, \alpha_I$ *s.t.* $G, \mathcal{P} \vdash T \xRightarrow{\alpha_I} K \in \mathsf{runs}(\sigma_0^G, \mathcal{R}, \mathcal{P}, \mathcal{I})$, *and* σ_0^G *is well-formed,*

- *If* $\mathsf{rlsA}(G, \mathcal{P} \vdash \mathsf{last}(T) \xRightarrow{\alpha} K)$: $\mathcal{K}(G, \mathcal{P} \vdash T \xRightarrow{\alpha} K, \sigma_0^G, \mathcal{R}, \mathcal{P}) \supseteq_{\preceq} \mathcal{K}_{rp}(T, \sigma_0^G, \mathcal{R}, \mathcal{P}, \alpha)$
- *If* $\mathsf{wkA}(G, \mathcal{P} \vdash \mathsf{last}(T) \xRightarrow{\alpha} K)$: $\mathcal{K}(G, \mathcal{P} \vdash T \xRightarrow{\alpha} K, \sigma_0^G, \mathcal{R}, \mathcal{P}) \supseteq_{\preceq} \mathcal{K}_{wp}(T, \sigma_0^G, \mathcal{R}, \mathcal{P}, \alpha)$
- *Otherwise:* $\mathcal{K}(G, \mathcal{P} \vdash T \xRightarrow{\alpha} K, \sigma_0^G, \mathcal{R}, \mathcal{P}) \supseteq_{\preceq} \mathcal{K}_p(T, \sigma_0^G, \mathcal{R}, \mathcal{P})$

Proof.

The proof is split between two cases depending on the action, shown below. In either case, we want to show that $\exists \tau' \in \mathcal{K}(G, \mathcal{P} \vdash T \xRightarrow{\alpha_I} K, \sigma_0^G, \mathcal{R}, \mathcal{P})$ s.t. $\tau \preceq \tau'$ for τ defined below

Case I: $\mathsf{rlsA}(G, \mathcal{P} \vdash \mathsf{last}(T) \xRightarrow{\alpha_I} K)$

Let $\tau \in \mathcal{K}_{rp}(T, \sigma_0^G, \mathcal{R}, \mathcal{P}, \alpha_I)$ and $\alpha' = (G, \mathcal{P} \vdash \mathsf{last}(T) \xRightarrow{\alpha_I} K) \downarrow_L$

Then from the definition of $\mathcal{K}_{rp}()$, there is a trace for which τ is the input $T_1 = G, \mathcal{P} \vdash K_0 \Longrightarrow^* K_1$ and $\tau = \mathsf{in}(T_1)$ and all of the following:

$T_1 \approx_L T$, $\mathsf{prog}(T_1)$, and $\mathsf{release}(T_1, \alpha')$

From $\mathsf{release}(T_1, \alpha')$, $\exists K_1', \alpha_{I,1}$ s.t. $G, \mathcal{P} \vdash T_1 \xRightarrow{\alpha_{I,1}} K_1'$ with $(G, \mathcal{P} \vdash K_1 \xRightarrow{\alpha_{I,1}} K_1') \downarrow_L = \alpha'$

Then from $T_1 \approx_L T$, we know that $(G, \mathcal{P} \vdash T \xRightarrow{\alpha_I} K) \approx_L (G, \mathcal{P} \vdash T_1 \xRightarrow{\alpha_{I,1}} K_1')$

From this and the definition of $\mathcal{K}()$, $\mathsf{in}(T_1) :: \mathsf{in}(G, \mathcal{P} \vdash K_1 \Longrightarrow K_1') \in \mathcal{K}(G, \mathcal{P}\mathcal{P}T \xRightarrow{\alpha_I} K, \sigma_0^G, \mathcal{R}, \mathcal{P})$

Let $\tau' = \mathsf{in}(T_1) :: \mathsf{in}(G, \mathcal{P} \vdash K_1 \Longrightarrow K_1')$, then we have $\tau \preceq \tau'$

Case II: $\mathsf{wkA}(G, \mathcal{P} \vdash \mathsf{last}(T) \xRightarrow{\alpha_I} K)$

Let $\tau \in \mathcal{K}_{wp}(T, \sigma_0^G, \mathcal{R}, \mathcal{P}, \alpha_I)$ and $\alpha' = (G, \mathcal{P} \vdash \mathsf{last}(T) \xRightarrow{\alpha_I} K) \downarrow_L$

Then from the definition of $\mathcal{K}_{wp}()$, there is a trace for which τ is the input $T_1 = G, \mathcal{P} \vdash K_0 \Longrightarrow^* K_1$ and $\tau = \mathsf{in}(T_1)$ and all of the following:

$T_1 \approx_L T$, $\mathsf{prog}(T_1)$, and $\mathsf{wkT}(T_1, \alpha')$

We also know from $\mathcal{K}_{wp}()$ there is a trace $T = G, \mathcal{P} \vdash K_0 \Longrightarrow^* K_2$ and $G, \mathcal{P} \vdash K_2 \xRightarrow{\alpha_I} K$

Subcase i: $\alpha' = \mathsf{br}(b)$

By assumption and from $\mathsf{wkT}(T_1, \alpha')$, $\exists K_1'$ s.t. $G, \mathcal{P} \vdash T_1 \Longrightarrow K_1'$ with $(G, \mathcal{P} \vdash K_1 \Longrightarrow K_1') \downarrow_L = \alpha'$

Then from $T_1 \approx_L T$, we know that $(G, \mathcal{P} \vdash T \xRightarrow{\alpha_I} K) \approx_L (G, \mathcal{P} \vdash T_1 \Longrightarrow K_1')$

From this and the definition of $\mathcal{K}()$, $\mathsf{in}(T_1) :: \mathsf{in}(G, \mathcal{P} \vdash K_1 \Longrightarrow K_1') \in \mathcal{K}(G, \mathcal{P} \vdash T \xRightarrow{\alpha_I} K, \sigma_0^G, \mathcal{R}, \mathcal{P})$

Let $\tau' = \mathsf{in}(T_1) :: \mathsf{in}(G, \mathcal{P} \vdash K_1 \Longrightarrow K_1')$, then we have $\tau \preceq \tau'$

Subcase ii: $\alpha' = \mathsf{gw}(x)$

By assumption and from $\mathsf{wkT}(T_1, \alpha')$, $\exists K_1', K_1''$ s.t. $G, \mathcal{P} \vdash T_1 \Longrightarrow^* K_1'$ with $(G, \mathcal{P} \vdash K_1 \Longrightarrow^* K_1') \downarrow_L = \cdot$ and $(G, \mathcal{P} \vdash K_1' \Longrightarrow^* K_1'') \downarrow_L = \alpha'$

Then from $T_1 \approx_L T$, we know that $(G, \mathcal{P} \vdash T \xRightarrow{\alpha_I} K) \approx_L (G, \mathcal{P} \vdash T_1 \Longrightarrow K_1')$

From this and the definition of $\mathcal{K}()$, $\mathsf{in}(T_1) :: \mathsf{in}(G, \mathcal{P} \vdash K_1 \Longrightarrow K_1'') \in \mathcal{K}(G, \mathcal{P} \vdash T \xRightarrow{\alpha_I} K, \sigma_0^G, \mathcal{R}, \mathcal{P})$

Let $\tau' = \mathsf{in}(T_1) :: \mathsf{in}(G, \mathcal{P} \vdash K_1 \Longrightarrow K_1'')$ then we have $\tau \preceq \tau'$

Subcase iii: $\alpha' = id.\mathsf{Ev}(v)$ or $\alpha' \in \{\bullet, ch(v)\}$

From $T_1 \approx_L T$ and Lemma 124 (Requirement **(WT1)**), $K_1 \approx_L K_2$

By assumption and from $\mathsf{prog}(T_1)$, $K_1 \approx_L K_2$, and Lemma 139 (Requirement **(WT4)**),

$\exists K_1'$ s.t. $G, \mathcal{P} \vdash K_1 \Longrightarrow^* K_1'$ with $(G, \mathcal{P} \vdash K_2 \Longrightarrow K) \approx_L (G, \mathcal{P} \vdash K_1 \Longrightarrow K_1')$

From this and $T_1 \approx_L$T, $(G, \mathcal{P} \vdash T_1 \Longrightarrow^* K_1') \approx_L (G, \mathcal{P} \vdash T \Longrightarrow K)$

Then from the definition of $\mathcal{K}()$, we know that

$\mathsf{in}(T_1) :: \mathsf{in}(G, \mathcal{P} \vdash K_1 \overset{\tau''}{\Longrightarrow^*} K_1') \in \mathcal{K}(G, \mathcal{P} \vdash T \xRightarrow{\alpha_I} K, \sigma_0^G, \mathcal{R}, \mathcal{P})$

Let $\tau' = \mathsf{in}(T_1) :: \mathsf{in}(G, \mathcal{P} \vdash K_1 \overset{\tau''}{\Longrightarrow^*} K_1')$ then

$\tau' \in \mathcal{K}(G, \mathcal{P} \vdash T \xRightarrow{\alpha_I} K, \sigma_0^G, \mathcal{R}, \mathcal{P})$ and $\tau \preceq \tau'$

Case III: $\neg\mathsf{rlsA}(G, \mathcal{P} \vdash \mathsf{last}(T) \xRightarrow{\alpha_I} K)$ and $\neg\mathsf{wkA}(G, \mathcal{P} \vdash \mathsf{last}(T) \xRightarrow{\alpha_I} K)$

Let $\tau \in \mathcal{K}_p(T, \sigma_0^G, \mathcal{R}, \mathcal{P}, \alpha_I)$

Then from the definition of $\mathcal{K}_p()$, there is a trace for which τ is the input $T_1 = G, \mathcal{P} \vdash K_0 \Longrightarrow^* K_1$ with $\tau = \mathsf{in}(T_1)$ and both of the following

$T_1 \approx_L T$ and $\mathsf{prog}(T_1)$

We also know from $\mathcal{K}_{rp}()$ there is a trace $T = G, \mathcal{P} \vdash K_0 \Longrightarrow^* K_2$ and $G, \mathcal{P} \vdash K_2 \xRightarrow{\alpha_I} K$

Subcase i: $(G, \mathcal{P} \vdash \mathsf{last}(T) \xRightarrow{\alpha_l} K) \downarrow_L = \cdot$

By assumption and from $T_1 \approx_L T$, we know that $T \approx_L T_1 \approx_L (G, \mathcal{P} \vdash T \xRightarrow{\alpha_l} K)$

Let $\tau' = \mathsf{in}(T_1)$, then from the definition of $\mathcal{K}()$, $\tau' \in \mathcal{K}(G, \mathcal{P} \vdash T \xRightarrow{\alpha_l} K, \sigma_0^G, \mathcal{R}, \mathcal{P})$ and $\tau \preceq \tau'$

Subcase ii: $(G, \mathcal{P} \vdash \mathsf{last}(T) \xRightarrow{\alpha_l} K) \downarrow_L \neq \cdot$

By assumption, and from the definition of $\downarrow_L$, α_l is a release event, branch, global write, low input, or low output. Then, $\mathsf{rlsA}(G, \mathcal{P} \vdash \mathsf{last}(T) \xRightarrow{\alpha_l} K)$ or $\mathsf{wkA}(G, \mathcal{P} \vdash \mathsf{last}(T) \xRightarrow{\alpha_l} K)$, which violates the assumption that neither condition holds. Therefore, this case holds vacuously.

□

C.4.3 Trace Requirements

Requirement (T1) Equivalent traces produce L-equivalent states

Lemma 123 (Equivalent Trace, Equivalent State)**.** *If $T_1 = G, \mathcal{P} \vdash K_1 \Longrightarrow^* K_1'$ and $T_2 = G, \mathcal{P} \vdash K_2 \Longrightarrow^* K_2'$ with $K_1 \approx_L K_2$ and $T_1 \approx_L T_2$, then $K_1' \approx_L K_2'$*

Proof.

By induction on $\mathsf{len}(T_1)$ and $\mathsf{len}(T_2)$

Base Case I: $\mathsf{len}(T_1) = 0$ and $\mathsf{len}(T_2) = n$

By assumption, $T_1 = K_1$, $K_1' = K_1$, and $T_1 \downarrow_L = \cdot$

Then from $T_1 \approx_L T_2$, $T_2 \downarrow_L = \cdot$ and Lemma 125 (Requirement **(T2)**) gives $K_2 \approx_L K_2'$

Then $K_1 \approx_L K_2$ gives $K_1' \approx_L K_2'$

Base Case II: $\mathsf{len}(T_1) = n$ and $\mathsf{len}(T_2) = 0$

The proof is similar to **Base Case I**

Inductive Case III: $\mathsf{len}(T_1) = n + 1$ and $\mathsf{len}(T_2) = m + 1$

We assume the conclusion holds for $\mathsf{len}(T_1) \leq n$ and $\mathsf{len}(T_2) \leq m$

By assumption, $T_1 = G, \mathcal{P} \vdash K_1 \Longrightarrow^* K_1'' \Longrightarrow K_1'$ with $\mathsf{len}(G, \mathcal{P} \vdash K_1 \Longrightarrow^* K_1'') = n$ and $T_2 = G, \mathcal{P} \vdash K_2 \Longrightarrow^* K_2'' \Longrightarrow K_2'$ with $\mathsf{len}(G, \mathcal{P} \vdash K_2 \Longrightarrow^* K_2'') = m$

Subcase i: $(G, \mathcal{P} \vdash K_1'' \Longrightarrow K_1') \downarrow_L = \cdot$

By assumption, $T_1 \approx_L (G, \mathcal{P} \vdash K_1 \Longrightarrow^* K_1'')$

From this and $T_1 \approx_L T_2$, $T_2 \approx_L (G, \mathcal{P} \vdash K_1 \Longrightarrow^* K_1'')$

Then the IH may be applied on $(G, \mathcal{P} \vdash K_1 \Longrightarrow^* K_1'')$ and T_2 which gives $K_1'' \approx_L K_2'$

By assumption and from Lemma 125 (Requirement **(T2)**), $K_1'' \approx_L K_1'$

Then from $K_1'' \approx_L K_2'$ and $K_1'' \approx_L K_1'$, we know that $K_1' \approx_L K_2'$

Subcase ii: $(G, \mathcal{P} \vdash K_2'' \Longrightarrow K_2') \downarrow_L = \cdot$

The proof is similar to **Subcase i**

Subcase iii: $(G, \mathcal{P} \vdash K_1'' \Longrightarrow K_1') \downarrow_L \neq \cdot$ and $(G, \mathcal{P} \vdash K_2'' \Longrightarrow K_2') \downarrow_L \neq \cdot$

By assumption and from $T_1 \approx_L T_2$, $(G, \mathcal{P} \vdash K_1 \Longrightarrow^* K_1'') \approx_L (G, \mathcal{P} \vdash K_2 \Longrightarrow^* K_2'')$ and

$(G, \mathcal{P} \vdash K_1'' \Longrightarrow K_1') \approx_L (G, \mathcal{P} \vdash K_2'' \Longrightarrow K_2')$

Then the IH may be applied on $(G, \mathcal{P} \vdash K_1 \Longrightarrow^* K_1'')$ and $(G, \mathcal{P} \vdash K_2 \Longrightarrow^* K_2'')$ which gives $K_1'' \approx_L K_2''$

Then from Lemma 144 (Requirement **(T5)**), $K_1' \approx_L K_2'$

□

Requirement (WT1) Equivalent traces produce L-equivalent states (Weak Secrecy)

Lemma 124 (Equivalent Trace, Equivalent State, Weak Secrecy). *If $T_1 = G, \mathcal{P} \vdash_w K_1 \Longrightarrow^* K_1'$ and $T_2 = G, \mathcal{P} \vdash_w K_2 \Longrightarrow^* K_2'$ with $K_1 \approx_L K_2$ and $T_1 \approx_L T_2$, then $K_1' \approx_L K_2'$*

Proof (sketch): The proof is the same as for Lemma 123, except that it uses Lemma 128 (Req **(WT2)**) and Lemma 147 (Req **(WT5)**) □

Requirement (T2) Empty traces produce L-equivalent states

Lemma 125. *If $T = G, \mathcal{P} \vdash K \Longrightarrow^* K'$ and $T \downarrow_L = \cdot$, then $K \approx_L K'$*

Proof.

By induction on the length of T.

Base Case I: $\mathsf{len}(T) = 0$

By assumption, $T = K$ and $K' = K$. Thus $K \approx_L K'$.

Inductive Case II: $\mathsf{len}(T) = n + 1$

By assumption, $T = G, \mathcal{P} \vdash K \Longrightarrow^* K_1 \Longrightarrow K_2$

Want to show $K \approx_L K_2$

From $T \downarrow_L = \cdot$, we also know that $(G, \mathcal{P} \vdash K \Longrightarrow^* K_1) \downarrow_L = \cdot$

IH on $(G, \mathcal{P} \vdash K \Longrightarrow^* K_1)$ gives $K \approx_L K_1$

Let $T' = G, \mathcal{P} \vdash K_1 \Longrightarrow K_2$, then from $T \downarrow_L = \cdot$, we also know that $T' \downarrow_L = \cdot$

Next, we want to show $K_1 \approx_L K_2$

Subcase i: T' ends in I-NR1 or I-NR2

By assumption, $\sigma_1^G = \sigma_2^G$, $\kappa_1 = \cdot$, and $G, \mathcal{P}, \sigma_1^G \vdash ks_1; \mathsf{lookupEHAll}(id.\mathsf{Ev}(v)) \rightsquigarrow_H ks_2$

From Lemma 182 (Requirement **(EH2)**), $ks_1 \approx_L ks_2$

Then, we know $K_1 \approx_L K_2$

Subcase ii: T' ends in I-R-Diff, I-R-Same, or I-L,

By assumption and from the definition of $\downarrow_L$ for execution traces, $(G, ks \vdash K_1 \Longrightarrow K_2) \downarrow_L \neq \cdot$

But this contradicts $T \downarrow_L = \cdot$, so this case holds vacuously

Subcase iii: T' ends in O, O-Skip, or O-Other with $pc \sqsubseteq L$

By assumption and from the definition of $\downarrow_L$ for execution traces, $(G, ks \vdash K_1 \Longrightarrow K_2) \downarrow_L \neq \cdot$

But this contradicts $T \downarrow_L = \cdot$, so this case holds vacuously

Subcase iv: T' ends in O, O-Skip, or O-Other with $pc \not\sqsubseteq L$

The conclusion follows from our security lattice (i.e., pc must be H) and Lemma 126

□

Lemma 126. *If $G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_1^G, \kappa^{\mathcal{V}} \xrightarrow{\alpha}_H \sigma_2^G, ks$, then $\sigma_1^G \approx_L \sigma_2^G$ and $(\mathcal{V}; \kappa^{\mathcal{V}}; H) \approx_L ks$*

Proof (sketch): By induction on the structure of $\mathcal{E} :: G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_1^G, \kappa \xrightarrow{\alpha}_H \sigma_2^G, ks$. The proof also uses Lemma 127 and Lemma 182 (Requirement **(EH2)**). □

Lemma 127. *If $G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1^{\mathcal{V}}, c_1 \xrightarrow{\alpha}_H \sigma_2^G, \sigma_2^{\mathcal{V}}, c_2, E$, then $\sigma_1^G \approx_L \sigma_2^G$*

Proof (sketch): By induction on the structure of $\mathcal{E} :: G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1^{\mathcal{V}}, c_1 \xrightarrow{\alpha}_H \sigma_2^G, \sigma_2^{\mathcal{V}}, c_2, E$. The proof also uses Lemma 157 (Requirement **(V2)**), and Lemmas 189.U, 190.U, 189.T, and 190.T (Requirement **(EH3)**).

□

Requirement (WT2) Empty traces produce L-equivalent states (Weak Secrecy)

Lemma 128. *If $T = G, \mathcal{P} \vdash_w K \Longrightarrow^* K'$ and $T \downarrow_L = \cdot$, then $K \approx_L K'$*

Proof (sketch): The proof is the same as for Lemma 125 (Requirement **(T2)**), except that it uses Lemma 129. The additional assumption that $\alpha \notin \{\mathsf{br}(b), \mathsf{gw}(x)\}$ follows from $T \downarrow_L = \cdot$. □

Lemma 129. *If $G, \mathcal{P}, \mathcal{V}, d \vdash_w \sigma_1^G, \kappa_1 \xrightarrow{\alpha}_H \sigma_2^G, \kappa_2$ with $\alpha \notin \{\mathsf{br}(b), \mathsf{gw}(x)\}$, then $\sigma_1^G \approx_L \sigma_2^G$ and $\kappa_1 \approx_L \kappa_2$*

Proof (sketch): The proof is the same as for Lemma 126 except that is uses Lemma 130, Note that Lemma 182 (Requirement **(EH2)**) is also used. □

Lemma 130. *If $G, \mathcal{V}, d \Vdash_w \sigma_1^G, \sigma_1^{\mathcal{V}}, c_1 \xrightarrow{\alpha}_H \sigma_2^G, \sigma_2^{\mathcal{V}}, c_2, E$, with $\alpha \notin \{\mathsf{br}(b), \mathsf{gw}(x)\}$ then $\sigma_1^G \approx_L \sigma_2^G$ and $\sigma_1^{\mathcal{V}} \approx_L \sigma_2^{\mathcal{V}}$ and $E \approx_L \cdot$*

Proof (sketch): The proof is similar to the one for Lemma 127. The cases for assign-G and assign-D which involve upgrades hold vacuously due to the assumption that $\alpha \notin \{\mathsf{br}(b), \mathsf{gw}(x)\}$. Otherwise, this proof uses Lemma 158 (Requirement **(WV2)**) instead of Lemma 157 (Requirement **(V2)**) and Lemma 192 (Requirement **(WEH3)**) instead of Lemma 189 (Requirement **(EH3)**), and Lemma 195 (Requirement **(WEH3)**) instead of Lemma 190 (Requirement **(EH3)**). □

Requirement (T3) H steps produce L-equivalent states and empty traces

Lemma 131 (H Step Equivalence). *If* $T = G, \mathcal{P} \vdash K_1 \Longrightarrow^* K_2$ *and* $\forall \alpha \in \tau, \mathsf{output}(\alpha)$ *with* $K_1 = \mathcal{R}, d; \sigma_1; ks_1$ *and* $ks_1 \approx_L \cdot$*, then,* $K_1 \approx_L K_2$ *and* $T \downarrow_L = \cdot$

Proof.

By induction on $\mathsf{len}(T)$

Base Case 1: $\mathsf{len}(T) = 0$

By assumption, $T = K_1$ and $K_2 = K_1$. Then, $K_1 \approx_L K_2$ and $T \downarrow_L = \cdot$

Base Case 2: $\mathsf{len}(T) = 1$

By assumption, $T = G, \mathcal{P} \vdash K_1 \xLongrightarrow{\alpha_I} K_2$

Case I: $\mathcal{E}$ ends in an input rule

This contradicts $\forall \alpha \in \tau, \mathsf{output}(\alpha)$, therefore this case holds vacuously

Case II: $\mathcal{E}$ ends in an output rule with $pc \sqsubseteq L$

This contradicts $ks_1 \approx_L \cdot$, therefore this case holds vacuously

Case III: $\mathcal{E}$ ends in an output rule with $pc \not\sqsubseteq L$

By assumption and from $ks_1 \approx_L \cdot$, we know $pc = H$ and $\mathcal{E}$ ends in O, O-Skip, or O-Other

And from the definitions of outCondition and output, $T \downarrow_L = \cdot$

From Lemma 126 (Requirement **(T2)**), $\sigma_1^G \approx_L \sigma_2^G$ and $(\mathcal{V}; \kappa; H) \approx_L ks$

Therefore, $K_1 \approx_L K_2$

Inductive Case: $\mathsf{len}(T) = n + 1$

By assumption, $T = G, \mathcal{P} \vdash K_1 \Longrightarrow^* K_1' \Longrightarrow K_2$

IH on $G, \mathcal{P} \vdash K_1 \Longrightarrow^* K_1'$ gives $K_1 \approx_L K_1'$ and $(G, \mathcal{P} \vdash K_1 \Longrightarrow^* K_1') \downarrow_L = \cdot$

From $K_1 \approx_L K_1'$ and $ks_1 \approx_L \cdot$, the ks in K_1' is $\approx_L \cdot$

By the same argument as **Base Case 2**, $K_1' \approx_L K_2$ and $(G, \mathcal{P} \vdash K_1' \Longrightarrow K_2) \downarrow_L = \cdot$

□

Lemma 132 (High Step Equivalence - MF, TT). *If* $T = G, \mathcal{P} \vdash K_1 \Longrightarrow^* K_C$ *with* $\mathsf{consumer}(K_C)$, $K_1 = \mathcal{R}, d; \sigma_1; ks_1$ *with* $ks_1 \not\approx_L \cdot$*, and* $\mathsf{highProducer}(K_1)$ *then* $\exists K_2$ *s.t.* $\mathsf{lowProducer}(K_2)$ *and* $T = G, \mathcal{P} \vdash K_1 \Longrightarrow^* K_2 \Longrightarrow^* K_C$ *with* $K_1 \approx_L K_2$ *and* $(G, \mathcal{P} \vdash K_1 \Longrightarrow^* K_2) \downarrow_L = \cdot$

Proof.

By induction on the length of T

Base Case: $\mathsf{len} = 0$

By assumption, $T = K_1 = K_C$. From this and $\mathsf{consumer}(K_C)$, we know $\mathsf{consumer}(K_1)$. But this contradicts $ks_1 \not\approx_L \cdot$ and $\mathsf{highProducer}(K_1)$, so this case holds vacuously.

Base Case: $\mathsf{len} = 1$

By assumption, $\mathcal{E} :: T = G, \mathcal{P} \vdash K_1 \Longrightarrow K_C$. From this and the structure of the operational semantics, $\mathcal{E}$ must end in O-NEXT with $ks_1 = (\mathcal{V}; \kappa; pc)$ and $\mathsf{consumer}(\kappa)$. But this contradicts $ks_1 \not\approx_L \cdot$, so this case holds vacuously.

Base Case: $\mathsf{len} = 2$

By assumption, $T = G, \mathcal{P} \vdash K_1 \Longrightarrow K_2 \Longrightarrow K_C$

Denote $\mathcal{D} :: G, \mathcal{P} \vdash K_1 \Longrightarrow K_2$ and $\mathcal{E} :: G, \mathcal{P} \vdash K_2 \Longrightarrow K_C$

Want to show $\mathsf{lowProducer}(K_2)$, $K_1 \approx_L K_2$, and $(G, \mathcal{P} \vdash K_1 \Longrightarrow K_2) \downarrow_L = \cdot$

Case I: $\mathcal{D}$ ends in an input rule

By assumption, $\mathsf{consumer}(K_1)$. But this contradicts $ks_1 \not\approx_L \cdot$ and $\mathsf{highProducer}(K_1)$, so this case holds vacuously.

Case II: $\mathcal{D}$ ends in O, O-SKIP, or O-OTHER

By assumption and from the structure of the operational semantics,

$\exists \mathcal{D}' :: G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_1^G, \kappa \xrightarrow{\alpha}_{pc} \sigma_2^G, ks$ with $ks_1 = (\mathcal{V}; \kappa; pc) :: \kappa_1'$ and $ks_2 = ks :: ks_1'$

From $ks_1 \not\approx_L \cdot$ and $\mathsf{highProducer}(K_1)$, $pc = H$ and $ks_1' \not\approx_L \cdot$

By assumption and from $pc = H$ and the definitions of $\mathsf{outCondition}$ and output,

$(G, \mathcal{P} \vdash K_1 \Longrightarrow K_2) \downarrow_L = \cdot$

Subcase i: $\mathcal{D}'$ ends in LC

By assumption and from $pc = H$, $\exists \mathcal{D}'' :: G, \mathcal{P}, \mathcal{V}, \sigma_1^G \vdash (\mathcal{V}; (\sigma, \mathsf{skip}, C, \cdot); H); \mathsf{lookupEHs}(E) \rightsquigarrow_H ks$

From the rules for $\mathsf{lookupEHs}$, which put $(\mathcal{V}; (\sigma, \mathsf{skip}, C, \cdot); H)$ at the top of the resulting ks, it must be the case that $ks_2 = (\mathcal{V}; (\sigma, \mathsf{skip}, C, \cdot); H) :: ks' :: ks_1'$

From $\mathsf{highProducer}(K_1)$ and Lemma 182 (Requirement **(EH2)**), $ks_2 \approx_L ks_1$

From $ks_2 = (\mathcal{V}; (\sigma, \mathsf{skip}, C, \cdot); H) :: ks' :: ks_1'$, we know $\mathsf{consumer}(ks_2)$

Then $\mathcal{E}$ must end in O-NEXT

From $ks_1' \not\approx_L \cdot$, it must be the case that $ks_C = (ks' :: ks_1') \neq \cdot$

Then $\neg\mathsf{consumer}(K_C)$, but this contradicts $\mathsf{consumer}(K_C)$, so this case holds vacuously

Subcase ii: $\mathcal{D}'$ ends in PTOC

By a similar argument to **Subcase i**:

$ks_2 = (\mathcal{V}; (\sigma, \mathsf{skip}, C, \cdot); H) :: ks_1'$. Then, $\mathcal{E}$ must end in O-NEXT and $ks_1' \not\approx_L \cdot$ means $ks_C \neq \cdot$ which contradicts $\mathsf{consumer}(K_C)$, so this case holds vacuously.

Subcase iii: $\mathcal{D}'$ ends in P

By assumption, the resulting ks will be in H producer state. Then, $\mathcal{E}$ must end in O, O-SKIP,

or O-OTHER, which contradicts $\mathsf{consumer}(K_C)$ since the only rule to shrink the ks is O-NEXT.

Subcase iv: $\mathcal{D}'$ ends in SME-H

If the resulting κ_H is in consumer state, the rest of the proof is similar to **Subcase i** or **Subcase ii**.

Otherwise, the resulting κ_H is in producer state and the rest of the proof is similar to **Subcase iii**.

Subcase v: $\mathcal{D}'$ ends in SME-L, SME-LTOH, or P-F

In all of these cases, $pc \sqsubseteq L$, which contradicts $pc = H$, so these cases hold vacuously.

Case III: $\mathcal{D}$ ends in O-NEXT

By assumption, $ks_1 = (\mathcal{V}; \kappa; pc) :: ks_2$

By assumption and from $\mathsf{highProducer}(K_1)$, $pc = H$ and $(G, \mathcal{P} \vdash K_1 \Longrightarrow K_2) \downarrow_L = \cdot$

By assumption and from $ks_1 = (\mathcal{V}; \kappa; pc) :: ks_2$ and $pc = H$, $K_1 \approx_L K_2$

From all of this and $ks_1' \not\approx_L \cdot$, it must be the case that $ks_2 \not\approx_L \cdot$

Then, $\mathsf{producer}(K_2)$ and from this, either

$\mathsf{lowProducer}(K_2)$, in which case the desired conclusion holds.

Otherwise, $\mathsf{highProducer}(K_2)$, in which case the proof proceeds similarly to **Case II.iii**.

Inductive Case: $\mathsf{len}(T) = n + 1$ for $n \geq 2$

By assumption, $T = G, \mathcal{P} \vdash K_1 \Longrightarrow K \Longrightarrow^* K_C$ with $\mathsf{len}(K \Longrightarrow^* K_C) = n$

Denote $\mathcal{D} :: K_1 \Longrightarrow K$

Case I: $\mathcal{D}$ ends in an input rule or O-NEXT

By assumption, $\mathsf{consumer}(K_1)$, but this contradicts $\mathsf{highProducer}(K_1)$, so this case holds vacuously

Case II: $\mathcal{D}$ ends in O, O-SKIP, or O-OTHER

By assumption and from the structure of the operational semantics,

$\exists \mathcal{D}' :: G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_1^G, \kappa \xrightarrow{\alpha}_{pc} \sigma^G, ks'$ with all of the following:

$ks_1 = (\mathcal{V}; \kappa; pc) :: \kappa_1'$, $ks = ks' :: ks_1'$, and $\sigma^G = \sigma_1^G$

From $ks_1 = (\mathcal{V}; \kappa; pc) :: \kappa_1'$, $ks_1 \not\approx_L \cdot$ and $\mathsf{highProducer}(K_1)$ we both of the following:

$pc = H$ and $ks_1' \not\approx_L \cdot$

By assumption and from $pc = H$ and the definitions of $\mathsf{outCondition}$ and output,

$(G, \mathcal{P} \vdash K_1 \Longrightarrow K) \downarrow_L = \cdot$

From $ks_1 = (\mathcal{V}; \kappa; pc) :: \kappa_1'$, $ks = ks' :: ks_1'$, and $\sigma^G = \sigma_1^G$, we know that $K_1 \approx_L K$

From $pc = H$ and by a similar argument as the subcases for **Base Case II**, above, $\mathsf{highProducer}(K)$

Then from $ks_1 \not\approx_L \cdot$ and Lemma 182 (Requirement **(EH2)**), the IH may be applied on

$G, \mathcal{P} \vdash K \Longrightarrow^* K_C$

The conclusion follows from the IH and $(G, \mathcal{P} \vdash K_1 \Longrightarrow K) \downarrow_L = \cdot$ and $K_1 \approx_L K$

Case III: $\mathcal{D}$ ends in O-Next

By assumption and from the structure of the operational semantics, we know all of the following:

$ks_1 = (\mathcal{V};\kappa;pc) :: \kappa_1'$, $ks = ks_1'$, and $\sigma^G = \sigma_1^G$

From $ks_1 = (\mathcal{V};\kappa;pc) :: \kappa_1'$, $ks_1 \not\approx_L \cdot$, and $\mathsf{highProducer}(K_1)$, we know both of the following:

$pc = H$ and $ks_1' \not\approx_L \cdot$

By assumption and from $pc = H$, $(G,\mathcal{P} \vdash K_1 \Longrightarrow K) \downarrow_L = \cdot$

From $ks_1 = (\mathcal{V};\kappa;pc) :: \kappa_1'$, $ks = ks_1'$, and $\sigma^G = \sigma_1^G$, we know that $K_1 \approx_L K$

And from $ks = ks_1'$ and $ks_1' \not\approx_L \cdot$, we know that $\mathsf{producer}(K)$

Subcase i: $\mathsf{lowProducer}(K)$

Then, let $K_2 = K$ and from $(G,\mathcal{P} \vdash K_1 \Longrightarrow K) \downarrow_L = \cdot$ and $K_1 \approx_L K$, the desired conclusion holds

Subcase ii: $\mathsf{highProducer}(K)$

Then, the IH may be applied on $G,\mathcal{P} \vdash K \Longrightarrow^* K_C$ and the desired conclusion follows from the IH, $(G,\mathcal{P} \vdash K_1 \Longrightarrow K) \downarrow_L = \cdot$ and $K_1 \approx_L K$

□

Requirement (T4) Strong one-step

Lemma 133 (Strong One-step). *If $K_1 \approx_L K_2$, $T_1 = G,\mathcal{P} \vdash K_1 \overset{\alpha_{I,1}}{\Longrightarrow} K_1'$ with $T_1 \downarrow_L \neq \cdot$, $\neg\mathsf{rlsA}(T_1)$, and $\mathsf{prog}(K_2)$, then $\exists K_2', T_2$ s.t. $T_2 = G,\mathcal{P} \vdash K_2 \Longrightarrow^* K_2'$ with $T_1 \approx_L T_2$ and $K_1' \approx_L K_2'$*

Proof.

We examine each case of $\mathcal{E} :: T_1 = G,\mathcal{P} \vdash K_1 \overset{\alpha_{I,1}}{\Longrightarrow} K_1'$

Denote $K_i = (\mathcal{R}_i, d_i, \sigma_i, ks_i)$ for $i \in \{1,2\}$

We refer to the following assumptions throughout:

(1) $T_1 \downarrow_L \neq \cdot$; (2) $\neg\mathsf{rlsA}(T_1)$; (3) $K_1 \approx_L K_2$; and (4) $\mathsf{prog}(K_2)$

Case I: $\mathcal{E}$ ends in I-NR1 or I-NR2

In both of these cases $T_1 \downarrow_L = \cdot$ or $T_1 \downarrow_L = \mathsf{rls}(...)$. This contradicts (1) and (2), so this case holds vacuously.

Case II: $\mathcal{E}$ ends in I-R-Diff or I-R-Same

In both of these cases, $\mathsf{rlsA}(T_1)$ This contradicts (2), so this case holds vacuously.

Case III: $\mathcal{E}$ ends in I-L

By assumption, $\mathcal{P}(\alpha_I) = L$, $\sigma_1' = \sigma_1$, and $G,\mathcal{P},\sigma_1 \vdash \cdot; \mathsf{lookupEH}(id.\mathsf{Ev}(v)) \rightsquigarrow . \ ks_1'$

Subcase i: $\mathsf{consumer}(K_2)$

By assumption and from $\mathcal{P}(\alpha_I) = L$, I-L may be applied to K_2 with input $id.\mathsf{Ev}(v)$, producing trace

$T_2 = G, \mathcal{P} \vdash K_2 \overset{id.\mathsf{Ev}(v)}{\Longrightarrow} K_2'$

Then, $G, \mathcal{P}, \sigma_2 \vdash \cdot; \mathsf{lookupEHAll}(id.\mathsf{Ev}(v)) \rightsquigarrow . \; ks_2'$ and $\sigma_2' = \sigma_2$

By assumption and from $\mathcal{P}(\alpha_I) = L$, $T_1 \downarrow_L = T_2 \downarrow_L$

From (3), $\sigma_1' = \sigma_1$, and $\sigma_2' = \sigma_2$, we know that $\sigma_1' \approx_L \sigma_2'$

From (3) and Lemma 172 (Requirement **(EH1)**), $ks_1' \approx_L ks_2'$

Then $K_1' \approx_L K_2'$

Subcase ii: $\neg\mathsf{consumer}(K_2)$

From (4), $\exists T'$ s.t. $T' = G, \mathcal{P} \vdash K_2 \overset{\tau}{\Longrightarrow}^* K_C$ where $\mathsf{cstate}(K_C)$ and $\forall \alpha \in \tau, \alpha \in \mathsf{output}(\tau)$

By assumption and from (3), $ks_2 \approx_L \cdot$

Then from Lemma 131 (Requirement **(T3)**), $K_C \approx_L K_2$ and $T' \downarrow_L = \cdot$

The rest of the proof proceeds the same as **Subcase i**

Case IV: $\mathcal{E}$ ends in O

By assumption, all of the following:

$ks_1 = (\mathcal{V}_1; \kappa_1; pc_1) :: ks_1''$, $\exists \mathcal{E}' :: G, \mathcal{P}, \mathcal{V}_1, d \vdash \sigma_1, \kappa_1 \xrightarrow{ch(v_1)}_{pc_1} \sigma_1', ks_1'''$, $ks_1' = ks_1''' :: ks_1''$, $\mathsf{producer}(\kappa_1)$,

$\mathsf{outCondition}_{\mathcal{V}_1}(\mathcal{P}, ch(v_1), pc_1) = \mathsf{true}$, and $\alpha_{l,1} = \mathsf{output}(\mathcal{P}, ch(v_1), pc_1)$

From $\alpha_{l,1} = \mathsf{output}(\mathcal{P}, ch(v_1), pc_1)$ and the definition of output: if $pc_1 = H$, then $T_1 \downarrow_L = \cdot$, which contradicts (1). Therefore, it must be the case that $pc_1 \sqsubseteq L$

Then, by assumption and from (3), $ks_2 = (\mathcal{V}_2; \kappa_2; pc_2) :: \kappa_2''$

Subcase i: $pc_2 \sqsubseteq L$

By assumption and from (3), $\mathcal{V}_1 = \mathcal{V}_2$, $\kappa_1 \approx_L \kappa_2$, and $ks_1'' \approx_L ks_2''$

From (2), (3), and Lemma 134, $\exists \mathcal{D}' :: G, \mathcal{P}, \mathcal{V}_2, d \vdash \sigma_2, \kappa_2 \xrightarrow{\alpha_2}_{pc_2} \sigma_2', ks_2'''$ with all of the following:

$pc_2 \sqsubseteq L$, $\sigma_1' \approx_L \sigma_2'$, $ks_1''' \approx_L ks_2'''$, and $\alpha_2 \approx_L ch(v_1)$

From $\alpha_2 \approx_L ch(v_1)$, $\alpha = ch(v_2)$ with $v_1 \approx_L v_2$

Then from Lemma 136, $\mathsf{outCondition}_{\mathcal{V}_2}(\mathcal{P}, ch(v_2), pc_2) = \mathsf{true}$

Then by assumption and from $ks_1'' \approx_L ks_2''$, O may be applied to K_2, producing trace

$T_2 = G, \mathcal{P} \vdash K_2 \overset{\alpha_{2,l}}{\Longrightarrow} K_2'$

Then $ks_1'' \approx_L ks_2''$ and $ks_1''' \approx_L ks_2'''$ gives $ks_2' = ks_2''' :: ks_2''$ with $ks_1' \approx_L ks_2'$

From this and $\sigma_1' \approx_L \sigma_2'$, $K_1' \approx_L K_2'$

From the trace ending in O, we also know $\alpha_{l,2} = \mathsf{output}(\mathcal{P}, ch(v_2), pc_2)$

Then from $v_1 \approx_L v_2$ and Lemma 138, $\alpha_{l,1} \approx_L \alpha_{l,2}$

Then from the definition of equivalence for execution traces, $T_1 \approx_L T_2$

Subcase ii: $pc_2 \not\sqsubseteq L$

From (4), $\exists T', K_C$ s.t. $T' = G, \mathcal{P} \vdash K_2 \Longrightarrow^* K_C$ and $\mathsf{consumer}(K_C)$

Then from Lemma 132 (Requirement **(T3)**), $\exists K'$ s.t. $K_2 \Longrightarrow^* K'$ and all of the following:

$\mathsf{lowProducer}(K')$, $K' \approx_L K_2$, and $(G, \mathcal{P} \vdash K_2 \Longrightarrow^* K') \downarrow_L = \cdot$

The rest of the proof proceeds the same as **Subcase i**. Trace equivalence uses

$(G, \mathcal{P} \vdash K_2 \Longrightarrow^* K') \downarrow_L = \cdot$ and state equivalence uses $K' \approx_L K_2$

Case V: $\mathcal{E}$ ends in O-Skip

The proof for this case is similar to **Case IV** except that it uses Lemma 137 instead of Lemma 136.

Case VI: $\mathcal{E}$ ends in O-Other

The proof for this case is similar to **Case V**. $\alpha_2 = \bullet$ follows from Lemma 134, which tells us that $\alpha_1 \approx_L \alpha_2$ and $\alpha_1 = \bullet$ by assumption.

Case VII: $\mathcal{E}$ ends in O-Next

The proof for this case is straightforward. From (6), $ks_1 = (\mathcal{V}_1; \kappa_1; pc_1) :: ks_1''$ and $ks_2 = (\mathcal{V}_2; \kappa_2; pc_2) :: ks_2''$ with $\mathcal{V}_1 = \mathcal{V}_2$ and $\kappa_1 \approx_L \kappa_2$ and $ks_1'' \approx_L ks_2''$ when $pc_2 \sqsubseteq L$. Then, from $\kappa_1 \approx_L \kappa_2$, $\mathsf{consumer}(\kappa_2)$, and O-Next can be applied to K_2, which gives $T_1 \approx_L T_2$. $K_1' \approx_L K_2'$ follows from $ks_1'' \approx_L ks_2''$.

When $pc_2 \not\sqsubseteq L$, the proof is similar to **Case IV.b**.

□

Lemma 134 (Strong One-Step - Single Execution Semantics). *If $G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_1^G, \kappa_1 \xrightarrow{\alpha_1}_{pc_1} \sigma_3^G, ks_1$ with $pc_1 \sqsubseteq L$, $\alpha_1 \neq \mathsf{declassify}(\iota, v)$, $\sigma_1^G \approx_L \sigma_2^G$, $\kappa_1 \approx_L \kappa_2$, and $pc_2 \sqsubseteq L$, then $G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_2^G, \kappa_2 \xrightarrow{\alpha_2}_{pc_2} \sigma_4^G, ks_2$ with $\sigma_3^G \approx_L \sigma_4^G$, $ks_1 \approx_L ks_2$, and $\alpha_1 \approx_L \alpha_2$*

Proof (sketch): By induction on the structure of $\mathcal{E} :: G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_1^G, \kappa_1 \xrightarrow{\alpha_1}_{pc_1} \sigma_3^G, ks_1$. The proof also uses Lemma 135, Lemma 182 (Requirement **(EH2)**), and Lemma 172 (Requirement **(EH1)**). □

Lemma 135 (Strong One-Step - Command Semantics). *If $pc_1, pc_2 \sqsubseteq L$ and $G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1, c \xrightarrow{\alpha_1}_{pc_1} \sigma_3^G, \sigma_1', c_1, E_1$, with $\alpha_1 \neq \mathsf{declassify}(\iota, v)$, and $\sigma_1^G \approx_L \sigma_2^G$, and $\sigma_1 \approx_L \sigma_2$, then $G, \mathcal{V}, d \Vdash \sigma_2^G, \sigma_2, c \xrightarrow{\alpha_2}_{pc_2} \sigma_4^G, \sigma_2', c_2, E_2$ with $\sigma_3^G \approx_L \sigma_4^G$, $\sigma_1' \approx_L \sigma_2'$, $\alpha_1 \approx_L \alpha_2$, $c_1 \approx_L c_2$, and $E_1 \approx_L E_2$.*

Proof (sketch): By induction on the structure of $\mathcal{E} :: G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1, c \xrightarrow{\alpha}_{pc_1} \sigma_3^G, \sigma_2, c_1, E_1$. The proof also uses Lemmas 152, 152.U, 152.T (Requirement **(E1)**), Lemmas 160.U and 160.T (Requirement **(V3)**), Lemmas 196.U, 196.T, 197.U, and 197.T (Requirement **(EH4)**), and Lemma 179 (Requirement **(EH1)**). □

Lemma 136. *If $v_1 \approx_L v_2$ and $pc_1, pc_2 \sqsubseteq L$, with $\mathsf{outCondition}_{\mathcal{V}}(\mathcal{P}, ch(v_1), pc_1) = \mathsf{true}$, then $\mathsf{outCondition}_{\mathcal{V}}(\mathcal{P}, ch(v_2), pc_2) = \mathsf{true}$*

Lemma 137. *If $v_1 \approx_L v_2$ and $pc_1, pc_2 \sqsubseteq L$, with $\mathsf{outCondition}_{\mathcal{V}}(\mathcal{P}, ch(v_1), pc_1) = \mathsf{false}$, then either:*

1. $\mathsf{outCondition}_{\mathcal{V}}(\mathcal{P}, ch(v_2), pc_2) = \mathsf{false}$, *or*

2. $\mathsf{outCondition}_{\mathcal{V}}(\mathcal{P}, ch(v_2), pc_2) = \mathsf{true}$ *and* $v_2 = \langle _|_ \rangle$

Lemma 138. *If* $v_1 \approx_L v_2$ *and* $pc_1, pc_2 \sqsubseteq L$ *with* $\alpha_{I,1} = \mathsf{output}(\mathcal{P}, ch(v_1), pc_1)$ *and* $\alpha_{I,2} = \mathsf{output}(\mathcal{P}, ch(v_2), pc_2)$, *then* $\alpha_{I,1} \approx_L \alpha_{I,2}$

Requirement (WT4) Strong one-step (Weak Secrecy)

Lemma 139 (Strong One-step, Weak Secrecy). *If* $K_1 \approx_L K_2$ *and* $T_1 = G, \mathcal{P} \vdash K_1 \overset{\alpha_{I,1}}{\Longrightarrow} K_1'$ *with* $T_1 \downarrow_L = \alpha'$, $\neg\mathsf{rlsA}(T_1)$, *and* $\mathsf{prog}(K_2)$, $\mathsf{wkT}(K_2, \alpha')$, *then* $\exists K_2', T_2, \alpha_{I,2}$ *s.t.* $T_2 = G, \mathcal{P} \vdash K_2 \Longrightarrow^* K_2'$ *with* $T_1 \approx_L T_2$ *and* $K_1' \approx_L K_2'$

Proof.

We examine each case of $\mathcal{E} :: T_1' = G, \mathcal{P} \vdash K_1 \overset{\alpha_{I,1}}{\Longrightarrow} K_1'$

We refer to the following assumptions throughout:

(1) $K_1 \approx_L K_2$; (2) $T_1 \downarrow_L = \alpha'$; (3) $\neg\mathsf{rlsA}(T_1)$; (4) $\mathsf{prog}(K_2)$; and (5) $\mathsf{wkT}(K_2, \alpha')$

Case I: $\mathcal{E}$ ends in I-NR1 or I-NR2

In both of these cases $T_1 \downarrow_L = \cdot$ or $T_1 \downarrow_L = \mathsf{rls}(...)$. This contradicts (2) and (3), so this case holds vacuously.

Case II: $\mathcal{E}$ ends in I-R-Diff or I-R-Same

In both of these cases, $\mathsf{rlsA}(T_1)$. This contradicts (2), so this case holds vacuously.

Case III: $\mathcal{E}$ ends in I-L

By assumption, all of the following: $\mathcal{P}(\alpha_I) = L$, $\sigma_1' = \sigma_1$, and $G, \mathcal{P}, \sigma_1 \vdash \cdot; \mathsf{lookupEH}(id.\mathsf{Ev}(v)) \rightsquigarrow. ks_1'$

Subcase i: $\mathsf{consumer}(K_2)$

By assumption and from $\mathcal{P}(\alpha_I) = L$, I-L may be applied to K_2 with input $id.\mathsf{Ev}(v)$, producing trace $T_2 = G, \mathcal{P} \vdash K_2 \overset{id.\mathsf{Ev}(v)}{\Longrightarrow} K_2'$

Then, $G, \mathcal{P}, \sigma_2 \vdash \cdot; \mathsf{lookupEHAll}(id.\mathsf{Ev}(v)) \rightsquigarrow. ks_2'$ and $\sigma_2' = \sigma_2$

Then from the definition of L-projection for execution traces, $T_1 \downarrow_L = T_2 \downarrow_L$

From (3), $\sigma_1' = \sigma_1$, and $\sigma_2' = \sigma_2$, we know that $\sigma_1' \approx_L \sigma_2'$

From Lemma 172 (Requirement **(EH1)**), $ks_1' \approx_L ks_2'$

Therefore, $K_1' \approx_L K_2'$

Subcase ii: $\neg\mathsf{consumer}(K_2)$

From (5) and from $\alpha_{I,1} = id.\mathsf{Ev}(v)$, $\exists T', K_{Ip}$ s.t. $T' = G, \mathcal{P} \vdash K_2 \overset{\tau}{\Longrightarrow}^* K_C$ where both of the following:

$\mathsf{cstate}(K_C)$ and $T' \downarrow_L = \cdot$

Then from Lemma 128 (Requirement **(WT2)**), $K_2 \approx_L K_C$ and the rest of the proof proceeds the same

as **Subcase i**.

Case IV: $\mathcal{E}$ ends in O

By assumption, $\exists \mathcal{E}' :: G, \mathcal{P}, \mathcal{V}_1, d \vdash \sigma_1, \kappa_1 \xrightarrow{ch(v_1)}_{pc_1} \sigma_1', ks_1'''$ with all of the following:

$ks_1 = (\mathcal{V}_1; \kappa_1; pc_1) :: ks_1''$, $ks_1' = ks_1''' :: ks_1''$, $\mathsf{producer}(\kappa_1)$, $\mathsf{outCondition}_{\mathcal{V}_1}(\mathcal{P}, ch(v_1), pc_1) = \mathsf{true}$, and

$\alpha_{l,1} = \mathsf{output}(\mathcal{P}, ch(v_1), pc_1)$

From the definition of output: if $pc_1 = H$, then $T_1 \downarrow_L = \cdot$, which contradicts (2). Therefore, it must be that $pc_1 \sqsubseteq L$

Then from (1), $ks_2 = (\mathcal{V}_2; \kappa_2; pc_2) :: ks_2''$

Subcase i: $pc_2 \sqsubseteq L$

From (1), $ks_1 = (\mathcal{V}_1; \kappa_1; pc_1) :: ks_1''$, and $ks_2 = (\mathcal{V}_2; \kappa_2; pc_2) :: ks_2''$,

$\mathcal{V}_1 = \mathcal{V}_2$, $\kappa_1 \approx_L \kappa_2$, and $ks_1'' \approx_L ks_2''$

By assumption and from (5), $\mathsf{wkK}(\kappa_2, \alpha_1)$

Then from (1), (3), $\kappa_1 \approx_L \kappa_2$, and Lemma 141, $\exists \mathcal{D}' :: G, \mathcal{P}, \mathcal{V}_2, d \vdash \sigma_2, \kappa_2 \xrightarrow{\alpha_2}_{pc_2} \sigma_2', ks_2'''$ with all of the following:

$pc_2 \sqsubseteq L$, $\sigma_1' \approx_L \sigma_2'$, $ks_1''' \approx_L ks_2'''$, and $\alpha_2 \approx_L ch(v_1)$

From $\alpha_2 \approx_L ch(v_1)$, $\alpha = ch(v_2)$ with $v_1 \approx_L v_2$

Then from $\mathsf{outCondition}_{\mathcal{V}_1}(\mathcal{P}, ch(v_1), pc_1) = \mathsf{true}$ and Lemma 136 (Requirement **(T4)**),

$\mathsf{outCondition}_{\mathcal{V}_2}(\mathcal{P}, ch(v_2), pc_2) = \mathsf{true}$

Then O may be applied to K_2, producing trace $T_2 = G, \mathcal{P} \vdash K_2 \xRightarrow{\alpha_{2,l}} K_2'$

Therefore, $ks_2' = ks_2''' :: ks_2''$, and from $ks_1'' \approx_L ks_2''$ and $ks_1''' \approx_L ks_2'''$, we know that $ks_1' \approx_L ks_2'$

From this and from $\sigma_1' \approx_L \sigma_2'$, $K_1' \approx_L K_2'$

From $\alpha_{l,2} = \mathsf{output}(\mathcal{P}, ch(v_2), pc_2)$ and Lemma 138 (Requirement **(T4)**), $\alpha_{l,1} \approx_L \alpha_{l,2}$

Then from the definition of equivalence for execution traces, $T_1 \approx_L T_2$

Subcase ii: $pc_2 \not\sqsubseteq L$

From (5) and (IV.6), $\exists T', K_{lp}$ s.t. all of the following:

$T' = G, \mathcal{P} \vdash K_2 \Longrightarrow^* K_{lp}$, $\mathsf{lowProducer}(K_{lp})$, and $T' \downarrow_L = \cdot$

Then from Lemma 128 (Requirement **(WT2)**), $K_2 \approx_L K_{lp}$ and the rest of the proof proceeds the same as **Subcase i**.

Trace equivalence uses $T' \downarrow_L = \cdot$ and state equivalence uses $K_2 \approx_L K_{lp}$.

Case V: $\mathcal{E}$ ends in O-Skip

By assumption, all of the following:

$ks_1 = (\mathcal{V}_1; \kappa_1; pc_1) :: ks_1''$, $\exists \mathcal{E}' :: G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_1, \kappa_1 \xrightarrow{ch(v_1)}_{pc_1}, \sigma_1', ks_1'''$, $\mathsf{producer}(\kappa)$,

$\mathsf{outCondition}_{\mathcal{I}}(\mathcal{P}, ch(v_1), pc_1) = \mathsf{false}$, and $\alpha_{l,1} = (\bullet, pc_1)$

If $pc_1 = H$, then $T_1 \downarrow_L = \cdot$, which contradicts (2). Therefore, it must be the case that $pc_1 \sqsubseteq L$

Then, from (1), $ks_2 = (\mathcal{V}_2; \kappa_2; pc_2) :: \kappa_2''$

Subcase i: $pc_2 \sqsubseteq L$

From (1), $\mathcal{V}_1 = \mathcal{V}_2$, $\kappa_1 \approx_L \kappa_2$, and $ks_1'' \approx_L ks_2''$

Then from (1), (3), $ks_1'' \approx_L ks_2''$, and Lemma 134, $\exists \mathcal{D}' :: G, \mathcal{P}, \mathcal{V}_2, d \vdash \sigma_2, \kappa_2 \xrightarrow{\alpha}_{pc_2} \sigma_2', ks_2'''$ with all of the following:

$pc_2 \sqsubseteq L$, $\sigma_1' \approx_L \sigma_2'$, $ks_1' \approx_L ks_2'$, and $\alpha \approx_L ch(v_1)$

From $\alpha \approx_L ch(v_1)$, $\alpha = ch(v_2)$ with $v_1 \approx_L v_2$

Then from $\mathsf{outCondition}_{\mathcal{I}}(\mathcal{P}, ch(v_1), pc_1) = \mathsf{false}$ and Lemma 137 (Requirement **(T4)**), one of the following must be true:

$\mathsf{outCondition}_{\mathcal{I}}(\mathcal{P}, ch(v_2), pc_2) = \mathsf{false}$ or $\mathsf{outCondition}_{\mathcal{I}}(\mathcal{P}, ch(v_2), pc_2) = \mathsf{true}$ and $v_2 = \langle _ | _ \rangle$

Subsubcase a: $\mathsf{outCondition}_{\mathcal{I}}(\mathcal{P}, ch(v_2), pc_2) = \mathsf{false}$ is true

By assumption, O-Skip may be applied to K_2, producing trace $T_2 = G, \mathcal{P} \vdash K_2 \overset{(\bullet, pc_2)}{\Longrightarrow} K_2'$

Then, from $\sigma_1' \approx_L \sigma_2'$ and $ks_1' \approx_L ks_2'$, we know that $K_1' \approx_L K_2'$

From the trace ending in O-Skip, $\alpha_{l,2} = (\bullet, pc_2)$

Then from the definition of equivalence for execution traces, $T_1 \approx_L T_2$

Subsubcase b: $\mathsf{outCondition}_{\mathcal{I}}(\mathcal{P}, ch(v_2), pc_2) = \mathsf{true}$ and $v_2 = \langle _ | _ \rangle$ is true

By assumption, O may be applied to K_2, producing trace $T_2 = G, \mathcal{P} \vdash K_2 \overset{\alpha_{l,2}}{\Longrightarrow} K_2'$

Then, from $\sigma_1' \approx_L \sigma_2'$ and $ks_1' \approx_L ks_2'$, we know that $K_1' \approx_L K_2'$

From $\mathsf{outCondition}_{\mathcal{I}}(\mathcal{P}, ch(v_1), pc_1) = \mathsf{false}$ and the definition of output, either:

$pc_1 = L$ and $\mathcal{P}(ch) = H$ or $pc_1 = \cdot$ and $v_1 \downarrow_{\mathcal{P}(ch)} = \cdot$

By assumption and from the structure of the operational semantics, $pc_2 = \cdot$ and $\mathcal{V}_2 = \mathsf{MF}$

If $pc_1 = L$ and $\mathcal{P}(ch) = H$ is true, then from the definition of output,

$\mathsf{output}(\mathcal{P}, ch(v_2), pc_2) = \langle ch(\mathsf{getFacet}(v_2, H)) | \bullet \rangle$

On the other hand, if $pc_1 = \cdot$ and $v_1 \downarrow_{\mathcal{P}(ch)} = \cdot$ is true, then $\mathcal{P}(ch) = H$ (because otherwise outCondition would have been false) and $\mathsf{output}(\mathcal{P}, ch(v_2), pc_2) = \langle ch(\mathsf{getFacet}(v_2, H)) | \bullet \rangle$

From $\alpha_{l,1} = (\bullet, pc_1)$ and the definition of equivalence for execution traces, $T_1 \approx_L T_2$

Subcase ii: $pc_2 \not\sqsubseteq L$

The proof for this case uses similar logic as **Subcase IV.b** to reach the assumptions for **Subcase i**, at which point the proof proceeds the same as **Subcase i**.

Case VI: $\mathcal{E}$ ends in O-Other

The proof for this case is similar to **Case V**. $\alpha_2 = \bullet$ follows from Lemma 141, which tells us that $\alpha_1 \approx_L \alpha_2$ and $\alpha_1 = \bullet$ by assumption.

Case VII: $\mathcal{E}$ ends in O-NEXT

The proof for this case is straightforward. From (1), $ks_1 = (\mathcal{V}_1; \kappa_1; pc_1) :: ks_1''$ and $ks_2 = (\mathcal{V}_2; \kappa_2; pc_2) :: ks_2''$ with $\mathcal{V}_1 = \mathcal{V}_2$ and $\kappa_1 \approx_L \kappa_2$ and $ks_1'' \approx_L ks_2''$ when $pc_2 \sqsubseteq L$. Then, from $\kappa_1 \approx_L \kappa_2$, $\mathsf{consumer}(\kappa_2)$, and O-NEXT can be applied to K_2, which gives $T_1 \approx_L T_2$. $K_1' \approx_L K_2'$ follows from $ks_1'' \approx_L ks_2''$. When $pc_2 \not\sqsubseteq L$, the proof is similar to **Case IV.b**.

□

Definition 140. $\mathsf{wkK}(\kappa, \alpha)$ *for the mid-level semantics is similar to* wkT *for execution traces: it says that when* $\alpha = \mathsf{br}(b)$*, the mid-level semantics can take a step from* κ *producing the same action.*

Lemma 141 (Strong One-Step - Single Execution Semantics, Weak Secrecy). *If* $G, \mathcal{P}, \mathcal{V}, d \vdash_w \sigma_1^G, \kappa_1 \xrightarrow{\alpha_1}_{pc_1} \sigma_3^G, ks_1$ *with* $pc_1 \sqsubseteq L$*,* $\alpha_1 \neq \mathsf{declassify}(\iota, v)$*,* $\sigma_1^G \approx_L \sigma_2^G$*,* $\kappa_1 \approx_L \kappa_2$*,* $\mathsf{wkK}(\kappa_2, \alpha_1)$ *and* $pc_2 \sqsubseteq L$*, then* $G, \mathcal{P}, \mathcal{V}, d \vdash_w \sigma_2^G, \kappa_2 \xrightarrow{\alpha_2}_{pc_2} \sigma_4^G, ks_2$ *with* $\sigma_3^G \approx_L \sigma_4^G$*,* $ks_1 \approx_L ks_2$*, and* $\alpha_1 \approx_L \alpha_2$

Proof (sketch): The proof is the same as for Lemma 134 (Requirement **(T4)**) except that it uses Lemma 143 instead of Lemma 135. The proof uses Lemma 182 (Requirement **(EH2)**) for the LC case. $\mathsf{wkK}(\kappa_2, \alpha_1)$ is used to show $\mathsf{wkC}(c_2, \sigma_2^G, \sigma_2, \alpha_1)$ for the command semantics. □

Definition 142. $\mathsf{wkC}(c, \sigma^G, \sigma, \alpha)$ *for the command semantics is similar to* wkT *for execution traces and* wkK *for mid-level semantics: it says that when* $\alpha = \mathsf{br}(b)$*, the command* c *can take a step under* σ^G *and* σ*, producing the same action*

Lemma 143 (Strong One-Step - Command Semantics, Weak Secrecy). *If* $pc_1, pc_2 \sqsubseteq L$ *and* $G, \mathcal{V}, d \Vdash_w \sigma_1^G, \sigma_1, c \xrightarrow{\alpha_1}_{pc_1} \sigma_3^G, \sigma_1', c_1, E_1$*, with* $\alpha_1 \neq \mathsf{declassify}(\iota, v)$*,* $\mathsf{wkC}(c, \sigma_2^G, \sigma_2, \alpha_1)$ *and* $\sigma_1^G \approx_L \sigma_2$*, and* $\sigma_1 \approx_L \sigma_2^G$*, then* $G, \mathcal{V}, d \Vdash_w \sigma_2^G, \sigma_2, c \xrightarrow{\alpha_2}_{pc_2} \sigma_4^G, \sigma_2', c_2, E_2$ *with* $\sigma_3^G \approx_L \sigma_4^G$*,* $\sigma_1' \approx_L \sigma_2'$*,* $\alpha_1 \approx_L \alpha_2$*,* $c_1' \approx_L c_2'$*, and* $E_1 \approx_L E_2$*.*

Proof.

By induction on the structure of $\mathcal{E} :: G, \mathcal{V}, d \Vdash_w \sigma_1^G, \sigma_1, c \xrightarrow{\alpha_1}_{pc_1} \sigma_3^G, \sigma_1', c_1, E_1$,

The proof is similar to the one for Lemma 135 (Requirement **(T4)**). The most noteworthy differences are shown below:

We refer to the following assumptions throughout:

(1) $pc_1, pc_2 \sqsubseteq L$; (2) $\alpha_1 \neq \mathsf{declassify}(\iota, v)$; (3) $\mathsf{wkC}(c, \sigma_2^G, \sigma_2, \alpha_1)$; (4) $\sigma_1^G \approx_L \sigma_2$; (5) $\sigma_1 \approx_L \sigma_2$

Case I: $\mathcal{E}$ ends in IF-TRUE

The cases for IF-FALSE, WHILE-TRUE, and WHILE-FALSE when $\mathcal{V} = \mathsf{TT}$ are similar.

The proofs for the cases where $\mathcal{V} \neq \mathsf{TT}$ are the same as for Lemma 135 since the rules are unchanged in those cases.

By assumption, all of the following:

$c = \mathsf{if}\ e\ \mathsf{then}\ c_1'\ \mathsf{else}\ c_2'$, $c_1 = c_1'$, $E_1 = \cdot$, $\sigma_3^G = \sigma_1^G$, $\sigma_1' = \sigma_1$, $\alpha_1 = \mathsf{brOutput}((\mathsf{true}, l_1))$, and

$G, \mathcal{V}, \sigma_1^G, \sigma_1 \vdash e \Downarrow_{pc_1}^{\mathsf{TT}} (\mathsf{true}, l_1)$

Then from (1), (4), (5), and Lemma 153 (Requirement **(WE1)**), $G, \mathcal{V}, \sigma_2^G, \sigma_2 \vdash e \Downarrow_{pc_2}^{\mathsf{TT}} v_2$ with $v_2 \approx_L (\mathsf{true}, l_1)$

From this, either $v_2 = (\mathsf{true}, l_2)$ and $l_1 = l_2$ or $v_2 = (\mathsf{false}, H)$ and $l_1 = H$

Subcase i: $v_2 = (\mathsf{true}, l_2)$ and $l_1 = l_2$ is true

Applying IF-TRUE produces trace $G, \mathcal{V}, d \Vdash_w \sigma_2^G, \sigma_2, x := e \xrightarrow{\alpha_2}_{pc_2} \sigma_2^G, \sigma_2, c_1', \cdot$ with all of the following:

$\sigma_4^G = \sigma_2^G$, $\sigma_2' = \sigma_2$, $c_2 = c_1'$, $E_2 = \cdot$, and $\alpha_2 = \mathsf{brOutput}((\mathsf{true}, l_2))$

Then from (4), $\sigma_4^G = \sigma_2^G$, and $\sigma_3^G = \sigma_1^G$, we know that $\sigma_3^G \approx_L \sigma_4^G$

From (5), $\sigma_2' = \sigma_2$, and $\sigma_1' = \sigma_1$, we know that $\sigma_1' \approx_L \sigma_2'$

From $\alpha_1 = \mathsf{brOutput}((\mathsf{true}, l_1))$, $\alpha_2 = \mathsf{brOutput}((\mathsf{true}, l_2))$, and the definition of $\mathsf{brOutput}$, either:

$\alpha_1 = \alpha_2 = \bullet$ (if $l_1 = l_2 = L$) or $\alpha_1 = \alpha_2 = \mathsf{br}(\mathsf{true})$ (if $l_1 = l_2 = H$)

Then, in either case $\alpha_1 \approx_L \alpha_2$

From $c_1 = c_1'$ and $c_2 = c_1'$, we know that $c_1 \approx_L c_2$

From $E_1 = \cdot$ and $E_2 = \cdot$, we know that $E_1 \approx_L E_2$

Subcase ii: $v_2 = (\mathsf{false}, H)$ and $l_1 = H$ is true

Applying IF-FALSE produces trace $G, \mathcal{V}, d \Vdash \sigma_2^G, \sigma_2, x := e \xrightarrow{\alpha_2}_{pc_2} \sigma_2^G, \sigma_2, c_2, \cdot$ with

$\alpha_2 = \mathsf{brOutput}((\mathsf{false}, H))$

From $\alpha_1 = \mathsf{brOutput}((\mathsf{true}, l_1))$, $\alpha_2 = \mathsf{brOutput}((\mathsf{true}, l_2))$, and the definition of $\mathsf{brOutput}$, we know that $\alpha_1 = \mathsf{br}(\mathsf{true})$ and $\alpha_2 = \mathsf{br}(\mathsf{false})$

But this contradicts (3), so this case holds vacuously

Case II: $\mathcal{E}$ ends in ASSIGN-G

The cases for ASSIGN-D and CREATEELEM when $\mathcal{V} = \mathsf{TT}$ are similar to the case from Lemma 135 (Requirement **(T4)**), except that it uses Lemma 161 (Requirement **(WV3)**) instead of Lemma 160 (Requirement **(V3)**) for ASSIGN-G, Lemma 198 (Requirement **(WEH4)**) instead of Lemma 196 (Requirement **(EH4)**) for ASSIGN-D, and Lemma 199 (Requirement **(WEH4)**) instead of Lemma 197 (Requirement **(EH4)**) for CREATEELEM. The proofs for the cases where $\mathcal{V} \neq \mathsf{TT}$ are the same as for Lemma 135 (Requirement **(T4)**) since the rules always return $\bullet$ in those cases, meaning they are effectively the same rules.

□

Requirement (T5) Weak one-step

Lemma 144 (Weak One-Step). *If $T_1 = G, \mathcal{P} \vdash K_1 \overset{\alpha_{l,1}}{\Longrightarrow} K_1'$ and $T_2 = G, \mathcal{P} \vdash K_2 \overset{\alpha_{l,2}}{\Longrightarrow} K_2'$, with $T_1 \approx_L T_2$, $K_1 \approx_L K_2$, $T_1 \downarrow_L \neq \cdot$, and $T_2 \downarrow_L \neq \cdot$, then $K_1' \approx_L K_2'$*

Proof.

We examine each case of $\mathcal{E} :: G, \mathcal{P} \vdash K_1 \overset{\alpha_{l,1}}{\Longrightarrow} K_1'$. Denote $\mathcal{D} :: G, \mathcal{P} \vdash K_2 \overset{\alpha_{l,2}}{\Longrightarrow} K_2'$.

We refer to the following assumptions throughout:

(1) $K_1 \approx_L K_2$; (2) $T_1 \approx_L T_2$; (3) $T_1 \downarrow_L \neq \cdot$; and (4) $T_2 \downarrow_L \neq \cdot$

Case I: $\mathcal{E}$ ends in I-NR1

By assumption, $G, \mathcal{P}, \sigma_1 \vdash \cdot; \mathsf{lookupEHAll}(id.\mathsf{Ev}(v)) \rightsquigarrow_H ks_1'$, $\sigma_1' = \sigma_1$, and $\mathcal{R}_1 = (\rho_1, \mathcal{D})$ with all of the following:

$\mathcal{D}(\rho_1, id.\mathsf{Ev}(v)) = (\rho_1', r, \bullet)$, $d_1' = \mathsf{update}(d_1, r)$, and $\mathcal{R}_1' = (\rho_1', \mathcal{D})$

From $\mathcal{R}_1 = (\rho_1, \mathcal{D})$ and (3), $T_1 \downarrow_L = \mathsf{rls}(\rho_1', r, \bullet)$

Then from (2), $\mathcal{D}$ must end in I-NR1 with some input $id'.\mathsf{Ev}'(v')$ producing declassification $\rho_1', r, \bullet$ with all of the following:

$G, \mathcal{P}, \sigma_2 \vdash \cdot; \mathsf{lookupEHAll}(id'.\mathsf{Ev}'(v')) \rightsquigarrow_H ks_2'$, $\sigma_2' = \sigma_2$, and $\mathcal{R}_2 = (\rho_2, \mathcal{D})$ with

$\mathcal{D}(\rho_2, id.\mathsf{Ev}(v)) = (\rho_1', r, \bullet)$, $d_2' = \mathsf{update}(d_2, r)$, and $\mathcal{R}_2' = (\rho_1', \mathcal{D})$

From $\mathcal{R}_1' = (\rho_1', \mathcal{D})$ and $\mathcal{R}_2' = (\rho_1', \mathcal{D})$, we know that $\mathcal{R}_1' = \mathcal{R}_2'$

From (1), $d_1' = \mathsf{update}(d_1, r)$, and $d_2' = \mathsf{update}(d_2, r)$, we know that $d_1' = d_2'$

From (1), $\sigma_1' = \sigma_1$, and $\sigma_2' = \sigma_2$, we know that $\sigma_1' = \sigma_2'$

From Lemma 182, $ks_1' \approx_L \cdot$ and $ks_2' \approx_L \cdot$

Thus, $ks_1' \approx_L ks_2'$

Therefore, $K_1' \approx_L K_2'$

Case II: $\mathcal{E}$ ends in I-NR2

By assumption, $T_1 \downarrow_L = \cdot$, which contradicts (3), so this case holds vacuously.

Case III: $\mathcal{E}$ ends in I-R-Diff

By assumption, $G, \mathcal{P}, \sigma_1 \vdash \cdot; \mathsf{lookupEHAt}(id.\mathsf{Ev}(v')) \rightsquigarrow_L ks_1''$,

$G, \mathcal{P}, \sigma_1 \vdash ks_1''; \mathsf{lookupEHAt}(id.\mathsf{Ev}(v)) \rightsquigarrow_H \kappa_1'$, $\sigma_1' = \sigma_1$, and $\mathcal{R}_1 = (\rho_1, \mathcal{D})$ with all of the following:

$\mathcal{D}(\rho_1, id.\mathsf{Ev}(v)) = (\rho_1', r, v')$, $d_1' = \mathsf{update}(d_1, r)$, and $\mathcal{R}_1' = (\rho_1', \mathcal{D})$

Then, $T_1 \downarrow_L = \mathsf{rls}(id.\mathsf{Ev}(v'))$

Then from (2), $\mathcal{D}$ must end in I-R-Diff or I-R-Same with input $\alpha_{l,2}$ s.t. $T_2 \downarrow_L = \mathsf{rls}(\rho_1', r, id.\mathsf{Ev}(v'))$

Subcase i: $\mathcal{D}$ ends in I-R-Diff

By assumption, all of the following:

$G, \mathcal{P}, \sigma_2 \vdash \cdot; \mathsf{lookupEHAt}(id.\mathsf{Ev}(v')) \rightsquigarrow_L ks_2''$, $G, \mathcal{P}, \sigma_2 \vdash ks_2''; \mathsf{lookupEHAt}(\alpha_{l,2}) \rightsquigarrow_H \kappa_2'$, $\sigma_2' = \sigma_2$, and

$\mathcal{R}_2 = (\rho_2, \mathcal{D})$ with

$\mathcal{D}(\rho_2, id.\mathsf{Ev}(v')) = (\rho_1', r, _)$, $d_2' = \mathsf{update}(d_2, r)$, and $\mathcal{R}_2' = (\rho_1', \mathcal{D})$

From $\mathcal{R}_1' = (\rho_1', \mathcal{D})$ and $\mathcal{R}_2' = (\rho_1', \mathcal{D})$, we know that $\mathcal{R}_1' = \mathcal{R}_2'$

From (1), $d_1' = \mathsf{update}(d_1, r)$, and $d_2' = \mathsf{update}(d_2, r)$, we know that $d_1' = d_2'$

From Lemma 172 (Requirement **(EH1)**), $ks_1'' \approx_L ks_2''$

From Lemma 182 (Requirement **(EH2)**), $ks_1'' \approx_L ks_1'$ and $ks_2'' \approx_L ks_2'$

Therefore, $ks_1' \approx_L ks_2'$

From (1), $\sigma_1' = \sigma_1$, and $\sigma_2' = \sigma_2$, we know that $\sigma_1' \approx_L \sigma_2'$

Thus, $K_1' \approx_L K_2'$

Subcase ii: $\mathcal{D}$ ends in I-R-SAME

The proof is similar to **Subcase i**

Case IV: $\mathcal{E}$ ends in I-R-SAME or I-L

The proof for these cases are similar to **Case III**

Case V: $\mathcal{E}$ ends in O

By assumption, all of the following:

$ks_1 = (\mathcal{V}; \kappa_1; pc_1) :: \kappa_1''$, $\mathsf{producer}(\kappa_1)$, $G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_1, \kappa_1 \xrightarrow{ch(v_1)}_{pc_1} \sigma_1', ks_1'''$, $\alpha_{l,1} = \mathsf{output}(\mathcal{P}, ch(v_1), pc_1)$,

$\mathsf{outCondition}_{\mathcal{V}}(\mathcal{P}, ch(v_1), pc_1) = \mathsf{true}$, and $ks_1' = ks_1''' :: ks_1''$

From (3) and the definition of trace projection, either $pc_1 \sqsubseteq L$ or $\mathcal{P}(ch) = L$ or $\alpha_{l,1} = \langle _ | _ \rangle$ and $\mathsf{getFacet}(\alpha_{l,1}, L) \neq \cdot$

If $\mathcal{P}(ch) = L$is true, then from the definition of output and outCondition: $pc_1 \sqsubseteq L$

If $\alpha_{l,1} = \langle _ | _ \rangle$ and $\mathsf{getFacet}(\alpha_{l,1}, L) \neq \cdot$ is true, then from the definition of output and outCondition: $pc_1 = \cdot$

Then, in any case, $pc_1 \sqsubseteq L$

From (1), $ks_2 = (\mathcal{V}_2; \kappa_2; pc_2) :: ks_2''$

From the operational semantics, either $\mathcal{D}$ ends in O or $\alpha_{l,2} = (_, pc_2)$

If $\mathcal{D}$ ends in O is true, then from (2) and by a similar argument to above: $pc_2 \sqsubseteq L$

If $\alpha_{l,2} = (_, pc_2)$is true, then from (4) and the definition of trace projection: $pc_2 \sqsubseteq L$

Therefore, $pc_2 \sqsubseteq L$

Then, $\mathcal{V}_2 = \mathcal{V}$, $\kappa_1 \approx_L \kappa_2$, and $ks_1'' \approx_L ks_2''$

Subcase i: $\mathcal{D}$ ends in O

By assumption, $\exists \mathcal{D}' :: G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_2, \kappa_2 \xrightarrow{ch(v_2)}_{pc_2} \sigma_2', ks_2'''$ and $ks_2' = ks_2'' :: ks_2'''$

From Lemma 145, $\sigma_1' \approx_L \sigma_2'$ and $ks_1''' \approx_L ks_2'''$

Then from $ks_1'' \approx_L ks_2''$ and $ks_1''' \approx_L ks_2'''$, we know that $ks_1' \approx_L ks_2'$

Therefore, $K_1' \approx_L K_2'$

Subcase ii: $\mathcal{D}$ ends in O-Skip or O-Other

The proofs for these cases are similar to **Subcase i**.

Subcase iii: $\mathcal{D}$ ends in O-Next

By assumption, $\mathsf{consumer}(\kappa_2)$

But this contradicts $\mathsf{producer}(\kappa_1)$ and $\kappa_1 \approx_L \kappa_2$, so this case holds vacuously.

Case VI: $\mathcal{E}$ ends in O-Skip or O-Other

The proof for this case is similar to the proof for **Case IV**. The biggest difference is that $pc_1 \sqsubseteq L$ follows from (3) and the definition of trace projection.

Case VII: $\mathcal{E}$ ends in O-Next

The proof for this case is straightforward. It begins by establishing that $pc_2 \sqsubseteq L$ and the rest follows from (1).

□

Lemma 145 (Weak One-Step - Single Execution Semantics). *If $G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_1^G, \kappa_1 \xrightarrow{\alpha_1}_{pc_1} \sigma_3^G, ks_1$, and $G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_2^G, \kappa_2 \xrightarrow{\alpha_2}_{pc_2} \sigma_4^G, ks_2$ with $pc_1, pc_2 \sqsubseteq L$, $\sigma_1^G \approx_L \sigma_2^G$, and $\kappa_1 \approx_L \kappa_2$, then $\sigma_3^G \approx_L \sigma_4^G$ and $ks_1 \approx_L ks_2$*

Proof (sketch): By induction on the structure of $\mathcal{E} :: G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_1^G, \kappa_1, \xrightarrow{\alpha_1}_{pc_1} \sigma_3^G, ks_1$. The proof also uses Lemma 182 (Requirement **(EH2)**), Lemma 172 (Requirement **(EH1)**), and Lemma 146. □

Lemma 146 (Weak One-Step - Command Semantics). *If $G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1, c \xrightarrow{\alpha_1}_{pc_1} \sigma_3^G, \sigma_1', c_1, E_1$ and $G, \mathcal{V}, d \Vdash \sigma_2^G, \sigma_2, c \xrightarrow{\alpha_2}_{pc_2} \sigma_4^G, \sigma_2', c_2, E_2$, with $pc_1, pc_2 \sqsubseteq L$, $\sigma_1^G \approx_L \sigma_2^G$, and $\sigma_1 \approx_L \sigma_2$, then $\sigma_3^G \approx_L \sigma_4^G$, $\sigma_1' \approx_L \sigma_2'$, $c_1 \approx_L c_2$, and $E_1 \approx_L E_2$*

Proof (sketch): By induction on the structure of $\mathcal{E} :: G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1, c \xrightarrow{\alpha_1}_{pc_1} \sigma_3^G, \sigma_1', c_1, E_1$. The proof also uses Lemmas 179, 152.U, and 152.T (Requirement **(E1)**), Lemmas 160.U and 160.T (Requirement **(V3)**), and Lemmas 196.U, 196.T, 197.U, and 197.T (Requirement **(EH4)**). □

Requirement (WT5) Weak one-step (Weak Secrecy)

Lemma 147 (Weak One-Step, Weak Secrecy). *If $T_1 = G, \mathcal{P} \vdash_w K_1 \stackrel{\alpha_{l,1}}{\Longrightarrow} K_1'$ and $T_2 = G, \mathcal{P} \vdash_w K_2 \stackrel{\alpha_{l,2}}{\Longrightarrow} K_2'$, with $T_1 \approx_L T_2$, $K_1 \approx_L K_2$, $T_1 \downarrow_L \neq \cdot$, and $T_2 \downarrow_L \neq \cdot$, then $K_1' \approx_L K_2'$*

Proof.

We examine each case of $\mathcal{E} :: G, \mathcal{P} \vdash_w K_1 \stackrel{\alpha_{l,1}}{\Longrightarrow} K_1'$. Denote $\mathcal{D} :: G, \mathcal{P} \vdash_w K_2 \stackrel{\alpha_{l,2}}{\Longrightarrow} K_2'$

Case I: $\mathcal{E}$ ends in an input rule

The proofs for these cases are the same as Lemma 144 (Requirement **(T5)**). They use Lemma 172 (Requirement **(EH1)**) and Lemma 182 (Requirement **(EH2)**).

Case II: $\mathcal{E}$ ends in and output rule and $T_1 \downarrow_L = T_2 \downarrow_L = \mathsf{gw}(_)$

Note that the only rules to emit gw(_) are ASSIGN-G-H, ASSIGN-D-H, and CREATEELEM-H, which only run in the H context. The proof for this case follows from Lemma 148.

Case III: $\mathcal{E}$ ends in an output rule O, O-SKIP, or O-OTHER and $T_1 \downarrow_L = T_2 \downarrow_L \neq \mathsf{gw}(_)$

The proof for this case is similar to the output cases for Lemma 144 (Requirement **(T5)**). Note that all other visible outputs still happen in the L context. These cases use Lemma 150 instead of Lemma 145. The biggest difference here is that we need to show $\alpha_1 = \alpha_2$ when $\alpha_1, \alpha_2 \neq ch(_)$ to apply Lemma 150.

Denote $\mathcal{E}' :: G, \mathcal{P}, \mathcal{V}, d \vdash_w \sigma_1, \kappa_1 \xrightarrow{\alpha_1}_{pc_1} \sigma_1', ks_1$ and $\mathcal{D}' :: G, \mathcal{P}, \mathcal{V}, d \vdash_w \sigma_2, \kappa_2 \xrightarrow{\alpha_2}_{pc_2} \sigma_2', ks_2$

Want to show: $\alpha_1 = \alpha_2$ when $\alpha_1, \alpha_2 \neq ch(_)$

Assume that $\alpha_1, \alpha_2 \neq ch(_)$

When $\mathcal{E}$ or $\mathcal{D}$ end in O or O-SKIP, we know that $\alpha_1 = ch(_)$ or $\alpha_2 = ch(_)$

Then, by assumption, $\mathcal{E}$ and $\mathcal{D}$ ends in O-OTHER with $\alpha_{l,1} = (\alpha_1, pc_1)$ and $\alpha_{l,2} = (\alpha_2, pc_2)$

Then, $\alpha_{l,1}, \alpha_{l,2} \neq \langle|\rangle$, $\alpha_{l,1}, \alpha_{l,2} \neq id.\mathsf{Ev}(v)$, and $\alpha_{l,1}, \alpha_{l,2} \neq \mathsf{rls}(_)$

Therefore, from the definition of trace projection, $T_1 \downarrow_L = \alpha_1$ and $T_2 \downarrow_L = \alpha_2$

Then from the definition of equivalence for traces, $\alpha_1 = \alpha_2$

Case IV: $\mathcal{E}$ ends in O-NEXT

The proof for this case is the same as from Lemma 144 (Requirement **(T5)**).

□

Lemma 148 (Weak One-Step - Single Execution Semantics - H context, Weak secrecy). *If $G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_1^G, \kappa_1 \xrightarrow{\alpha_1}_H \sigma_3^G, ks_1$, and $G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_2^G, \kappa_2 \xrightarrow{\alpha_2}_H \sigma_4^G, ks_2$ with $\alpha_1 = \alpha_2 = \mathsf{gw}(_)$, $\sigma_1^G \approx_L \sigma_2^G$, then $\sigma_3^G \approx_L \sigma_4^G$ and $ks_1 \approx_L ks_2$*

Proof (sketch): By induction on the structure of both traces. The only cases where $\alpha_1 = \alpha_2 = \mathsf{gw}(_)$ is when the traces end in P or SME-H. The first case uses Lemma 149 and the second uses the IH. Since the resulting ks is still in the H context in both cases, showing $ks_1 \approx_L ks_2$ is trivial. Lemma 149 is used to show that $\sigma_3^G \approx_L \sigma_4^G$ □

Lemma 149 (Weak One-Step - Command Semantics, Weak Secrecy). *If $G, \mathcal{V}, d \Vdash_w \sigma_1^G, \sigma_1, c_1 \xrightarrow{\alpha_1}_H \sigma_3^G, \sigma_1', c_1, E_1$ and $G, \mathcal{V}, d \Vdash_w \sigma_2^G, \sigma_2, c_2 \xrightarrow{\alpha_2}_H \sigma_4^G, \sigma_2', c_2, E_2$, with $\alpha_1 = \alpha_2 = \mathsf{gw}(_)$ and $\sigma_1^G \approx_L \sigma_2^G$ then $\sigma_3^G \approx_L \sigma_4^G$*

Proof.

By induction on the structure of $\mathcal{E} :: G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1, c_1 \xrightarrow{\alpha_1}_{pc_1} \sigma_3^G, \sigma_1', c_1, E_1$.

Denote $\mathcal{D} :: G, \mathcal{V}, d \Vdash \sigma_2^G, \sigma_2, c_2 \xrightarrow{\alpha_2}_{pc_2} \sigma_4^G, \sigma_2', c_2, E_2$

From $\alpha_1 = \alpha_2 = \mathsf{gw}(_)$ and the assumption that $\mathcal{E}$ and $\mathcal{D}$ are in the H context, $\mathcal{E}$ and $\mathcal{D}$ must end in ASSIGN-G-H, ASSIGN-D-H, or CREATEELEM

Case I: $\mathcal{E}$ ends in ASSIGN-G-H

By assumption, $\mathsf{assignW}_{G\downarrow_g}(\sigma_1^G, H, x, v_1) = (\sigma_3^G, \alpha_1)$

By assumption and from $\alpha_1 = \alpha_2 = \mathsf{gw}(_)$, we know that $\alpha_1 = \mathsf{gw}(x)$ and $\alpha_2 = \mathsf{gw}(x)$

Then, $\mathcal{D}$ must end in ASSIGN-G-H with $\mathsf{assignW}_{G\downarrow_g}(\sigma_2^G, H, x, v_2) = (\sigma_4^G, \alpha_2)$

From Lemma 159 (Requirement **(WV2)**), $\sigma_3^G \approx_L \sigma_4^G$

Case II: $\mathcal{E}$ ends in ASSIGN-D-H

By assumption, $\mathsf{assignW}_{G\downarrow_{EH}}(\sigma_1^G, H, id, v_1) = (\sigma_3^G, \alpha_1)$

By assumption and from $\alpha_1 = \alpha_2 = \mathsf{gw}(_)$, we know that $\alpha_1 = \mathsf{gw}(id)$ and $\alpha_2 = \mathsf{gw}(id)$

Then, $\mathcal{D}$ ends in ASSIGN-D-H or CREATEELEM

Subcase i: $\sigma_1^G(id) = (id, v, M, l)$ with $l \sqsubseteq L$

By assumption and from $\sigma_1^G \approx_L \sigma_2^G$, we know that $id \in \sigma_2^G(id)$

Then, $\mathcal{D}$ must end in ASSIGN-D-H with $\mathsf{assignW}_{G\downarrow_{EH}}(\sigma_2^G, H, id, v_2) = (\sigma_4^G, \alpha_2)$

From Lemma 191 (Requirement **(WEH3)**), $\sigma_3^G \approx_L \sigma_4^G$

Subcase ii: $\sigma_1^G(id) = (id, v, M, l)$ with $l \not\sqsubseteq L$

By assumption and from $\sigma_1^G \approx_L \sigma_2^G$, either

$id \in \sigma_2^G$ with $\sigma_2^G(id) = (id, _, _, H)$ or $id \notin \sigma_2^G$

In the first case, the proof is the same as **Subcase i**.

In the second case, $\mathcal{D}$ must end in CREATEELEM with $\mathsf{assignW}_{G\downarrow_{EH}}(\sigma_2^G, H, id, v) = (\sigma_4^G, \alpha_2)$

From Lemma 193 (Requirement **(WEH3)**), $\sigma_3^G \approx_L \sigma_4^G$

Case III: $\mathcal{E}$ ends in CREATEELEM

The proof for this case is similar to **Case II**. It uses Lemma 194 (Requirement **(WEH3)**) instead of Lemma 191.

□

Lemma 150 (Weak One-Step - Single Execution Semantics, Weak Secrecy). *If $G, \mathcal{P}, \mathcal{V}, d \vdash_w \sigma_1^G, \kappa_1 \xrightarrow{\alpha_1}_{pc_1} \sigma_3^G, ks_1$, and $G, \mathcal{P}, \mathcal{V}, d \vdash_w \sigma_2^G, \kappa_2 \xrightarrow{\alpha_2}_{pc_2} \sigma_4^G, ks_2$ with $pc_1, pc_2 \sqsubseteq L$, $\alpha_1 = \alpha_2$ when $\alpha_1, \alpha_2 \neq ch(_)$ $\sigma_1^G \approx_L \sigma_2^G$, and $\kappa_1 \approx_L \kappa_2$, then $\sigma_3^G \approx_L \sigma_4^G$ and $ks_1 \approx_L ks_2$*

Proof (sketch): The proof is similar to the one for Lemma 145 (Requirement **(T5)**) except that it uses Lemma 151 instead of Lemma 146. It uses Lemma 182 (Requirement **(EH2)**) and Lemma 172 (Requirement **(EH1)**). □

Lemma 151 (Weak One-Step - Command Semantics, Weak Secrecy). *If $G, \mathcal{V}, d \Vdash_w \sigma_1^G, \sigma_1, c \xrightarrow{\alpha_1}_{pc_1} \sigma_3^G, \sigma_1', c_1, E_1$ and $G, \mathcal{V}, d \Vdash_w \sigma_2^G, \sigma_2, c \xrightarrow{\alpha_2}_{pc_2} \sigma_4^G, \sigma_2', c_2, E_2$, with $pc_1, pc_2 \sqsubseteq L$, $\alpha_1 = \alpha_2$ when $\alpha_1, \alpha_2 \neq ch(_)$, $\sigma_1^G \approx_L \sigma_2^G$, and $\sigma_1 \approx_L \sigma_2$, then $\sigma_3^G \approx_L \sigma_4^G$, $\sigma_1' \approx_L \sigma_2'$, $c_1 \approx_L c_2$, and $E_1 \approx_L E_2$*

Proof.

By induction on the structure of $\mathcal{E} :: G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1, c \xrightarrow{\alpha_1}_{pc_1} \sigma_3^G, \sigma_1', c_1, E_1$.

Denote $\mathcal{D} :: G, \mathcal{V}, d \Vdash \sigma_2^G, \sigma_2, c \xrightarrow{\alpha_2}_{pc_2} \sigma_4^G, \sigma_2', c_2, E_2$

For the most part, this proof is the same as the one for Lemma 146 (Requirement **(T5)**) except that Lemma 153 (Requirement **(WE1)**) is used instead of Lemma 152 (Requirement **(E1)**). It uses Lemma 179 (Requirement **(EH1)**) for the TRIGGER case. We show the most interesting cases below.

Case I: $\mathcal{E}$ ends in IF-TRUE-BR

By assumption, all of the following:

$\mathcal{V} = \mathsf{TT}$, $c = \mathsf{if}\ e\ \mathsf{then}\ c_1'\ \mathsf{else}\ c_2'$, $G, \mathsf{TT}, \sigma_1^G, \sigma_1 \vdash e \Downarrow_{pc_1}^{\mathsf{TT}} (\mathsf{true}, l_1)$, $l_1 \not\sqsubseteq L$, $\alpha_1 = \mathsf{br}(\mathsf{true})$, $c_1 = c_1'$, $\sigma_3^G = \sigma_1^G$, $\sigma_1' = \sigma_1$, and $E_1 = \cdot$

Then, $\mathcal{D}$ must end in IF-TRUE-BR, IF-FALSE-BR, IF-TRUE, or IF-FALSE with $G, \mathsf{TT}, \sigma_2^G, \sigma_2 \vdash e \Downarrow_{pc_2}^{\mathsf{TT}} (b, l_2)$

From Lemma 153 (Requirement **(WV1)**), $(\mathsf{true}, l_1) \approx_L (b, l_2)$

Then from $l_1 \not\sqsubseteq L$, we know that $l_2 \not\sqsubseteq L$

Then, $\mathcal{D}$ must end in IF-TRUE-BR, IF-FALSE-BR

From $\alpha_1 = \alpha_2$ when $\alpha_1, \alpha_2 \neq ch(_)$ and $\alpha_1 = \mathsf{br}(\mathsf{true})$, it must be the case that $\alpha_2 = \mathsf{br}(\mathsf{true})$

Then, $\mathcal{D}$ must end in IF-TRUE-BR with all of the following:

$c_2 = c_1'$, $\sigma_4^G = \sigma_2^G$, $\sigma_2' = \sigma_2$, and $E_2 = \cdot$

From $\sigma_1^G \approx_L \sigma_2^G$ and $\sigma_3^G = \sigma_1^G$, we know that $\sigma_3^G \approx_L \sigma_4^G$

From $\sigma_1 \approx_L \sigma_2$, $\sigma_1' = \sigma_1$, and $\sigma_2' = \sigma_2$, we know that $\sigma_1' \approx_L \sigma_2'$

From $c_1 = c_1'$ and $c_2 = c_1'$, we know that $c_1 \approx_L c_2$

From $E_1 = \cdot$ and $E_2 = \cdot$, we know that $E_1 \approx_L E_2$

The proof for IF-FALSE-BR is similar to **Case I**

The proofs for IF-TRUE and IF-FALSE is similar to **Case I**. Lemma 153 is used to show that the labels on the branch condition are $\sqsubseteq L$.

□

C.4.4 Expression Requirements

Requirement (E1) Equivalent traces produce L-equivalent states

Lemma 152. *If* $\sigma_1^G \approx_L \sigma_2^G$ *and* $\sigma_1 \approx_L \sigma_2$ *with* $pc_1, pc_2 \sqsubseteq L$ *and* $G, \mathcal{V}, \sigma_1^G, \sigma_1 \vdash e \Downarrow_{pc_1}^i v_1$*, then*

Unstructured EH storage: $G, \mathcal{V}, \sigma_2^G, \sigma_2 \vdash e \Downarrow_{pc_2}^i v_2$ *with* $v_1 \simeq_L v_2$

Tree-structured EH storage: $G, \mathcal{V}, \sigma_2^G, \sigma_2 \vdash e \Downarrow_{pc_2}^i v_2$ *with* $v_1 \simeq_L^{\sigma_1, \sigma_2} v_2$

Proof.

By induction on the structure of $\mathcal{E} :: G, \mathcal{V}, \sigma_1^G, \sigma_1 \vdash e \Downarrow_{pc_1}^i v_1$ Denote $\mathcal{D} :: G, \mathcal{V}, \sigma_2^G, \sigma_2 \vdash e \Downarrow_{pc_2}^i v_2$

Unstructured EH storage:

Case U.I: $\mathcal{E}$ ends in VAR

By assumption, all of the following:

$e = x$, $v_1 = \mathsf{toDst}(v_1', pc_1, i)$, and

if $x \in \sigma_1^G$, then $v_1' = \mathsf{var}_{G\downarrow_g}(\sigma_1^G, pc_1, x)$; otherwise, $v_1' = \mathsf{var}_{\mathcal{V}}(\sigma_1, pc_1, x)$

From $e = x$, the last rule applied to $\mathcal{D}$ must have been VAR and all of the following:

$v_2 = \mathsf{toDst}(v_2', pc_2, i)$ and

if $x \in \sigma_2^G$, then $v_2' = \mathsf{var}_{G\downarrow_g}(\sigma_2^G, pc_2, x)$; otherwise, $v_2' = \mathsf{var}_{\mathcal{V}}(\sigma_2, pc_2, x)$

Recall that in our semantics, the set of global variables is static, so $x \in \sigma_1^G$ is true iff $x \in \sigma_2^G$ is true and $v_1' = \mathsf{var}_{\mathcal{V}}(\sigma_1, pc_1, x)$ is true iff $v_2' = \mathsf{var}_{\mathcal{V}}(\sigma_2, pc_2, x)$ is true

Then from Lemma 154.U (Requirement **(V1)**), $v_1' \simeq_L v_2'$

And from Lemma 155.U (Requirement **(V1)**), $v_1 \simeq_L v_2$

Case U.II: $\mathcal{E}$ ends in BOP

By assumption, all of the following:

$e = e_1 \mathbin{\mathsf{bop}} e_2$, $\exists \mathcal{E}_1 :: G, \mathcal{V}, \sigma_1^G, \sigma_1 \vdash e_1 \Downarrow_{pc_1}^i v_{1,1}$, $\exists \mathcal{E}_2 :: G, \mathcal{V}, \sigma_1^G, \sigma_1 \vdash e_2 \Downarrow_{pc_1}^i v_{2,1}$, and $v_1 = v_{1,1} \mathbin{\mathsf{bop}} v_{2,1}$

From $e = e_1 \mathbin{\mathsf{bop}} e_2$, the last rule applied to $\mathcal{D}$ must have been BOP and all of the following:

$\exists \mathcal{D}_1 :: G, \mathcal{V}, \sigma_2^G, \sigma_2 \vdash e_1 \Downarrow_{pc_2}^i v_{1,2}$, $\exists \mathcal{D}_2 :: G, \mathcal{V}, \sigma_1^G, \sigma_1 \vdash e_2 \Downarrow_{pc_2}^i v_{2,2}$, and $v_2 = v_{1,2} \mathbin{\mathsf{bop}} v_{2,2}$

IH on $\mathcal{E}_1, \mathcal{D}_1$ and $\mathcal{E}_2, \mathcal{D}_2$ gives $v_{1,1} \simeq_L v_{1,2}$ and $v_{2,1} \simeq_L v_{2,2}$

Thus, $v_1 \simeq_L v_2$

Case U.III: $\mathcal{E}$ ends in EHAPI

By assumption, all of the following:

$e = \mathsf{ehAPI}(id, e_1, \cdots, e_n)$, $\forall i \in [1, n], \exists \mathcal{E}_i :: G, \mathcal{V}, \sigma_1^G, \sigma_1 \vdash e_i \Downarrow_{pc_1}^{G\downarrow_{EH}} v_{i,1}$,

$v_1' = \mathsf{ehAPI}_{G\downarrow_{EH}}(\sigma_1^G, pc_1, id, v_{1,1}, \cdots, v_{n,1})$, and $v_1 = \mathsf{toDst}(v_1', pc_1, i)$

From $e = \mathsf{ehAPI}(id, e_1, \cdots, e_n)$, the last rule applied to $\mathcal{D}$ must have been EHAPI and all of the following:

$\forall i \in [1, n], \exists \mathcal{D}_i :: G, \mathcal{V}, \sigma_2^G, \sigma_2 \vdash e_i \Downarrow_{pc_2}^{\mathcal{I}\downarrow_{EH}} v_{i,2}$, $v_2' = \mathsf{ehAPI}_{G\downarrow_{EH}}(\sigma_2^G, pc_2, id, v_{1,2}, \cdots, v_{n,2})$, and

$v_2 = \mathsf{toDst}(v_2', pc_2, i)$

IH on $\mathcal{E}_i, \mathcal{D}_i, \forall i \in [1, n]$ gives $\forall i \in [1, n], v_{i,1} \simeq_L v_{i,2}$

Then, from Lemma 162.U (Requirement **(EH1)**), $v_1 \simeq_L v_2$

Tree-structured EH storage:

The proof is similar to the one for the Unstructured storage, except that it uses Lemma 154.T (Requirement **(V1)**), Lemma 155.T (Requirement **(V1)**), and Lemma 162.T (Requirement **(EH1)**).

□

Requirement (WE1) Equivalent traces produce L-equivalent states (Weak Secrecy)

Lemma 153. *If* $\sigma_1 \approx_L \sigma_2$*, with* $pc_1, pc_2 \sqsubseteq L$ *and* $G, \mathcal{V}, \sigma_1^G, \sigma_1 \vdash e \Downarrow_{pc_1}^i v_1$*, then* $G, \mathcal{V}, \sigma_2^G, \sigma_2 \vdash e \Downarrow_{pc_2}^i v_2$ *with* $v_1 \approx_L v_2$

Proof (sketch): The proof is similar to the one for Lemma 152 (Requirement **(E1)**) except that it uses Lemma 156 (Requirement **(WV1)**) instead of Lemma 154 (Requirement **(V1)**) and Lemma 180 (Requirement **(WEH1)**) instead of Lemma 162 (Requirement **(EH1)**). □

C.4.5 Variable Storage Requirements

Requirement (V1) L lookups are equivalent

Lemma 154. *If* $\sigma_1 \approx_L \sigma_2$ *and* $pc_1, pc_2 \sqsubseteq L$*, then for*

Unstructured EH storage:

- $\mathsf{var}_{\mathcal{V}}(\sigma_1, pc_1, x) = v_1$ *and* $\mathsf{var}_{\mathcal{V}}(\sigma_2, pc_2, x) = v_2$ *then* $v_1 \simeq_L v_2$
- $\mathsf{var}_{\mathcal{G}}(\sigma_1, pc_1, x) = v_1$ *and* $\mathsf{var}_{\mathcal{G}}(\sigma_2, pc_1, x) = v_2$ *then* $v_1 \simeq_L v_2$

Tree-structured EH storage:

- $\mathsf{var}_{\mathcal{V}}(\sigma_1, pc_1, x) = v_1$ *and* $\mathsf{var}_{\mathcal{V}}(\sigma_2, pc_2, x) = v_2$ *then* $v_1 \simeq_L^{\sigma_1,\sigma_2} v_2$
- $\mathsf{var}_{\mathcal{G}}(\sigma_1, pc_1, x) = v_1$ *and* $\mathsf{var}_{\mathcal{G}}(\sigma_2, pc_1, x) = v_2$ *then* $v_1 \simeq_L^{\sigma_1,\sigma_2} v_2$

Proof (sketch): Only cases for $\mathcal{V} \neq \mathsf{TT}$ and $\mathcal{G} \neq \mathsf{TS}$ are needed, since the other cases are proven in the weak secrecy version: Lemma 156 (Requirement **(WV1)**). The proof is by straightforward induction on the structure of $\mathcal{E} :: \mathsf{var}_i(\sigma_1, pc_1, x) = v_1$ and $\mathcal{D} :: \mathsf{var}_i(\sigma_2, pc_2, x) = v_2$ □

Lemma 155.

Unstructured EH storage: *If* $v_1 \simeq_L v_2$*, with* $pc_1, pc_2 \sqsubseteq L$ *and* $\mathsf{toDst}(v_1, pc_1, i) = v_1'$*, then* $\mathsf{toDst}(v_2, pc_2, i) = v_2'$ *with* $v_1' \simeq_L v_2'$

Tree-structured EH storage: *If* $v_1 \simeq_L^{\sigma_1,\sigma_2} v_2$*, with* $pc_1, pc_2 \sqsubseteq L$ *and* $\mathsf{toDst}(v_1, pc_1, i) = v_1'$*, then* $\mathsf{toDst}(v_2, pc_2, i) = v_2'$ *with* $v_1' \simeq_L^{\sigma_1,\sigma_2} v_2'$

Proof (sketch): This proof is by straightforward case analysis on the structure of v_1 and v_2. □

Requirement (WV1) L lookups are equivalent (Weak Secrecy)

Lemma 156. *If $\sigma_1 \approx_L \sigma_2$ and $pc_1, pc_2 \sqsubseteq L$, then for*

Unstructured EH storage:

- $\mathsf{var}_{\mathcal{V}}(\sigma_1, pc_1, x) = v_1$ *and* $\mathsf{var}_{\mathcal{V}}(\sigma_2, pc_2, x) = v_2$ *then* $v_1 \approx_L v_2$
- $\mathsf{var}_{\mathcal{G}}(\sigma_1, pc_1, x) = v_1$ *and* $\mathsf{var}_{\mathcal{G}}(\sigma_2, pc_1, x) = v_2$ *then* $v_1 \approx_L v_2$

Tree-structured EH storage:

- $\mathsf{var}_{\mathcal{V}}(\sigma_1, pc_1, x) = v_1$ *and* $\mathsf{var}_{\mathcal{V}}(\sigma_2, pc_2, x) = v_2$ *then* $v_1 \approx_L^{\sigma_1,\sigma_2} v_2$
- $\mathsf{var}_{\mathcal{G}}(\sigma_1, pc_1, x) = v_1$ *and* $\mathsf{var}_{\mathcal{G}}(\sigma_2, pc_1, x) = v_2$ *then* $v_1 \approx_L^{\sigma_1,\sigma_2} v_2$

Proof (sketch): Only the cases for TT and TS are needed since the other cases follow from Lemma 154 (Requirement **(V1)**). The proof is by straightforward induction on the structure of $\mathcal{E} :: \mathsf{var}_i(\sigma_1, pc_1, x) = v_1$ and $\mathcal{D} :: \mathsf{var}_i(\sigma_2, pc_2, x) = v_2$. □

Requirement (V2) H assignments are unobservable

Lemma 157. $\mathsf{assign}_{\mathcal{G}}(\sigma, H, x, v) \approx_L \sigma$

Proof (sketch): Only cases for $\mathcal{G} \neq$ TS are needed since the other cases are proven in the weak secrecy version: Lemma 158 (Requirement **(WV2)**) The proof is by case analysis on $\mathcal{G}$ □

Requirement (WV2) H assignments are unobservable (Weak Secrecy)

Lemma 158. *If* $\mathsf{assignW}_{\mathcal{G}}(\sigma, H, x, v) = (\sigma', \bullet)$, *then* $\sigma \approx_L \sigma'$

Proof (sketch): Only cases for $\mathcal{G} =$ TS are needed since the other cases follow from Lemma 157 (Requirement **(V2)**). The proof only requires two cases: one where $x \in \sigma$ and one where $x \notin \sigma$. □

Lemma 159. *If $\sigma_1 \approx_L \sigma_2$ and* $\mathsf{assignW}_{\mathcal{G}}(\sigma_1, H, x, v) = (\sigma_1', \mathsf{gw}(x))$, *and* $\mathsf{assignW}_{\mathcal{G}}(\sigma_2, H, x, v) = (\sigma_2', \mathsf{gw}(x))$ *then* $\sigma_1' \approx_L \sigma_2'$

Proof.

Denote $\mathcal{D} :: \mathsf{assignW}_{\mathcal{G}}(\sigma_1, H, x, (v_1, l_1)) = (\sigma_1', \alpha_1)$ and $\mathcal{E} :: \mathsf{assignW}_{\mathcal{G}}(\sigma_2, H, x, (v_2, l_2)) = (\sigma_2', \alpha_2)$

By assumption, $\alpha_1 = \alpha_2 = \mathsf{gw}(x)$ and $\sigma_1 \approx_L \sigma_2$

From $\alpha_1 = \alpha_2 = \mathsf{gw}(x)$ and since only the TS semantics can produce $\mathsf{gw}(_)$, $\mathcal{D}$ and $\mathcal{E}$ end in ASSIGN-GW, meaning that $\sigma_1' = \sigma_1[x \mapsto (v_1, l_1 \sqcup H)]$ and $\sigma_2' = \sigma_2[x \mapsto (v_2, l_2 \sqcup H)]$

From our security lattice, $l_1 \sqcup H = H$ and $l_2 \sqcup H = H$

Thus, $\sigma_1' \approx_L \sigma_2'$

□

Requirement (V3) L assignments are equivalent

Lemma 160. *If $\sigma_1 \approx_L \sigma_2$ and $pc_1, pc_2 \in \{L, \cdot\}$, with*

Unstructured EH storage: $v_1 \approx_L v_2$, *then* $\mathsf{assign}_i(\sigma_1, pc_1, x, v_1) \approx_L \mathsf{assign}_i(\sigma_2, pc_2, x, v_2)$

Tree-structured EH storage: $v_1 \approx_L^{\sigma_1,\sigma_2} v_2$, *then* $\mathsf{assign}_i(\sigma_1, pc_1, x, v_1) \approx_L \mathsf{assign}_i(\sigma_2, pc_2, x, v_2)$

Proof (sketch): The proof is straightforward. We consider each mechanism i separately and have one case for each pc. □

Requirement (WV3) L assignments are equivalent (Weak Secrecy)

Lemma 161. *If $\sigma_1 \approx_L \sigma_2$ and $pc_1, pc_2 \in \{L, \cdot\}$, with $v_1 \approx_L v_2$, then for $(\sigma_1', \alpha_1) = \mathsf{assignW}_i(\sigma_1, pc_1, x, v_1)$ and $(\sigma_2', \alpha_2) = \mathsf{assignW}_i(\sigma_2, pc_2, x, v_2)$, $\sigma_1' \approx_L \sigma_2'$ and $\alpha_1 = \alpha_2$*

Proof (sketch): Only the cases for $i \in \{\mathsf{TT}, \mathsf{TS}\}$ are needed, since the other cases always produce $\alpha_1 = \alpha_2 = \bullet$, so their proofs follow from Lemma 160 (Requirement **(V3)**). The proof is by case analysis on $\mathcal{D} :: (\sigma_1', \alpha_1) = \mathsf{assignW}_i(\sigma_1, pc_1, x, (v_1, l_1))$ □

C.4.6 Event Handler Storage Requirements

Requirement (EH1) L lookups are equivalent

Lemma 162. *If $\sigma_1 \approx_L \sigma_2$ and $pc_1, pc_2 \sqsubseteq L$ and*

Unstructured EH storage: $\forall i \in [1, n], v_{i,1} \simeq_L v_{i,2}$ *with* $v_1 = \mathsf{ehAPIe}(\mathcal{G}, \sigma_1, pc_1, id, v_{1,1}, \cdots, v_{n,1})$ *and* $v_2 = \mathsf{ehAPIe}(\mathcal{G}, \sigma_2, pc_2, id, v_{1,2}, \cdots, v_{n,2})$, *then* $v_1 \simeq_L v_2$

Tree-structured EH storage: $\forall i \in [1, n], v_{i,1} \simeq_L^{\sigma_1,\sigma_2} v_{i,2}$ *with* $v_1 = \mathsf{ehAPIe}(\mathcal{G}, \sigma_1, pc_1, a_1, v_{1,1}, \cdots, v_{n,1})$ *and* $v_2 = \mathsf{ehAPIe}(\mathcal{G}, \sigma_2, pc_2, a_2, v_{1,2}, \cdots, v_{n,2})$, *then* $v_1 \simeq_L^{\sigma_1,\sigma_2} v_2$

Proof (sketch): By induction on the structure of $\mathcal{E} :: v_1 = \mathsf{ehAPIe}(...)$ and $\mathcal{D} :: v_2 = \mathsf{ehAPIe}(...)$. The case where $\mathcal{E}$ or $\mathcal{D}$ end in EHAPI uses Lemma 163.U (for the unstructured DOM) or Lemma 163.T (for the tree-structured DOM). □

Lemma 163. *If $\sigma_1 \approx_L \sigma_2$ and $pc_{l,1}, pc_{l,2} \sqsubseteq L$, then*

Unstructured EH storage: *if* $\forall i \in [1, n] v_{i,1} \simeq_L v_{i,2}$, *then* $\mathsf{ehAPIe}_{\mathcal{G}}(\sigma_1, pc_1, id, v_{1,1}, \cdots, v_{n,1}) \simeq_L$

$\mathsf{ehAPIe}_{\mathcal{G}}(\sigma_2, pc_2, id, v_{1,2}, \cdots, v_{n,2})$

Tree-structured EH storage: *if* $\forall i \in [1, n] v_{i,1} \simeq_L^{\sigma_1,\sigma_2} v_{i,2}$, *then* $\mathsf{ehAPIe}_{\mathcal{G}}(\sigma_1, pc_1, id, v_{1,1}, \cdots, v_{n,1}) \simeq_L^{\sigma_1,\sigma_2}$

$\mathsf{ehAPIe}_{\mathcal{G}}(\sigma_2, pc_2, id, v_{1,2}, \cdots, v_{n,2})$

Proof (sketch): There are additional lemmas for each event handler API.

Unstructured EH storage: The conclusion follows from Lemma 166.U when ehAPIe is getVal.

Tree-structured EH storage: The conclusion follows from Lemma 166.T when ehAPIe is getVal, Lemma 167 when ehAPIe is getChildren, Lemma 168 when ehAPIe is moveRoot, Lemma 169 when ehAPIe is moveUp, Lemma 170 when ehAPIe is moveDown, and Lemma 171 when ehAPIe is moveRight. □

Lemma 164. *If* $\sigma_1 \approx_L \sigma_2$ *and* $pc_1, pc_2 \sqsubseteq L$ *then*

Unstructured EH storage: $\mathsf{lookup}_\mathcal{G}(\sigma_1, pc_1, id) \approx_L \mathsf{lookup}_\mathcal{G}(\sigma_2, pc_2, id)$

Tree-structured EH storage: $\mathsf{lookup}_\mathcal{G}(\sigma_1, pc_1, id) \approx_L^{\sigma_1,\sigma_2} \mathsf{lookup}_\mathcal{G}(\sigma_2, pc_2, id)$

Proof (sketch): The proof is by induction on the structure of $\mathcal{E} :: \mathsf{lookup}_\mathcal{G}(\sigma_1, pc_1, id)$ and $\mathcal{D} :: \mathsf{lookup}_\mathcal{G}(\sigma_2, pc_2, id)$. The cases for the tree-structured DOM use Lemma 165. □

Lemma 165. *If* $\sigma_1 \approx_L \sigma_2$*,* $pc_1, pc_2 \sqsubseteq L$ *and* $A_1 \approx_L^{\sigma_1,\sigma_2} A_2$*, then* $\mathsf{lookupA}_\mathcal{G}(\sigma_1, pc_1, id, A_1) \approx_L^{\sigma_1,\sigma_2} \mathsf{lookupA}_\mathcal{G}(\sigma_2, pc_2, id, A_2)$

Proof (sketch): The proof is by induction on the structure of $\mathcal{E} :: \mathsf{lookupA}_\mathcal{G}(\sigma_1, pc_1, id, A_1)$ and $\mathcal{D} :: \mathsf{lookupA}_\mathcal{G}(\sigma_2, pc_2, id, A_2)$. and largely relies on the assumption that $A_1 \approx_L^{\sigma_1,\sigma_2} A_2$. □

Lemma 166. *If* $\sigma_1 \approx_L \sigma_2$ *and* $pc_1, pc_2 \sqsubseteq L$*, then*

Unstructured EH storage: $\mathsf{getVal}_\mathcal{G}(\sigma_1, pc_1, id) \simeq_L \mathsf{getVal}_\mathcal{G}(\sigma_2, pc_2, id)$

Tree-structured EH storage: *if* $a_1 \approx_L^{\sigma_1,\sigma_2} a_2$ $\mathsf{getVal}_\mathcal{G}(\sigma_1, pc_1, a_1) \simeq_L^{\sigma_1,\sigma_2} \mathsf{getVal}_\mathcal{G}(\sigma_2, pc_2, a_2)$

Proof (sketch): Only the cases for $\mathcal{G} \neq$ TS are needed. The other cases are proven in the weak secrecy version: Lemma 181 (Requirement **(WE1)**). In the proof, we examine each case of $\mathcal{G}$. The cases for the unstructured DOM use Lemma 164.U, while the ones for the tree-structured DOM rely on the assumption that $a_1 \approx_L^{\sigma_1,\sigma_2} a_2$. □

Lemma 167. *If* $\sigma_1 \approx_L \sigma_2$*,* $pc_1, pc_2 \sqsubseteq L$*, and* $a_1 \approx_L^{\sigma_1,\sigma_2} a_2$*, then* $\mathsf{getChildren}_\mathcal{G}(\sigma_1, pc_1, a_1) \simeq_L^{\sigma_1,\sigma_2} \mathsf{getChildren}_\mathcal{G}(\sigma_2, pc_2, a_2)$

Proof (sketch): Note that getChildren is defined only for the tree-structured EH storage. We split the proof by $\mathcal{G}$ and then proceed by induction over the structure of $\mathcal{D} :: \mathsf{getChildren}_\mathcal{G}(\sigma_1, pc_1, a_1) = v_1$ and $\mathcal{E} :: \mathsf{getChildren}_\mathcal{G}(\sigma_2, pc_2, a_2) = v_2$. □

Lemma 168. *If* $\sigma_1 \approx_L \sigma_2$ *and* $pc_1, pc_2 \sqsubseteq L$*, then* $\mathsf{moveRoot}_\mathcal{G}(\sigma_1, pc_1) \simeq_L^{\sigma_1,\sigma_2} \mathsf{moveRoot}_\mathcal{G}(\sigma_2, pc_2)$

Proof (sketch): Note that moveRoot is defined only for the tree-structured EH storage. The proof is very straightforward, with just one case per $\mathcal{G}$. □

Lemma 169. *If* $\sigma_1 \approx_L \sigma_2$*,* $pc_1, pc_2 \sqsubseteq L$*, and* $a_1 \approx_L^{\sigma_1,\sigma_2} a_2$*, then* $\mathsf{moveUp}_\mathcal{G}(\sigma_1, pc_1, a_1) \simeq_L^{\sigma_1,\sigma_2} \mathsf{moveUp}_\mathcal{G}(\sigma_2, pc_2, a_2)$

Proof (sketch): Note that moveUp is defined only for the tree-structured EH storage. We split the proof by $\mathcal{G}$ and then proceed by induction over the structure of $\mathcal{D} :: \mathsf{moveUp}_{\mathcal{G}}(\sigma_1, pc_1, a_1) = a_1'$ and $\mathcal{E} :: \mathsf{moveUp}_{\mathcal{G}}(\sigma_2, pc_2, a_2) = a_2'$. □

Lemma 170. *If $\sigma_1 \approx_L \sigma_2$, $pc_1, pc_2 \sqsubseteq L$, and $a_1 \approx_L^{\sigma_1,\sigma_2} a_2$, then $\mathsf{moveDown}_{\mathcal{G}}(\sigma_1, pc_1, a_1) \simeq_L^{\sigma_1,\sigma_2} \mathsf{moveDown}_{\mathcal{G}}(\sigma_2, pc_2, a_2)$*

Proof (sketch): Note that $\mathsf{moveDown}$ is defined only for the tree-structured EH storage. We split the proof by $\mathcal{G}$ and then proceed by induction over the structure of $\mathcal{D} :: \mathsf{moveDown}_{\mathcal{G}}(\sigma_1, pc_1, a_1) = a_1'$ and $\mathcal{E} :: \mathsf{moveDown}_{\mathcal{G}}(\sigma_2, pc_2, a_2) = a_2'$. □

Lemma 171. *If $\sigma_1 \approx_L \sigma_2$, $pc_1, pc_2 \sqsubseteq L$, and $a_1 \approx_L^{\sigma_1,\sigma_2} a_2$, then $\mathsf{moveRight}_{\mathcal{G}}(\sigma_1, pc_1, a_1) \simeq_L^{\sigma_1,\sigma_2} \mathsf{moveRight}_{\mathcal{G}}(\sigma_2, pc_2, a_2)$*

Proof (sketch): Note that $\mathsf{moveRight}$ is defined only for the tree-structured EH storage. We split the proof by $\mathcal{G}$ and then proceed by induction over the structure of $\mathcal{D} :: \mathsf{moveRight}_{\mathcal{G}}(\sigma_1, pc_1, a_1) = a_1'$ and $\mathcal{E} :: \mathsf{moveRight}_{\mathcal{G}}(\sigma_2, pc_2, a_2) = a_2'$ □

Lemma 172. *If $\sigma_1 \approx_L \sigma_2$ and $ks_1 \approx_L ks_2$ with $E_1 \approx_L E_2$ and $pc_1, pc_2 \sqsubseteq L$, and one of the following:*

1. *When $G, \mathcal{P}, \sigma_1 \vdash ks_1; \mathsf{lookupEHAt}(id.\mathsf{Ev}(v)) \rightsquigarrow_{pc_1} ks_1'$, and $G, \mathcal{P}, \sigma_2 \vdash ks_2; \mathsf{lookupEHAt}(id.\mathsf{Ev}(v)) \rightsquigarrow_{pc_2} ks_2'$,*
2. *When $G, \mathcal{P}, \sigma_1 \vdash ks_1; \mathsf{lookupEHAll}(id.\mathsf{Ev}(v)) \rightsquigarrow_{pc_1} ks_1'$, and $G, \mathcal{P}, \sigma_2 \vdash ks_2; \mathsf{lookupEHAll}(id.\mathsf{Ev}(v)) \rightsquigarrow_{pc_2} ks_2'$,*
3. *When $G, \mathcal{P}, \sigma_1 \vdash ks_1; \mathsf{lookupEHAt}(id.\mathsf{Ev}(v)) \rightsquigarrow_L ks_1'$ and $G, \mathcal{P}, \sigma_2 \vdash ks_2; \mathsf{lookupEHAll}(id.\mathsf{Ev}(v)) \rightsquigarrow_{\cdot} ks_2'$, or*
4. *When $G, \mathcal{P}, \sigma_1 \vdash ks_1; \mathsf{lookupEHs}(E_1) \rightsquigarrow_{pc_1} ks_1'$, and $G, \mathcal{P}, \sigma_2 \vdash ks_2; \mathsf{lookupEHs}(E_2) \rightsquigarrow_{pc_2} ks_2'$,*

then $ks_1' \approx_L ks_2'$

Proof.

By induction on the structure of $\mathcal{D} :: G, \mathcal{P}, \sigma_1 \vdash ks_1; \mathsf{lookupEHAPI}(...) \rightsquigarrow_{pc_1} ks_1'$ and

$\mathcal{E} :: G, \mathcal{P}, \sigma_2 \vdash ks_2; \mathsf{lookupEHAPI}(...) \rightsquigarrow_{pc_2} ks_2'$

We refer to the following assumptions throughout:

(1) $\sigma_1 \approx_L \sigma_2$; (2) $ks_1 \approx_L ks_2$; (3) $pc_1, pc_2 \sqsubseteq L$; and (4) $E_1 \approx_L E_2$

Case I: $\mathcal{D}$ and $\mathcal{E}$ end in LOOKUPEHAPI

$\mathcal{C}_1 \approx_L \mathcal{C}_2$ follows from Lemma 175, Lemma 176, or Lemma 177, depending on the APIs.

The conclusion follows from (2) and Lemma 173

Case II: $\mathcal{D}$ ends in LOOKUPEHs-s

If $E_2 = \cdot$, then $\mathcal{E}$ ends in the same rule. Then, the proof follows from (2). Otherwise, $\mathcal{E}$ ends in LOOKUPEHS-R and the proof follows from Lemma 182 (Requirement **(EH2)**).

Case III: $\mathcal{E}$ ends in LOOKUPEHS-S

The proof for this case is similar to **Case II**.

Case IV: $\mathcal{D}$ ends in LOOKUPEHS-R

If $l_1, l_2 \sqsubseteq L$, then $\mathcal{E}$ ends in the same rule and the proof follows from Lemmas 178 and 178, and the IH.

If $l_1 \not\sqsubseteq L$ or $l_2 \not\sqsubseteq L$, then the proof follows from Lemmas 185 and 186 (Requirement **(EH2)**), and the IH.

Case V: $\mathcal{E}$ ends in LOOKUPEHS-R

The proof for this case is similar to the one for **Case IV**.

□

Lemma 173. *If* $\mathcal{C}_1 \approx_L \mathcal{C}_2$ *and* $ks_1 = \mathsf{createK}(\mathcal{P}, id.\mathsf{Ev}(v), \mathcal{C}_1)$ *and* $ks_2 = \mathsf{createK}(\mathcal{P}, id.\mathsf{Ev}(v), \mathcal{C}_2)$*, then* $ks_1 \approx_L ks_2$

Proof (sketch): The proof is by straightforward induction on the structure of $\mathcal{D} :: \mathsf{createK}(\mathcal{P}, id.\mathsf{Ev}(v), \mathcal{C}_1)$ and $\mathcal{E} :: \mathsf{createK}(\mathcal{P}, id.\mathsf{Ev}(v), \mathcal{C}_2)$. It uses Lemma 174 for L event handlers and Lemma 187 (Requirement **(EH2)**) for any H event handlers. □

Lemma 174. *If* $pc_1, pc_2 \sqsubseteq L$ *and* $ks_1 = \mathsf{crtK}_{\mathcal{V}}(eh, v, pc_1)$ *and* $ks_2 = \mathsf{crtK}_{\mathcal{V}}(eh, v, pc_2)$*, then* $ks_1 \approx_L ks_2$

Proof (sketch): This proof is straightforward. It uses the definition of $\approx_L$ for κ. □

Lemma 175. $\sigma_1 \approx_L \sigma_2$ *with* $pc_1, pc_2 \sqsubseteq L$*, then* $\mathsf{lookupEHAll}_G(\sigma_1, pc_1, id.\mathsf{Ev}(v)) \approx_L \mathsf{lookupEHAll}_G(\sigma_2, pc_2, id.\mathsf{Ev}(v))$

Proof.

By induction on the structure of $\mathcal{E} :: \mathcal{C}_1 = \mathsf{lookupEHAll}_G(\sigma_1, id.\mathsf{Ev}(v), pc_1)$ and

$\mathcal{D} :: \mathcal{C}_2 = \mathsf{lookupEHAll}_G(\sigma_2, id.\mathsf{Ev}(v), pc_2)$

Want to show $\mathcal{C}_1 \approx_L \mathcal{C}_2$

Case I: $\mathcal{D}$ ends in LOOKUPEHALL-S

By assumption, all of the following:

$\mathsf{lookup}_{G\downarrow_{EH}}(\sigma_1, pc_1, id) = \phi_1$, $\mathsf{valOf}(\phi_1) = \mathsf{NULL}$, and $\mathcal{C}_1 = \cdot$

Subcase i: $pc_2 = L$

From Lemma 164.U (for the unstructured EH storage), either

(i.1) $\mathsf{lookup}_{G\downarrow_{EH}}(\sigma_2, pc_2, id) = \phi_2 \approx_L \phi_1$ with $\mathsf{labOf}(\phi_1, pc_1) = \mathsf{labOf}(\phi_2, pc_2) = L$; or

(i.2) $\mathsf{lookup}_{G\downarrow_{EH}}(\sigma_2, pc_2, id) = \phi_2 \approx_L \phi_1$ with $\mathsf{labOf}(\phi_1, pc_1) = \mathsf{labOf}(\phi_2, pc_2) = H$

From Lemma 164.T (for the tree-structured EH storage),

(i.3) $\mathsf{lookup}_{G\downarrow_{EH}}(\sigma_2, pc_2, id) \approx_L^{\sigma_2, \sigma_1} \phi_1$

Subsubcase a: (i.1) or (i.3) (above) are true

From (i.1) (for the unstructured EH storage) and (i.3) (for the tree-structured EH storage), $\mathcal{E}$ ends in LOOKUPEHALL-S with $\mathcal{C}_2 = \cdot$

Then from $\mathcal{C}_1 = \cdot$, we know that $\mathcal{C}_1 \approx_L \mathcal{C}_2$

Subsubcase b: (i.2) (above) is true

From (i.2), $\mathcal{C}_1 \approx_L \cdot$ and either

$\mathcal{E}$ ends in LOOKUPEHALL; or $\mathcal{E}$ ends in LOOKUPEHALL-S

In the first case, then from (i.2), $\mathcal{C}_2 \approx_L \cdot$

Otherwise, $\mathcal{C}_2 = \cdot$

Thus, from $\mathcal{C}_1 \approx_L \cdot$, we know that $\mathcal{C}_1 \approx_L \mathcal{C}_2$

Subcase ii: $pc_2 = \cdot$

By assumption, $\mathcal{E}$ ends in LOOKUPEHALL-NC-MERGE and all of the following:

$\exists \mathcal{E}' :: \mathsf{lookupEHAll}_G(\sigma_2, H, id.\mathsf{Ev}(v)) = \mathcal{C}_H$, $\exists \mathcal{E}'' :: \mathsf{lookupEHAll}_G(\sigma_2, L, id.\mathsf{Ev}(v)) = \mathcal{C}_L$, and $\mathcal{C}_2 = \mathsf{mergeC}(\mathcal{C}_H, \mathcal{C}_L)$

From Lemma 183 (Requirement **(EH2)**), $\mathcal{C}_H \approx_L \cdot$

IH on $\mathcal{D}$ and $\mathcal{E}''$ gives $\mathcal{C}_L \approx_L \mathcal{C}_1$

Then from the definition of mergeC, $\mathcal{C}_1 \approx_L \mathcal{C}_2$

Case II: $\mathcal{E}$ ends in LOOKUPEHALL-S

The proof for this case is similar to **Case I**.

Case III: $\mathcal{E}$ ends in LOOKUPEHALL

By assumption, all of the following:

$\mathsf{lookup}_{G\downarrow_{EH}}(\sigma_1, pc_1, id) = \phi_1$, $\mathsf{valOf}(\phi_1) \neq \mathsf{NULL}$, $\mathsf{labOf}(\phi_1, pc_1) = l_1$, $\mathcal{C}_1 = (\phi_1.M(\mathsf{Ev}) \downarrow_{pc_1}) \sqcup pc_1 \sqcup l_1$, and $pc_1 = L$

From Lemma 164.U (for the unstructured EH storage), we know that either

(III.1) $\mathsf{lookup}_{G\downarrow_{EH}}(\sigma_2, pc_2, id) = \phi_2 \approx_L \phi_1$ with $\mathsf{labOf}(\phi_1, pc_1) = \mathsf{labOf}(\phi_1, pc_2) = L$; or

(III.2) $\mathsf{lookup}_{G\downarrow_{EH}}(\sigma_2, pc_2, id) = \phi_2 \approx_L \phi_1$ with $\mathsf{labOf}(\phi_1, pc_1) = \mathsf{labOf}(\phi_1, pc_2) = H$

Then from Lemma 164.T (for the tree-structured EH storage), we know that

(III.3) $\mathsf{lookup}_{G\downarrow_{EH}}(\sigma_2, pc_2, id) \approx_L^{\sigma_2,\sigma_1} \phi_1$

Subcase i: $pc_2 = L$ and (III.1) or (III.3) is true

By assumption, $\mathcal{E}$ ends in LOOKUPEHALL and all of the following:

$\mathsf{labOf}(\phi_2, pc_2) = l_2$, $\mathcal{C}_2 = (\phi_2.M(\mathsf{Ev}) \downarrow_{pc_2}) \sqcup pc_2 \sqcup l_2$, and $pc_2 = L$

Then, we know that $l_1 = l_2 = L$

From the definition of $\downarrow_L$ for M (which projects L- and

$\cdot$-labeled event handlers to be labeled with L), we also know that $\mathcal{C}_1 = \phi_1.M(\mathsf{Ev}) \downarrow_L$ and

$\mathcal{C}_2 = \phi_2.M(\mathsf{Ev}) \downarrow_L$

Then from the definition of $\downarrow_L$ for M, $\mathcal{C}_1 \approx_L \mathcal{C}_2$

Subcase ii: $pc_2 = \cdot$

By assumption, $\mathcal{E}$ ends in LOOKUPEHALL-NC-MERGE and the rest of the proof is similar to **Subcase I.ii**.

Subcase iii: (III.1) is true

The proof for this case is similar to **Subcase I.i.b**.

Case IV: $\mathcal{D}$ ends in LOOKUPEHALL

The proof for this case is similar to the one for **Case III**.

Case V: $\mathcal{E}$ ends in LOOKUPEHALL-NC-MERGE

By assumption, all of the following:

$\mathcal{E}' :: \mathcal{C}_H = \mathsf{lookupEHAll}_{\mathcal{G}}(\sigma_1, H, id.\mathsf{Ev}(v))$,

$\mathcal{E}'' :: \mathcal{C}_L = \mathsf{lookupEHAll}_{\mathcal{G}}(\sigma_1, L, id.\mathsf{Ev}(v))$, and

$\mathcal{C}_1 = \mathsf{mergeC}(\mathcal{C}_H, \mathcal{C}_L)$

From Lemma 183 (Requirement **(EH2)**), $\mathcal{C}_H \approx_L \cdot$

IH on $\mathcal{D}$ and $\mathcal{E}''$ gives $\mathcal{C}_L \approx_L \mathcal{C}_1$

Then, from the definition of mergeC, $\mathcal{C}_1 \approx_L \mathcal{C}_2$

Case VI: $\mathcal{D}$ ends in LOOKUPEHALL-NC-MERGE

The proof is similar to **Case V**

□

Lemma 176. *$\sigma_1 \approx_L \sigma_2$ and $pc_1, pc_2 \sqsubseteq L$, then* $\mathsf{lookupEHAt}_{\mathcal{G}}(\sigma_1, pc_1, id.\mathsf{Ev}(v_1)) \approx_L \mathsf{lookupEHAt}_{\mathcal{G}}(\sigma_2, pc_2, id.\mathsf{Ev}(v_2))$

Lemma 177. *$\sigma_1 \approx_L \sigma_2$, then* $\mathsf{lookupEHAt}_{\mathcal{G}}(\sigma_1, L, id.\mathsf{Ev}(v)) \approx_L \mathsf{lookupEHAll}_{\mathcal{G}}(\sigma_2, \cdot, id.\mathsf{Ev}(v))$

Proof (sketch): The proof for this case is similar to Lemma 175 and uses the fact that $(M(\mathsf{Ev})@L) \sqcup L$ returns the same thing as $M(\mathsf{Ev}) \downarrow_L$ and Lemma 183 (Requirement **(EH2)**) is used to show that $\mathsf{lookupEHAll}_{\mathcal{G}}(\sigma_2, \cdot, id.\mathsf{Ev}(v)) \approx_L \mathsf{lookupEHAll}_{\mathcal{G}}(\sigma_2, L, id.\mathsf{Ev}(v))$. □

Lemma 178. *$\sigma_1 \approx_L \sigma_2$ and $pc_1, pc_2 \sqsubseteq L$, then* $\mathsf{lookupEHs}_{\mathcal{G}}(\sigma_1, pc_1, id.\mathsf{Ev}(v)) \approx_L \mathsf{lookupEHs}_{\mathcal{G}}(\sigma_2, pc_2, id.\mathsf{Ev}(v))$

Proof (sketch): Follows from Lemma 175 and Lemma 176. □

Lemma 179. *If $\sigma_1 \approx_L \sigma$, $pc_1, pc_2 \sqsubseteq L$, and $v_1 \approx_L v_2$ then* $\mathsf{triggerEH}_{\mathcal{G}}(\sigma_1, pc_1, id, \mathsf{Ev}, v_1) \approx_L \mathsf{triggerEH}_{\mathcal{G}}(\sigma_2, pc_2, id, \mathsf{Ev}, v_2)$

Proof (sketch): The proof is by induction on the structure of $\mathcal{E}$ :: $\mathsf{triggerEH}_{\mathcal{G}}(\sigma_1, pc_1, id, \mathsf{Ev}, v_1)$ and $\mathcal{D}$ :: $\mathsf{triggerEH}_{\mathcal{G}}(\sigma_2, pc_2, id, \mathsf{Ev}, v_2)$ and uses Lemma 164.U (for lookups in the unstructured DOM), Lemma 165 (for lookups in the tree-structured DOM), and Lemma 188 (Req **(EH2)**). □

Requirement (WEH1) L lookups are equivalent

Lemma 180. *If $\sigma_1 \approx_L \sigma_2$ and $pc_1, pc_2 \sqsubseteq L$ and $\forall i \in [1, n], v_{i,1} \approx_L v_{i,2}$ with $v_{i,1} \downarrow_{pc_1} = v_{i,1}$, $v_{i,2} \downarrow_{pc_2} = v_{i,2}$, and $t_1 = \mathsf{ehAPI}_{\mathcal{I}}(\sigma_1, pc_1, id, v_{1,1}, \cdots, v_{n,1})$ and $t_2 = \mathsf{ehAPI}_{\mathcal{I}}(\sigma_2, pc_2, id, v_{1,2}, \cdots, v_{n,2})$, then $t_1 \approx_L t_2$*

Proof (sketch): The proof is similar to the one for Lemma 162 (Requirement **(EH1)**). It uses Lemma 164 (Requirement **(EH1)**) and Lemma 181 instead of Lemma 166. □

Lemma 181. *If $pc_1, pc_2 \sqsubseteq L$ and $\phi_1 \approx_L \phi_2$, with $\mathsf{getValG}_{\mathcal{G}}(pc_1, \phi_1) = v_1$ and $\mathsf{getValG}_{\mathcal{G}}(pc_1, \phi_2) = v_2$, then $v_1 \approx_L v_2$*

Proof (sketch): Only the cases for $\mathcal{G} = \mathsf{TS}$ are needed. The other cases follow from Lemma 166 (Requirement **(EH1)**). □

Requirement (EH2) H EH lookups are unobservable

Lemma 182. *If any of the following:*

1. $G, \mathcal{P}, \sigma \vdash ks; \mathsf{lookupEHAll}(id.\mathsf{Ev}(v)) \rightsquigarrow_H ks'$ *or*
2. $G, \mathcal{P}, \sigma \vdash ks; \mathsf{lookupEHAt}(id.\mathsf{Ev}(v)) \rightsquigarrow_H ks'$ *or*
3. $G, \mathcal{P}, \sigma \vdash ks; \mathsf{lookupEHs}(E) \rightsquigarrow_H ks'$ *or*
4. $G, \mathcal{P}, \sigma \vdash ks; \mathsf{lookupEHs}(E) \rightsquigarrow_{pc} ks'$

then $ks \approx_L ks'$

Proof (sketch):

By induction on the structure of $\mathcal{E}$ :: $G, \mathcal{P}, \sigma \vdash ks; \mathsf{lookupEHAPI}(...) \rightsquigarrow_{pc} ks'$

Case I: $\mathcal{E}$ ends in LOOKUPEHAPI with $pc = H$

The proof follows from Lemma 183 (Requirement **(EH2)**), Lemma 184 and Lemma 186.

Case II: $\mathcal{E}$ ends in LOOKUPEHs-R with $pc = H$

Follows from Lemma 185 (Requirement **(EH2)**), Lemma 186, and the IH.

Case III: $\mathcal{E}$ ends in LOOKUPEHs-R with $E \approx_L \cdot$

The assumption that $E \approx_L \cdot$ allows us to apply Lemma 185 (Requirement **(EH2)**). Then, the proof is similar to **Case II**.

Case IV: $\mathcal{E}$ ends in LOOKUPEHs-s

Follows from assumption that $ks' = ks$.

□

Lemma 183. $\mathsf{lookupEHAll}_G(\sigma, id.\mathsf{Ev}(v), H) \approx_L \cdot$

Proof (sketch): The case for LOOKUPEHALL follows from our security lattice (everything is joined with the *pc*, which is H, here). The case for LOOKUPEHALL-S is straightforward. The $pc = \cdot$ in LOOKUPEHALL-NC-MERGE, so this case holds vacuously. □

Lemma 184. $\mathsf{lookupEHAt}_G(\sigma, id.\mathsf{Ev}(v), H) \approx_L \cdot$

Proof (sketch): The case for LOOKUPEHAT follows from our security lattice (everything is joined with the *pc*, which is H, here). The case for LOOKUPEHAT-S is straightforward. The $pc = \cdot$ in LOOKUPEHAT-NC-MERGE, so this case holds vacuously. □

Lemma 185. $\mathsf{lookupEHs}_G(\sigma, id.\mathsf{Ev}(v), H) \approx_L \cdot$

Proof (sketch): The proof follows from Lemma 183 (Requirement **(EH2)**) and Lemma 184 □

Lemma 186. *If $\mathcal{C} \approx_L \cdot$ and $ks = \mathsf{createK}(\mathcal{P}, id.\mathsf{Ev}(v), \mathcal{C})$ then $ks \approx_L \cdot$*

Proof (sketch): This proof is by straightforward induction on the structure of $\mathcal{D} :: \mathsf{createK}(\mathcal{P}, id.\mathsf{Ev}(v), \mathcal{C})$. It uses Lemma 187. □

Lemma 187. *If $ks = \mathsf{crtK}_{\mathcal{V}}(eh, v, H)$ then $ks \approx_L \cdot$*

Proof (sketch): This proof is straightforward and follows directly from the assumption that the $pc = H$. □

Lemma 188. $\mathsf{triggerEH}_{\mathcal{G}}(\sigma, H, id, \mathsf{Ev}, v) \approx_L \cdot$

Proof (sketch):

By induction on the structure of $\mathcal{E} :: \mathsf{triggerEH}_{\mathcal{G}}(\sigma, H, id, \mathsf{Ev}, v) = E$

Want to show $E \approx_L \cdot$

Note that the rules are very similar for each enforcement mechanism and EH storage, so we do not consider them separately.

Case I: $\mathcal{E}$ ends in TRIGGEREH

By assumption, and from our security lattice, $E = (id.\mathsf{Ev}(v), H)$, therefore $E \approx_L \cdot$

Case II: $\mathcal{E}$ ends in TRIGGEREH-S

By assumption, $E = \cdot$, therefore $E \approx_L \cdot$

Case iii: $\mathcal{E}$ ends in TRIGGEREH-NC

By assumption, $pc = \cdot$, but we only want to consider cases where $pc = H$, so this case holds vacuously.

□

Requirement (EH3) H updates are unobservable

Lemma 189.

Unstructured EH storage: $\mathsf{assign}_{\mathcal{G}}(\sigma, H, id, v) \approx_L \sigma$

Tree structure EH storage: $\mathsf{assign}_{\mathcal{G}}(\sigma, H, a, v) \approx_L \sigma$

Proof (sketch): Only the cases for $\mathcal{G} \neq \mathsf{TS}$ are considered. The other cases are proven in the weak secrecy version: Lemma 191 and Lemma 192 (Requirement **(WEH3)**). The proof is straightforward. We examine each case of $\mathcal{E} :: \mathsf{assign}_{\mathcal{G}}(\sigma, H, id, v)$ (for the unstructured EH storage) and $\mathcal{E} :: \mathsf{assign}_{\mathcal{G}}(\sigma, H, a, v)$ (for the tree-structured EH storage) for each $\mathcal{G}$. In every case, only the H view is changed, so the resulting store is equivalent. □

Lemma 190. *For any $\mathcal{G}$, all of the following hold:*

Unstructured EH storage:

- $\mathsf{createElem}_{\mathcal{G}}(\sigma, H, id, v) \approx_L \sigma$
- $\mathsf{registerEH}_{\mathcal{G}}(\sigma, H, id, eh) \approx_L \sigma$

Tree structure EH storage:

- $\mathsf{createChild}_{\mathcal{G}}(\sigma, H, id, a_p, v) \approx_L \sigma$
- $\mathsf{createSibling}_{\mathcal{G}}(\sigma, H, id, a_s, v) \approx_L \sigma$
- $\mathsf{registerEH}_{\mathcal{G}}(\sigma, H, a, eh) \approx_L \sigma$

Proof (sketch): For createElem, only the cases for $\mathcal{G} \neq \mathsf{TS}$ are considered. The other cases are proven in the weak secrecy version: Lemma 194 and Lemma 195 (Requirement **(WEH3)**). We consider each $\mathcal{G}$ and EH store structure separately and then proceed by induction on the structure of $\mathcal{E} :: \mathsf{createElem}_{\mathcal{G}}(\sigma, H, id, v) \approx_L \sigma$, then $\mathcal{E} :: \mathsf{registerEH}_{\mathcal{G}}(\sigma, H, id, eh) \approx_L \sigma$, etc. The proof for createElem for the unstructured EH store also uses Lemma 189 (Requirement **(EH3)**). □

Requirement (WEH3) H updates are unobservable (Weak Secrecy)

Lemma 191. *If $\sigma_1 \approx_L \sigma_2$ and $\mathsf{assignW}_{\mathcal{G}}(\sigma_1, H, id, (v_1, l_1)) = (\sigma_1', \mathsf{gw}(x))$ and $\mathsf{assignW}_{\mathcal{G}}(\sigma_2, H, id, (v_2, l_2)) = (\sigma_2', \mathsf{gw}(x))$, then $\sigma_1' \approx_L \sigma_2'$*

Proof. Only the cases for $\mathcal{G} = \mathsf{TS}$ are considered. The other cases are not considered since TS is the only one which emits gw(_) events.

Denote $\mathcal{D} :: \mathsf{assignW}_{\mathcal{G}}(\sigma_1, H, id, (v_1, l_1))$ and $\mathcal{E} :: \mathsf{assignW}_{\mathcal{G}}(\sigma_2, H, id, (v_2, l_2))$.

From the assumption that $\mathcal{D}$ and $\mathcal{E}$ produce $\mathsf{gw}(x)$, $\mathcal{D}$ and $\mathcal{E}$ must end in TS-ASSIGN-GW

By assumption and from our security lattice, $\sigma_1' = \sigma_1[x \mapsto (v_1, H)]$ and $\sigma_2' = \sigma_2[x \mapsto (v_2, H)]$.

Therefore, from the assumption that $\sigma_1 \approx_L \sigma_2$, $\sigma_1' \approx_L \sigma_2'$. □

Lemma 192. *If* $\mathsf{assignW}_{\mathcal{G}}(\sigma, H, id, (v, l)) = (\sigma', \bullet)$*,then* $\sigma \approx_L \sigma'$

Proof.

Only the cases for $\mathcal{G} = \mathsf{TS}$ are considered. The other cases follow from Lemma 189 (Requirement **(EH3)**).

We examine each case of $\mathcal{E} :: \mathsf{assignW}_{\mathcal{G}}(\sigma, H, id, (v, l))$

Case I: $\mathcal{E}$ ends in TS-ASSIGN

By assumption and from our security lattice, $H \sqsubseteq \mathsf{labOf}(\sigma(x), H)$ and $\sigma' = \sigma[x \mapsto (v, H)]$

Then from the definition of $\approx_L$ for values, $\sigma \approx_L \sigma'$

Case II: $\mathcal{E}$ ends in TS-ASSIGN-S

By assumption, $\sigma' = \sigma$. Therefore, $\sigma \approx_L \sigma'$.

Case III: $\mathcal{E}$ ends in TS-ASSIGN-GW

We only consider cases which emit $\bullet$, therefore this case holds vacuously.

□

Lemma 193. *If* $\sigma_1 \approx_L \sigma_2$*, and* $\mathsf{assignW}_{\mathcal{G}}(\sigma_1, H, id, (v_1, l_1)) = (\sigma_1', \mathsf{gw}(id))$ *and* $\mathsf{createElemW}_{\mathcal{G}}(\sigma_2, H, id, (v_2, l_2)) = (\sigma_2', \mathsf{gw}(id))$ *then* $\sigma_1' \approx_L \sigma_2'$

Proof.

Only the cases for $\mathcal{G} = \mathsf{TS}$ are considered. The other cases are not considered since TS is the only one which emits $\mathsf{gw}(_)$ events.

Denote $\mathcal{D} :: \mathsf{assignW}_{\mathcal{G}}(\sigma_1, H, id, (v_1, l_1)) = (\sigma_1', \mathsf{gw}(id))$ and $\mathcal{E} :: \mathsf{createElemW}_{\mathcal{G}}(\sigma_2, H, id, (v_2, l_2)) = (\sigma_2', \mathsf{gw}(id))$

From the assumption that $\mathcal{D}$ and $\mathcal{E}$ produce $\mathsf{gw}(id)$, $\mathcal{D}$ must end in TS-ASSIGNEH-GW where (from our security lattice)

$\sigma_1(id) = (id, _, M, l_\phi)$ and $\sigma_1' = \sigma_1[id \mapsto (id, (v_1, H), M, l_\phi)]$

We examine each case of $\mathcal{E}$

Case I: $\mathcal{E}$ ends in TS-CREATE-U1-GW

By assumption and from our security lattice, all of the following:

$\mathsf{lookup}_{\mathsf{TS}}(\sigma_2, H, id) = (id, v', M, l')$, $\phi_2 = (id, (v_2, H), M, l')$, and $\sigma_2' = \sigma_2[id \mapsto \phi_2]$

Then, from $\sigma_1 \approx_L \sigma_2$ and since the node labels do not change in either assignment, $\sigma_1' \approx_L \sigma_2'$

Case II: $\mathcal{E}$ ends in TS-CREATE-NC

We only consider cases where $pc = H$, so this case holds vacuously.

Case III: $\mathcal{E}$ ends in TS-CREATE, TS-CREATE-U1, or TS-CREATE-U2

We only consider rules which emit $\mathsf{gw}(id)$, so these cases hold vacuously.

□

Lemma 194. *If* $\sigma_1 \approx_L \sigma_2$ *and* $\mathsf{createElemW}_{\mathcal{G}}(\sigma_1, H, id, (v_1, l_1)) = (\sigma_1', \mathsf{gw}(x))$ *and* $\mathsf{createElemW}_{\mathcal{G}}(\sigma_2, H, id, (v_2, l_2)) = (\sigma_2', \mathsf{gw}(x))$, *then* $\sigma_1' \approx_L \sigma_2'$

Proof.

Only the cases for $\mathcal{G} = \mathsf{TS}$ are considered. The other cases are not considered since TS is the only one which emits $\mathsf{gw}(_)$ events.

Denote $\mathcal{D} :: \mathsf{createElemW}_{\mathcal{G}}(\sigma_1, H, id, (v_1, l_1)) = (\sigma_1', \mathsf{gw}(id))$ and

$\mathcal{E} :: \mathsf{createElemW}_{\mathcal{G}}(\sigma_2, H, id, (v_2, l_2)) = (\sigma_2', \mathsf{gw}(id))$

We examine each case of $\mathcal{D}$

Case I: $\mathcal{D}$ ends in TS-CREATE-U1-GW

By assumption and from our security lattice, both of the following

$\mathsf{lookup}_{\mathsf{TS}}(\sigma_1, H, id) = (id, _, M_1, l_{\phi,1})$ and $\sigma_1' = \sigma_1[id \mapsto (id, (v_1, H), M_1, l_{\phi,1})$

From the assumption that $\mathcal{E}$ emits $\mathsf{gw}(id)$ and runs in the H context, $\mathcal{E}$ ends in TS-CREATE-U1-GW and

$\mathsf{lookup}_{\mathsf{TS}}(\sigma_2, H, id) = (id, _, M_2, l_{\phi,2})$ and $\sigma_2' = \sigma_2[id \mapsto (id, (v_2, H), M_2, l_{\phi,2})$

Then from $\sigma_1 \approx_L \sigma_2$ and the definition of $\approx_L$ for TS nodes, either

$l_{\phi_1} = l_{\phi_2} = H$; or $l_{\phi_1} = l_{\phi_2} = L$ and $M_1 \approx_L M_2$

Then from $\sigma_1 \approx_L \sigma_2$, $\sigma_1' \approx_L \sigma_2'$

Case II: $\mathcal{D}$ ends in TS-CREATE, TS-CREATE-U1, or TS-CREATE-U2

We only consider rules which emit $\mathsf{gw}(id)$, so these cases hold vacuously.

Case III: $\mathcal{D}$ ends in TS-CREATE-NC

We only consider cases where $pc = H$, so this case holds vacuously.

□

Lemma 195. *If* $\mathsf{createElemW}_{\mathcal{G}}(\sigma, H, id, (v, l)) = (\sigma', \bullet)$, *then* $\sigma \approx_L \sigma'$

Proof.

Only the cases for $\mathcal{G} = \mathsf{TS}$ are considered. The other cases follow from Lemma 190 (Requirement **(EH3)**).

We examine each case of $\mathcal{D} :: \mathsf{createElemW}_{\mathcal{G}}(\sigma, H, id, v) = (\sigma, \bullet)$

Case I: $\mathcal{D}$ ends in TS-CREATE

By assumption, $\mathsf{lookup}_{\mathsf{TS}}(\sigma, H, id) = (\mathsf{NULL}, _)$ and $\sigma' = \sigma[id \mapsto (id, (v, l), \cdot, H)]$

Then from the definition of lookup for TS, $id \notin \sigma$

Thus, $\sigma \approx_L \sigma'$

Case II: $\mathcal{D}$ ends in TS-CREATE-U1

By assumption and from our security lattice, all of the following:

$\mathsf{lookup}_{\mathsf{TS}}(\sigma, H, id) = (id, (v', l'), M, l_\phi)$, $\sigma' = \sigma_1[id \mapsto (id, (v, H), M, l_\phi)$, and $H \sqsubseteq l'$

Thus, $\sigma \approx_L \sigma'$

Case III: $\mathcal{D}$ ends in TS-CREATE-U2

By assumption, $\mathsf{lookup}_{\mathsf{TS}}(\sigma, H, id) = (id, v', M, l')$ with $l' \not\sqsubseteq H$

But from our security lattice, such an l' does not exist. So this case holds vacuously.

Case IV: $\mathcal{D}$ ends in TS-CREATE-U1-GW

We only consider rules which emit • so this case holds vacuously.

Case V: $\mathcal{D}$ ends in TS-CREATE-NC

We only consider cases where $pc = H$ so this case holds vacuously.

□

Requirement (EH4) L updates are equivalent

Lemma 196. *If $\sigma_1 \approx_L \sigma_2$, $pc_1, pc_2 \sqsubseteq L$, then*

Unstructured EH storage: *if $v_1 \approx_L v_2$, then* $\mathsf{assign}_{\mathcal{G}}(\sigma_1, pc_1, id, v_1) \approx_L \mathsf{assign}_{\mathcal{G}}(\sigma_2, pc_2, id, v_2)$

Tree structure EH storage: *if $v_1 \approx_L^{\sigma_1,\sigma_2} v_2$, and $a_1 \approx_L^{\sigma_1,\sigma_2} a_2$, then* $\mathsf{assign}_{\mathcal{G}}(\sigma_1, pc_1, a, v_1) \approx_L \mathsf{assign}_{\mathcal{G}}(\sigma_2, pc_2, a, v_2)$

Proof (sketch): Only the cases for $\mathcal{G} \neq \mathsf{TS}$ are considered. The other cases are proven in the weak secrecy version: Lemma 198. The proof is by induction on the structure of $\mathcal{D} :: \mathsf{assign}_{\mathcal{G}}(\sigma_1, pc_1, id, v_1) = \sigma_1'$ and $\mathcal{E} :: \mathsf{assign}_{\mathcal{G}}(\sigma_2, pc_2, id, v_2) = \sigma_2'$ The unstructured DOM also uses Lemma 189.U (Requirement **(EH3)**) for the assignments to the H copy of the multi-store, and the tree-structured DOM uses Lemma 189.T for the same reason. Meanwhile the unstructured DOM uses Lemma 164.U to show the DOM nodes are equivalent after lookup, while the tree-structured DOM uses the assumption that $a_1 \approx_L^{\sigma_1,\sigma_2} a_2$. □

Lemma 197. *If $\sigma_1 \approx_L \sigma_2$, $pc_1, pc_2 \sqsubseteq L$, then*

Unstructured EH storage: *If $v_1 \approx_L v_2$, then*

- $\mathsf{createElem}_{\mathcal{G}}(\sigma_1, pc_1, id, v_1) \approx_L \mathsf{createElem}_{\mathcal{G}}(\sigma_2, pc_2, id, v_2)$
- $\mathsf{registerEH}_{\mathcal{G}}(\sigma_1, pc_1, id, eh) \approx_L \mathsf{registerEH}_{\mathcal{G}}(\sigma_2, pc_2, id, eh)$

Tree structure EH storage: *If $v_1 \approx_L^{\sigma_1,\sigma_2} v_2$, $a_{p,1} \approx_L^{\sigma_1,\sigma_2} a_{p,2}$, $a_{s,1} \approx_L^{\sigma_1,\sigma_2} a_{s,2}$, and $a_1 \approx_L^{\sigma_1,\sigma_2} a_2$, then*

- $\mathsf{createChild}_{\mathcal{G}}(\sigma_1, pc_1, id, a_{p,1}, v_1) \approx_L \mathsf{createChild}_{\mathcal{G}}(\sigma_2, pc_2, id, a_{p,1}, v_2)$
- $\mathsf{createSibling}_{\mathcal{G}}(\sigma_1, pc_1, id, a_{s,1}, v_1) \approx_L \mathsf{createSibling}_{\mathcal{G}}(\sigma_2, pc_2, id, a_{s,2}, v_2)$
- $\mathsf{registerEH}_{\mathcal{G}}(\sigma_1, pc_1, a_1, eh) \approx_L \mathsf{registerEH}_{\mathcal{G}}(\sigma_2, pc_2, a_2, eh)$

Proof (sketch): Only the cases for $\mathcal{G} \neq \mathsf{TS}$ are considered. The other cases are proven in the weak secrecy version: Lemma 199. We consider each $\mathcal{G}$ and EH store structure separately and then proceed by induction on the structure of $\mathcal{D}$ and $\mathcal{E}$ for $\mathcal{D} :: \mathsf{createElem}_{\mathcal{G}}(\sigma_1, pc_1, id, v_1)$ and $\mathcal{E} :: \mathsf{createElem}_{\mathcal{G}}(\sigma_2, pc_2, id, v_2)$, then $\mathcal{D} :: \mathsf{registerEH}_{\mathcal{G}}(\sigma_1, pc_1, id, eh)$ and $\mathcal{E} :: \mathsf{registerEH}_{\mathcal{G}}(\sigma_2, pc_2, id, eh)$, etc. Lemma 164.U is used for the unstructured DOM to show that DOM nodes are equivalent after lookup (we use Lemma 165 (Requirement **(EH1)**) for a similar reason for the tree-structured DOM). We also use Lemma 196.U (Requirement **(EH4)**) to show that updates to existing DOM nodes result in equivalent stores. The unstructured DOM also uses Lemma 190.U (Requirement **(EH3)**) for assignments to the H copy of the multi-store (we use Lemma 190.T (Requirement **(EH3)**) for the tree-structured DOM). □

Requirement (WEH4) L updates are equivalent (Weak Secrecy)

Lemma 198. *If* $\sigma_1 \approx_L \sigma_2$, $pc_1, pc_2 \sqsubseteq L$, $v_1 \approx_L v_2$, *and* $\alpha_1 = \alpha_2$, *then for* $\mathsf{assign}_{\mathcal{G}}(\sigma_1, pc_1, id, v_1) = (\sigma_1', \alpha_1)$ *and* $\mathsf{assign}_{\mathcal{G}}(\sigma_2, pc_2, id, v_2) = (\sigma_2', \alpha_2)$, $\sigma_1' \approx_L \sigma_2'$

Proof.

Only the cases for $\mathcal{G} = \mathsf{TS}$ are considered. The other cases are not considered since they follow from Lemma 196

We examine each case of $\mathcal{D} :: \mathsf{assign}_{\mathcal{G}}(\sigma_1, pc_1, id, v_1) = (\sigma_1', \alpha_1)$

Denote $\mathcal{E} :: \mathsf{assign}_{\mathcal{G}}(\sigma_2, pc_2, id, v_2) = (\sigma_2', \alpha_2)$

We refer to the following assumptions throughout:

(1) $\sigma_1 \approx_L \sigma_2$; (2) $pc_1, pc_2 \sqsubseteq L$; and (3) $v_1 \approx_L v_2$

From $\mathcal{G} = \mathsf{TS}$ and (3), we also refer to:

(4) $v_1 = (v_1', l_1)$ and $v_2 = (v_2', l_2)$ and (5) $(v_1', l_1) \approx_L (v_2', l_2)$

Case I: $\mathcal{D}$ ends in TS-ASSIGNEH

By assumption, all of the following:

$\sigma_1(id) = (id, (v_1'', l_1''), M_1, l_1')$, $\sigma_1' = \sigma_1[id \mapsto (id, (v_1', l_1 \sqcup pc_1 \sqcup l_1'), M_1, l_1')]$, $\alpha_1 = \bullet$, and $l_1 \sqcup pc_1 \sqsubseteq l_1''$

Then from (2), $l_1 \sqsubseteq l_1''$

Subcase i: $l_1' \sqsubseteq L$

By assumption and from (1), all of the following:

$\sigma_2(id) = (id, (v_2'', l_2''), M_2, l_2')$ with $l_2' \sqsubseteq L$, $(v_1'', l_1'') \approx_L (v_2'', l_2'')$, and $M_1 \approx_L M_2$

Then, from $l_1 \sqsubseteq l''_1$, (5) and $(v''_1, l''_1) \approx_L (v''_2, l''_2)$, it must be the case that $l_2 \sqsubseteq l''_2$

Then from (2) and our security lattice, $l_2 \sqcup pc_2 \sqsubseteq l''_2$

Then, $\mathcal{E}$ ends in TS-ASSIGNEH with $\sigma'_2 = \sigma_2[id \mapsto (id, (v'_2, l_2 \sqcup pc_2 \sqcup l'_2), M_2)]$ and $\alpha_2 = \bullet$

From (1), (2), (5), and $l'_1, l'_2 \sqsubseteq L$ $\sigma'_1 \approx_L \sigma'_2$

And from $\alpha_1 = \bullet$ and $\alpha_2 = \bullet$, we know that $\alpha_1 = \alpha_2$

Subcase ii: $l'_1 \not\sqsubseteq L$

By assumption and from (I.2), (I.5) and our security lattice, $l''_1 = H$ and $\sigma'_1 \approx_L \sigma'_2$

If $id \notin \sigma_2$, then from TS-ASSIGNEH-S, $\sigma'_2 = \sigma_2$ and $\alpha_2 = \bullet$

Then, in this case we know $\sigma'_1 \approx_L \sigma'_2$ and $\alpha_1 = \alpha_2$

Otherwise, $id \in \sigma_2$, and by assumption and from (1), $\sigma_2(id) = (id, (v''_2, l''_2), M_2, l'_2)$ with $l'_2 \not\sqsubseteq L$

Then, $\mathcal{E}$ ends in TS-ASSIGNEH with

$\sigma'_2 = \sigma_2[id \mapsto (id, (v'_2, l_2 \sqcup pc_2 \sqcup l'_2), M_2, l'_2)]$ and $\alpha_2 = \bullet$

Then, from $l'_1, l'_2 \not\sqsubseteq l$, we know that $\sigma'_1 \approx_L \sigma'_2$

And from $\alpha_1 = \bullet$ and $\alpha_2 = \bullet$, we know that $\alpha_1 = \alpha_2$

Case II: $\mathcal{D}$ ends in TS-ASSIGNEH-S

By assumption, all of the following:

$id \notin \sigma_1$, $\sigma'_1 = \sigma_1$, and $\alpha_1 = \bullet$

From (1) and $id \notin \sigma_1$, either

$id \notin \sigma_2$; or $\mathsf{labOf}(\sigma_2(id), _) = H$

In the first case, $\mathcal{E}$ ends in TS-ASSIGNEH-S with $\sigma'_2 = \sigma_2$ and $\alpha_2 = \bullet$

Thus, from $\sigma'_1 = \sigma_1$, and $\alpha_1 = \bullet$, we know that $\sigma'_1 \approx_L \sigma'_2$ and $\alpha_1 = \alpha_2$

In the other case, $\mathcal{E}$ ends in TS-ASSIGNEH with

$\sigma'_2 = \sigma_2[id \mapsto (id, (v, l_2 \sqcup pc_2 \sqcup H), \sigma_2(id).M, H)]$ and $\alpha_2 = \bullet$

Then from $\mathsf{labOf}(\sigma_2(id), _) = H$, we know that $\sigma_2 \approx_L \sigma'_2$

From (1), $\sigma'_1 = \sigma_1$, and $\sigma_2 \approx_L \sigma'_2$, we know that $\sigma'_1 \approx_L \sigma'_2$

And from $\alpha_1 = \bullet$ and $\alpha_2 = \bullet$, we know that $\alpha_1 = \alpha_2$

Case III: $\mathcal{D}$ ends in TS-ASSIGNEH-GW

By assumption, all of the following:

$\sigma_1(id) = (id, (v''_1, l''_1), M_1, l'_1)$ with $l'_1 \sqsubseteq L$ and $l_1 \sqcup pc_1 \not\sqsubseteq l''_1$

$\sigma'_1 = \sigma_1[id \mapsto (id, (v'_1, l_1 \sqcup pc_1 \sqcup l'_1), M_1, l'_1)]$, and $\alpha_1 = \mathsf{gw}(id)$

Then from (1), we know all of the following:

$\sigma_2(id) = (id, (v''_2, l''_2), M_2, l'_2)$ with $l'_2 \sqsubseteq L$ and $M_1 \approx_L M_2$, and

$(v_1'', l_1'') \approx_L (v_2'', l_2'')$

From (2), $l_1 \sqcup pc_1 \not\sqsubseteq l_1''$, and our security lattice, $l_1 = H$ and $l_1'' = L$

Then, from (5), we know that $l_2 = H$

And from $(v_1'', l_1'') \approx_L (v_2'', l_2'')$, we know that $l_2'' = L$

From (2) and our security lattice, $\mathcal{E}$ must end in TS-ASSIGNEH-GW with

$\sigma_2' = \sigma_2[id \mapsto (id, (v_2', l_2 \sqcup pc_2 \sqcup l_2'), M_2, l_2')]$ and $\alpha_2 = \mathsf{gw}(id)$

From $l_1, l_2 = H$ and our security lattice,

$(v_1', l_1 \sqcup pc_1 \sqcup l_1') \approx_L (v_2', l_2 \sqcup pc_2 \sqcup l_2')$

Then from $M_1 \approx_L M_2$ and $l_1', l_2' \sqsubseteq L$, we know that $\sigma_1' \approx_L \sigma_2'$

And from $\alpha_1 = \mathsf{gw}(id)$ and $\alpha_2 = \mathsf{gw}(id)$, we know that $\alpha_1 = \alpha_2$

□

Lemma 199. *If $\sigma_1 \approx_L \sigma$, and $pc_1, pc_2 \sqsubseteq L$ and $v_1 \approx_L v_2$, then for $(\sigma_1', \alpha_1) = \mathsf{createElem}_{\mathcal{G}}(\sigma_1, pc_1, id, v_1)$ and $(\sigma_2', \alpha_2) = \mathsf{createElem}_{\mathcal{G}}(\sigma_2, pc_2, id, v_2)$, $\sigma_1' \approx_L \sigma_2'$ and $\alpha_1 = \alpha_2$*

Proof.

Only the cases for $\mathcal{G} = \mathsf{TS}$ are considered. The other cases are not considered since they follow from Lemma 197

By induction on the structure of $\mathcal{D} :: \mathsf{createElem}_{\mathcal{G}}(\sigma_1, pc_1, id, v_1) = (\sigma_1', \alpha_1)$ and $\mathcal{E} :: \mathsf{createElem}_{\mathcal{G}}(\sigma_2, pc_2, id, v_2) = (\sigma_2', \alpha_2)$

We refer to the following assumptions throughout:

(1) $\sigma_1 \approx_L \sigma_2$; (2) $pc_1, pc_2 \sqsubseteq L$; and (3) $v_1 \approx_L v_2$

From $\mathcal{G} = \mathsf{TS}$ and (3), we also refer to:

(4) $v_1 = (v_1', l_1)$ and $v_2 = (v_2', l_2)$ and (5) $(v_1', l_1) \approx_L (v_2', l_2)$

Case I: $\mathcal{D}$ ends in TS-CREATE

By assumption, all of the following:

$\mathsf{lookup}_{\mathsf{TS}}(\sigma_1, L, id) = (\mathsf{NULL}, _)$, $\sigma_1' = \sigma_1[id \mapsto (id, (v_1', l_1), \cdot, L)]$, and $\alpha_1 = \bullet$

Then from Lemma 164 (Requirement **(EH1)**), $\mathsf{lookup}_{\mathsf{TS}}(\sigma_2, L, id) \approx_L (\mathsf{NULL}, _)$

Then either

(I.1) $\mathsf{lookup}_{\mathsf{TS}}(\sigma_2, L, id) = (\mathsf{NULL}, _)$; or (I.2) $\mathsf{lookup}_{\mathsf{TS}}(\sigma_2, L, id) = (id, (v_2'', l_2''), M, H)$

Subcase i: (I.1) is true

From (I.1), $\mathcal{E}$ must end in TS-CREATE with

$\sigma_2' = \sigma_2[id \mapsto (id, (v_2', l_2), \cdot, L)]$ and $\alpha_2 = \bullet$

Then from (1), (5), and $\sigma_1' = \sigma_1[id \mapsto (id, (v_1', l_1), \cdot, L)]$, we know that $\sigma_1' \approx_L \sigma_2'$

And from $\alpha_1 = \bullet$ and $\alpha_2 = \bullet$, we know that $\alpha_1 = \alpha_2$

Subcase ii: (I.2) is true

From (I.2) and (2), $\mathcal{E}$ ends in TS-CREATE-U2 with all of the following:

$\mathsf{lookup}_{\mathsf{TS}}(\sigma_2, L, id) = (id, (v''_2, l''_2), M_2, H)$, $\sigma'_2 = \sigma_2[id \mapsto (id, (v'_2, l_2), M_2, L)$, and $\alpha_2 = \bullet$

From Lemma 200, $M_2 \approx_L \cdot$

Then from (1), (5), and $\sigma'_1 = \sigma_1[id \mapsto (id, (v'_1, l_1), \cdot, L)]$, we know that $\sigma'_1 \approx_L \sigma'_2$

And from $\alpha_1 = \bullet$ and $\alpha_2 = \bullet$, we know that $\alpha_1 = \alpha_2$

Subcase iii: $pc_2 = \cdot$

The proof for this case follows from applying the IH on $\mathcal{D}$ and the premise of $\mathcal{E}$.

Case II: $\mathcal{E}$ ends in TS-CREATE

The proof is similar to **Case I**.

Case III: $\mathcal{D}$ ends in TS-CREATE-U1

By assumption, all of the following:

$\mathsf{lookup}_{\mathsf{TS}}(\sigma_1, L, id) = (id, (v''_1, l''_1), M_1, l'_1)$, $l'_1 \sqsubseteq L$, $\sigma'_1 = \sigma_1[id \mapsto (id, (v'_1, l_1 \sqcup L \sqcup l'_1), M_1, l'_1)]$, $\alpha_1 = \bullet$, and

(III.1) $l_1 \sqcup L \sqsubseteq l''_1$ or $l'_1 \not\sqsubseteq L$

From $l'_1 \sqsubseteq L$ and since TS never contains $\cdot$, we know that $l'_1 = L$

Then from (III.1) and from our security lattice, $l_1 \sqsubseteq l''_1$

From Lemma 164 (Requirement **(EH1)**), $\mathsf{lookup}_{\mathsf{TS}}(\sigma_2, L, id) = (id, (v''_2, l''_2), M_2, l'_2)$ with

$l'_2 \sqsubseteq L$, $(v''_1, l''_1) \approx_L (v''_2, l''_2)$ and $M_1 \approx_L M_2$

Subcase i: $pc_2 = L$

By assumption, $\mathcal{E}$ ends in TS-CREATE-U1 or TS-CREATE-U1-GW

From (5), $l''_1 = l''_2$ and $l_1 = l_2$

Then from $l_1 \sqsubseteq l''_1$, we know that $l_2 \sqsubseteq l''_2$

Then, $\mathcal{E}$ must end in TS-CREATE-U1 with $\sigma'_2 = \sigma_2[id \mapsto (id, (v'_2, l_2 \sqcup L \sqcup l'_2), M_2, l'_2)]$ and $\alpha_2 = \bullet$

Then, from $\sigma'_1 = \sigma_1[id \mapsto (id, (v'_1, l_1 \sqcup L \sqcup l'_1), M_1, l'_1)]$ and $\sigma'_2 = \sigma_2[id \mapsto (id, (v'_2, l_2 \sqcup L \sqcup l'_2), M_2, l'_2)]$,

we know that $\sigma'_1 \approx_L \sigma'_2$

And from $\alpha_1 = \bullet$ and $\alpha_2 = \bullet$, we know that $\alpha_1 = \alpha_2$

Subcase ii: $pc_2 = \cdot$

The proof for this case follows from applying the IH on $\mathcal{D}$ and the premise of $\mathcal{E}$.

Case IV: $\mathcal{E}$ ends in TS-CREATE-U1 or TS-CREATE-U1-GW

The proof is similar to **Case III**.

Case V: $\mathcal{D}$ ends in TS-CREATE-U1-GW

The proof is similar to **Case III**

Case VI: $\mathcal{D}$ ends in TS-CREATE-U2

By assumption, all of the following:

$\mathsf{lookup}_{\mathsf{TS}}(\sigma_1, L, id) = (id, (v_1'', l_1''), M_1, l_1')$, $l_1' \not\sqsubseteq L$, $\sigma_1' = \sigma_1[id \mapsto (id, (v_1', l_1 \sqcup L), M_1, L)]$, and $\alpha_1 = \bullet$

From $l_1' \not\sqsubseteq L$ and our security lattice, $l_1' = H$

The from Lemma 164 (Requirement **(EH1)**), either

(VI.1) $\mathsf{lookup}_{\mathsf{TS}}(\sigma_2, L, id) = (id, (v_2'', l_2''), M_2, l_2')$ with $l_2' = H$; or

(VI.2) $\mathsf{lookup}_{\mathsf{TS}}(\sigma_2, L, id) = (\mathsf{NULL}, H)$

Subcase i: (VI.1) is true

From (VI.1), $\mathcal{E}$ ends in TS-CREATE-U2 with

$\sigma_2' = \sigma_2[id \mapsto (id, (v_2', l_2 \sqcup L), M_2, L)]$ and $\alpha_2 = \bullet$

From Lemma 200, $M_1 \approx_L M_2 \approx_L \cdot$

From (1), (5), $\sigma_1' = \sigma_1[id \mapsto (id, (v_1', l_1 \sqcup L), M_1, L)]$, and $\sigma_2' = \sigma_2[id \mapsto (id, (v_2', l_2 \sqcup L), M_2, L)]$,

we know that $\sigma_1' \approx_L \sigma_2'$

And from $\alpha_1 = \bullet$ and $\alpha_2 = \bullet$, we know that $\alpha_1 = \alpha_2$

Subcase ii: (VI.2) is true

From (VI.2), $\mathcal{E}$ ends in TS-CREATE with $\sigma_2' = \sigma_2[id \mapsto (id, (v_2', l_2), \cdot, L)]$ and $\alpha_2 = \bullet$

From Lemma 200, $M_1 \approx_L \cdot$

From (1), (5), $\sigma_1' = \sigma_1[id \mapsto (id, (v_1', l_1 \sqcup L), M_1, L)]$, and $\sigma_2' = \sigma_2[id \mapsto (id, (v_2', l_2), \cdot, L)]$, we know that $\sigma_1' \approx_L \sigma_2'$

And from $\alpha_1 = \bullet$ and $\alpha_2 = \bullet$, we know that $\alpha_1 = \alpha_2$

Subcase iii: $pc_2 = \cdot$

The proof for this case follows from applying the IH on $\mathcal{D}$ and the premise of $\mathcal{E}$.

Case VII: $\mathcal{E}$ ends in TS-CREATE-U2

The proof is similar to **Case VI**.

Case VIII: $\mathcal{D}$ or $\mathcal{E}$ ends in TS-CREATE-NC

The proof for this case follows from applying the IH on the premise of $\mathcal{D}$ and $\mathcal{E}$.

□

Lemma 200. *If $\phi = (id, v, M, H)$ were produced by our operational semantics starting in a well-formed state, then $M \approx_L \cdot$*

Proof (sketch): Starting in a well-formed state, the invariant holds, since all event handlers registered to H

nodes are labeled H. New nodes created with label H also satisfy the invariant since they are created with an empty event handler map.

The only way to register new event handlers is through one of the registerEH rules. TS-REGISTEREH adds the event handler to the event handler map and performs a join with the label of the node; thus, any new event handler will have label H if the label of the node is H.

The only rule which changes the label of a node is TS-CREATE-U2. This rule changes the label of the node to match the *pc* only when the label is *above* the *pc*, i.e., it changes the label from H to L. Therefore, the invariant that tainted nodes never have publicly visible event handlers is maintained by our semantics. □

C.4.7 Progress-insensitive Noninterference for TT

Theorem 30 (Soundness (TT)). *If event handlers are enforced with* $\mathcal{V} \in \{\mathsf{TT}, \mathsf{SME}, \mathsf{MF}\}$ *and the global storage is enforced with* $G \in \{\mathsf{SMS}, \mathsf{FS}\}$*, then the composition of these event handlers and global stores in our framework satisfies progress-insensitive noninterference.*

Proof (sketch): The proof follows the same format as the proof for Theorem 25. We consider two cases: one where the last event was a declassification, and another where it was not. In the case that the last event was a declassification, the proof follows from the definition of $\mathcal{K}_{rp}()$. When the last event was not a declassification, the proof follows from Lemma 123 (Requirement **(T1)**) and Lemma 133 (Requirement **(T4)**). □

Recall that we structure our requirements to be extensible and easily updated. Here, we outline all of the supporting lemmas for proving Theorem 30 and highlight the ones which need updates to prove compositions with TT secure.

Trace Requirements The proof for Requirement **(T1)** does not need to be changed to prove PINI for TT. It follows from Lemma 125 (Requirement **(T2)**) and Lemma 144 (Requirement **(T5)**).

The proof for Requirement **(T2)** does not need to be changed to prove PINI for TT. It follows from Lemma 182 (Requirement **(EH2)**), Lemma 157 (Requirement **(V2)**), Lemma 189 (Requirement **(EH3)**), and Lemma 190 (Requirement **(EH3)**).

The proof for Requirement **(T3)** does not need to be changed to prove PINI for TT. It follows from Lemma 126 (Requirement **(T2)**) and Lemma 182 (Requirement **(EH2)**).

The proof for Requirement **(T4)** does not need to be changed to prove PINI for TT. It follows from Lemma 172 and Lemma 179 (Requirement **(EH1)**), Lemma 131 and Lemma 132 (Requirement **(T3)**),

Lemma 182 (Requirement **(EH2)**), Lemma 152 (Requirement **(E1)**), Lemma 160 (Requirement **(V3)**), and Lemma 196 and Lemma 197 (Requirement **(EH4)**).

The proof for Requirement **(T5)** does not need to be changed to prove PINI for TT. It follows from Lemma 172 and Lemma 179 (Requirement **(EH1)**), Lemma 182 (Requirement **(EH2)**), Lemma 152 (Requirement **(E1)**), Lemma 160 (Requirement **(V3)**), and Lemma 196 and Lemma 197 (Requirement **(EH4)**).

Expression Requirements The proof for Requirement **(E1)** does not need to be changed to prove PINI for TT. It follows from Lemma 154 and Lemma 155 (Requirement **(V1)**) and Lemma 162 (Requirement **(EH1)**).

Variable Store Requirements The proofs for Requirement **(V1)** need to be updated to prove TT is secure, but the proofs for Requirements **(V2)** and **(V3)** (global variable storage and variable assignment, respectively) do not need to be changed, nor do they depend on any other requirements. We outline the changes for Requirement **(V1)** below.

Note that we also do not need to add/change any proofs to say that assignments do not leak anything. This is because assignments in the H context can only leak through the global store, which is already proven secure by Requirement **(V2)** for multi-storage techniques. Intuitively, assignments from TT would either change the H copy of the store (which does not leak anything) or would be replaced with a default value in the L copy of the store (which also does not leak anything).

EH storage Requirements None of the proofs for the event handler storage requirements need to be changed to prove that TT satisfies PINI security, nor do they depend on any other requirements that need to be changed.

Note: When we compose TT with a multi-storage technique, the TT event handlers in the L context no longer receive secrets (i.e., none of the values become tainted). So, there is also no reason to taint default values. We change TT-VAR-DV so that it returns dv labeled with the pc. This also helps maintain the invariant that the local TT store does not have any tainted values when the $pc = L$ (Lemma 202).

Lemma 201. *If* $\sigma_1 \approx_L \sigma_2$ *and* $pc_1, pc_2 \sqsubseteq L$, *then for* $\mathsf{var}_{\mathsf{TT}}(\sigma_1, pc_1, x) = v_1$ *and* $\mathsf{var}_{\mathsf{TT}}(\sigma_2, pc_2, x) = v_2$ *then* $v_1 \simeq_L v_2$

Proof.

By induction on the structure of $\mathcal{E} :: \mathsf{var}_{\mathsf{TT}}(\sigma_1, pc_1, x) = v_1$ and $\mathcal{D} :: \mathsf{var}_{\mathsf{TT}}(\sigma_2, pc_2, x) = v_2$

From Lemma 202, $\forall x \in (\sigma_1^{\mathsf{TT}}, \sigma_2^{\mathsf{TT}})\mathsf{labOf}(x, _) = L$

Then, $\sigma_1 = \sigma_2$

Case I: $\mathcal{E}$ ends in TT-VAR

By assumption and from $\forall x \in (\sigma_1^{\mathsf{TT}}, \sigma_2^{\mathsf{TT}})\mathsf{labOf}(x, _) = L$ we know that $x \in \sigma_1$ and $\sigma_1(x) = (v_1, L)$

Then, from $\sigma_1 = \sigma_2$, $\mathcal{D}$ ends in TT-VAR with $\sigma_2(x) = (v_2, L)$

From $\sigma_1 = \sigma_2$, $\sigma_1(x) = (v_1, L)$, and $\sigma_2(x) = (v_2, L)$, the desired conclusion holds

Case II: $\mathcal{D}$ ends in TT-VAR

The proof is similar to **Case I**

Case III: $\mathcal{E}$ ends in TT-VAR-DV

By assumption and from (2), $x \notin \sigma_1$ and $v_1 = (\mathsf{dv}, L)$

Then, from $\sigma_1 = \sigma_2$, we know that $x \notin \sigma_2$

From this, $\mathcal{D}$ must end in TT-VAR-DV with $v_2 = (\mathsf{dv}, L)$

From $v_1 = (\mathsf{dv}, L)$ and $v_2 = (\mathsf{dv}, L)$, the desired conclusion holds

Case IV: $\mathcal{D}$ ends in TT-VAR-DV

The proof is similar to **Case III**

□

Lemma 202. *Whenever a public event handler is running,* $\forall x \in \sigma^{\mathsf{TT}}, \mathsf{labOf}(x, _) = L$

Proof (sketch): To prove that the local store only contains public values while public event handlers are running, we need to show that the condition holds when the event handler begins and is maintained until the event handler finishes.

When the event handler begins running, the local store is empty, so the condition holds trivially.

As the event handler runs, the store is changed by ASSIGN-L. The value assigned is given by the expression semantics. From the assumption that the local store only contains public values, and from Lemma 152 (for expressions involving shared variables), the value being assigned is also public. Therefore, the condition is maintained throughout the event handler execution. □

Bibliography

[1] Johan Agat. Transforming out timing leaks. In *Proceedings of the 27th ACM Symposium on Principles of Programming Languages*, POPL '00, 2000. 5

[2] Amir A. Ahmadian and Musard Balliu. Dynamic policies revisited. In *Proceedings of the 2022 IEEE 7th European Symposium on Security and Privacy*, EuroSP '22, 2022. 9

[3] Maximilian Algehed and Cormac Flanagan. Transparent IFC enforcement: Possibility and (in)efficiency results. In *Proceedings of the 2020 IEEE Computer Security Foundations Symposium*, CSF '20, 2020. 7, 39

[4] Maximilian Algehed, Alejandro Russo, and Cormac Flanagan. Optimising faceted secure multi-execution. In *Proceedings of the 2019 IEEE Computer Security Foundations Symposium*, CSF '19, 2019. 7

[5] Ana Gualdina Almeida Matos, José Fragoso Santos, and Tamara Rezk. An information flow monitor for a core of DOM: Introducing references and live primitives. In *Proceedings of the International Symposium on Trustworthy Global Computing*, TGC '14, 2014. 1, 93

[6] Kalev Alpernas, Cormac Flanagan, Sadjad Fouladi, Leonid Ryzhyk, Mooly Sagiv, Thomas Schmitz, and Keith Winstein. Secure serverless computing using dynamic information flow control. In *Proceedings of the 2018 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '18, 2018. 95

[7] Aslan Askarov and Stephen Chong. Learning is change in knowledge: Knowledge-based security for dynamic policies. In *Proceedings of the 2012 IEEE Computer Security Foundations Symposium*, CSF '12, 2012. 9, 11

[8] Aslan Askarov, Stephen Chong, and Heiko Mantel. Hybrid monitors for concurrent noninterference. In *Proceedings of the 2015 IEEE Computer Security Foundations Symposium*, CSF '15, 2015. 1, 95

[9] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proceedings of the European Symposium on Research in Computer Security*, ESORICS '08, 2008. 5, 72

[10] Aslan Askarov and Andrew Myers. Attacker control and impact for confidentiality and integrity. *Logical Methods in Computer Science*, 7(3), 2011. 11, 59

[11] Aslan Askarov and Andrei Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, 2007. 10, 14

[12] Aslan Askarov and Andrei Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Proceedings of the 2009 IEEE Computer Security Foundations Symposium*, CSF '09, 2009. 2

[13] Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In *Proceedings of the 2009 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '09, 2009. 1, 2, 8, 69, 71

[14] Thomas H. Austin and Cormac Flanagan. Permissive dynamic information flow analysis. In *Proceedings of the 2010 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '10, 2010. 1, 2, 8, 69, 95

[15] Thomas H. Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In *Proceedings of the 39th ACM Symposium on Principles of Programming Languages*, POPL '12, 2012. 1, 2, 6, 69

[16] Thomas H Austin, Tommy Schmitz, and Cormac Flanagan. Multiple facets for dynamic information flow with exceptions. *ACM Transactions on Programming Languages and Systems*, 39(3), 2017. 7

[17] Thomas H. Austin, Jean Yang, Cormac Flanagan, and Armando Solar-Lezama. Faceted execution of policy-agnostic programs. In *Proceedings of the 2013 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '13, 2013. 2, 7, 69

[18] Musard Balliu. A logic for information flow analysis of distributed programs. In *Proceedings of the Nordic Conference on Secure IT Systems*, NordSec '13, 2013. 9

[19] Anindya Banerjee, David A Naumann, and Stan Rosenberg. Expressive declassification policies and modular static enforcement. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, 2008. 10, 22

[20] Iulia Bastys, Musard Balliu, and Andrei Sabelfeld. If this then what? Controlling flows in IoT apps. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, 2018. 1, 3, 69

[21] Iulia Bastys, Frank Piessens, and Andrei Sabelfeld. Tracking information flow via delayed output. In *Proceedings of the Nordic Conference on Secure IT Systems*, NordSec '18, 2018. 1, 69

[22] Lujo Bauer, Shaoying Cai, Limin Jia, Timothy Passaro, Michael Stroucken, and Yuan Tian. Run-time monitoring and formal analysis of information flows in Chromium. In *Proceedings of the 2015 Network and Distributed System Security Symposium*, NDSS '15, 2015. 1, 2, 6, 8, 13, 69, 94

[23] Abhishek Bichhawat, Vineet Rajani, Deepak Garg, and Christian Hammer. Information flow control in WebKit's JavaScript bytecode. In *Proceedings of the International Conference on Principles of Security and Trust*, POST '14, 2014. 6

[24] Abhishek Bichhawat, Vineet Rajani, Jinank Jain, Deepak Garg, and Christian Hammer. WebPol: Fine-grained information flow policies for web browsers. In *Proceedings of the European Symposium on Research in Computer Security*, ESORICS '17, 2017. 1, 2, 6, 8, 13, 69

[25] Nataliia Bielova, Dominique Devriese, Fabio Massacci, and Frank Piessens. Reactive non-interference for a browser model. In *Proceedings of the International Conference on Network and System Security*, NSS '11, 2011. 4, 93

[26] Nataliia Bielova and Tamara Rezk. Spot the difference: Secure multi-execution and multiple facets. In *Proceedings of the European Symposium on Research in Computer Security*, ESORICS '16, 2016. 6, 68

[27] Aaron Bohannon and Benjamin C. Pierce. Featherweight Firefox: Formalizing the core of a web browser. In *Proceedings of the 2010 USENIX Conference on Web Application Development*, WebApps '10, 2010. 94

[28] Aaron Bohannon, Benjamin C Pierce, Vilhelm Sjöberg, Stephanie Weirich, and Steve Zdancewic. Reactive noninterference. In *Proceedings of the 2009 ACM Conference on Computer and Communications Security*, CCS '09, 2009. 2, 3, 17

[29] Iulia Bolos§teanu and Deepak Garg. Asymmetric secure multi-execution with declassification. In *Proceedings of the International Conference on Principles of Security and Trust*, POST '16, 2016. 9

[30] Roberto Capizzi, Antonio Longo, V. N. Venkatakrishnan, and A. Prasad Sistla. Preventing information leaks through shadow executions. In *Proceedings of the 2008 Annual Computer Security Applications Conference*, ACSAC '08, 2008. 68

[31] Ethan Cecchetti, Andrew Myers, and Owen Arden. Nonmalleable information flow control. In *Proceedings of the 2017 ACM Conference on Computer and Communications Security*, CCS '17, 2017. 2, 11, 12, 32, 34, 51, 53

[32] Stephen Chong and Andrew C. Myers. Decentralized robustness. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop*, CSFW '06, 2006. 11

[33] Andrey Chudnov and David A. Naumann. Inlined information flow monitoring for JavaScript. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '15, 2015. 6

[34] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for JavaScript. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, PLDI '09, 2009. 6

[35] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. FlowFox: a web browser with flexible and precise information flow control. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, 2012. 1, 6, 16, 68, 69, 94

[36] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5), 1976. 4

[37] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7), 1977. 1

[38] Dominique Devriese and Frank Piessens. Noninterference through secure multi-execution. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, 2010. 1, 2, 6, 69, 72

[39] Mohan Dhawan and Vinod Ganapathy. Analyzing information flow in JavaScript-based browser extensions. In *Proceedings of the 2009 Computer Security Applications Conference*, ACSAC '09, 2009. 6

[40] Tim Disney and Cormac Flanagan. Gradual information flow typing. In *Proceedings of the 2nd International Workshop on Scripts to Programs Evolution*, STOP '11, 2011. 1

[41] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '10, 2010. 1, 69, 71, 95

[42] José Fragoso Santos, Thomas Jensen, Tamara Rezk, and Alan Schmitt. Hybrid typing of secure information flow in a JavaScript-like language. In *Proceedings of the 10th International Symposium on Trustworthy Global Computing*, TGC '15, pages 63–78, 2016. 1

[43] Joseph A. Goguen and José Meseguer. Security policies and security models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, 1982. 1, 4

[44] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. Secure multi-execution of web scripts: Theory and practice. *Journal of Computer Security*, 22(4), 2014. 75

[45] Daniel Hedin, Luciano Bello, and Andrei Sabelfeld. Value-sensitive hybrid information flow control for a JavaScript-like language. In *Proceedings of the 2015 IEEE Computer Security Foundations Symposium*, CSF '15, 2015. 1

[46] Daniel Hedin, Luciano Bello, and Andrei Sabelfeld. Information-flow security for JavaScript and its APIs. *Journal of Computer Security (JCS)*, 24(2), 2016. 6

[47] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. JSFlow: Tracking information flow in JavaScript and its APIs. In *Proceedings of the ACM Symposium on Applied Computing*, SAC '14, 2014. 6

[48] Daniel Hedin and Andrei Sabelfeld. Information-flow security for a core of JavaScript. In *Proceedings of the 2012 IEEE Computer Security Foundations Symposium*, CSF '12, 2012. 6

[49] Daniel Hedin and Andrei Sabelfeld. A perspective on information-flow control. In *Software Safety and Security*. IOS Press, 2012. 5

[50] Nevin Heintze and Jon G Riecke. The SLam calculus: programming with secrecy and integrity. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, POPL '98, 1998. 1

[51] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. An empirical study of privacy-violating information flows in JavaScript web applications. In *Proceedings of the 2010 ACM Conference on Computer and Communications Security*, CCS '10, 2010. 1, 6

[52] Limin Jia, Jassim Aljuraidan, Elli Fragkaki, Lujo Bauer, Michael Stroucken, Kazuhide Fukushima, Shinsaku Kiyomoto, and Yutaka Miyake. Run-time enforcement of information-flow properties on android (extended abstract). In *Proceedings of the European Symposium on Research in Computer Security*, ESORICS '13, 2013. 1, 3, 69, 95

[53] Seth Just, Alan Cleary, Brandon Shirley, and Christian Hammer. Information flow analysis for JavaScript. In *Proceedings of the ACM Workshop on Programming Language and Systems Technologies for Internet Clients*, PLASTIC '11, 2011. 6

[54] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, 2007. 95

[55] Gurvan Le Guernic. Automaton-based confidentiality monitoring of concurrent programs. In *Proceedings of the 2007 IEEE Computer Security Foundations Symposium*, CSF '07, 2007. 95

[56] Gurvan Le Guernic, Anindya Banerjee, Thomas Jensen, and David Schmidt. Automata-based confidentiality monitoring. In *Proceedings of the 11th Annual Asian Computing Science Conference*, ASIAN '06, 2006. 1

[57] Zhou Li, Kehuan Zhang, and XiaoFeng Wang. Mash-IF: Practical information-flow control within client-side mashups. In *2010 IEEE/IFIP International Conference on Dependable Systems & Networks*, DSN '10, 2010. 8

[58] Heiko Mantel. On the composition of secure systems. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, SP '02, 2002. 5

[59] McKenna McCall, Abhishek Bichhawat, and Limin Jia. Compositional information flow monitoring for reactive programs. In *Proceedings of the 2022 IEEE 7th European Symposium on Security and Privacy*, EuroSP '22, 2022. 69

[60] McKenna McCall, Hengruo Zhang, and Limin Jia. Knowledge-based security of dynamic secrets for reactive programs. In *Proceedings of the 2018 IEEE Computer Security Foundations Symposium*, CSF '18, 2018. 13

[61] Daryl McCullough. Specifications for multi-level security and a hook-up. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, SP '87, 1987. 5

[62] Daryl McCullough. Noninterference and the composability of security properties. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, SP '88, 1988. 5

[63] Daryl McCullough. A hookup theorem for multilevel security. *IEEE Transactions on Software Engineering*, 16(6), 1990. 5

[64] Scott Moore, Aslan Askarov, and Stephen Chong. Precise enforcement of progress-insensitive security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, 2012. 5

[65] Scott Moore and Stephen Chong. Static analysis for efficient hybrid information-flow control. In *Proceedings of the 2011 IEEE Computer Security Foundations Symposium*, CSF '11, 2011. 1

[66] Andrew C Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification. In *Proceedings of Computer Security Foundations Workshop*, CSFW '04, 2004. 2, 11, 13, 32

[67] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security (JCS)*, 14(2), 2006. 68

[68] Adwait Nadkarni, Benjamin Andow, William Enck, and Somesh Jha. Practical DIFC enforcement on android. In *25th USENIX Security Symposium*, USENIX '16, 2016. 3, 95

[69] Minh Ngo, Nataliia Bielova, Cormac Flanagan, Tamara Rezk, Alejandro Russo, and Thomas Schmitz. A better facet of dynamic information flow control. In *Proceedings of the 2018 World Wide Web Conference*, WWW '18, 2018. 7

[70] Minh Ngo, Frank Piessens, and Tamara Rezk. Impossibility of precise and sound termination-sensitive security enforcements. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy*, SP '18, 2018. 7

[71] Willard Rafnsson and Andrei Sabelfeld. Compositional information-flow security for interactive systems. In *Proceedings of the 2014 IEEE Computer Security Foundations Symposium*, CSF '14, 2014. 5

[72] Willard Rafnsson and Andrei Sabelfeld. Secure multi-execution: Fine-grained, declassification-aware, and transparent. *Journal of Computer Security (JCS)*, 24(1), 2016. 8

[73] Vineet Rajani, Abhishek Bichhawat, Deepak Garg, and Christian Hammer. Information flow control for event handling and the DOM in web browsers. In *Proceedings of the 2015 IEEE Computer Security Foundations Symposium*, CSF '15, 2015. 1, 93, 95

[74] Bob Reynders, Dominique Devriese, and Frank Piessens. Multi-tier functional reactive programming for the web. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! '14, 2014. 4

[75] Alejandro Russo and Andrei Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proceedings of the 2010 IEEE Computer Security Foundations Symposium*, CSF '10, 2010. 1

[76] Alejandro Russo, Andrei Sabelfeld, and Andrey Chudnov. Tracking information flow in dynamic tree structures. In *Proceedings of the European Symposium on Research in Computer Security*, ESORICS '09, 2009. 80, 81, 93

[77] Andrei Sabelfeld. The impact of synchronisation on secure information flow in concurrent programs. In *Proceedings of the Perspectives of Systems Informatics: 4th International Andrei Ershov Memorial Conference*, PSI '01, 2001. 95

[78] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003. 1, 5

[79] Andrei Sabelfeld and Alejandro Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Proceedings of the Perspectives of Systems Informatics: 7th International Andrei Ershov Memorial Conference*, PSI "09, 2009. 1

[80] Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings of Computer Security Foundations Workshop*, CSFW '00, 2000. 95

[81] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security (JCS)*, 17(5), 2009. 8

[82] José Fragoso Santos and Tamara Rezk. An information flow monitor-inlining compiler for securing a core of JavaScript. In *Proceedings of the 29th International Information Security and Privacy Conference*, SEC '14, 2014. 1

[83] Thomas Schmitz, Maximilian Algehed, Cormac Flanagan, and Alejandro Russo. Faceted secure multi execution. In *Proceedings of the 2018 ACM Conference on Computer and Communications Security*, CCS '18, 2018. 7

[84] Daniel Schoepe, Musard Balliu, Benjamin C. Pierce, and Andrei Sabelfeld. Explicit secrecy: A policy for taint tracking. In *Proceedings of the 1st IEEE European Symposium on Security and Privacy*, EuroSP '16, 2016. 2, 5, 8, 9, 11, 69, 70, 72, 89, 95

[85] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, POPL '98, 1998. 95

[86] Steven Sprecher, Christoph Kerschbaumer, and Engin Kirda. SoK: All or nothing - a postmortem of solutions to the third-party script inclusion permission model and a path forward. In *Proceedings of the 2022 IEEE 7th European Symposium on Security and Privacy*, EuroSP '22, 2022. 1, 93

[87] Deian Stefan, Alejandro Russo, Pablo Buiras, Amit Levy, John C Mitchell, and David Mazieres. Addressing covert termination and timing channels in concurrent information flow systems. In *Proceedings of the 17th ACM International Conference on Functional Programming*, ICFP '12, 2012. 95

[88] Deian Stefan, Edward Z. Yang, Brad Karp, Petr Marchenko, Alejandro Russo, and David Mazières. Protecting users by confining JavaScript with COWL. In *Proceedings of the USENIX conference on Operating Systems Design and Implementation*, OSDI '14, 2014. 1, 6, 8, 69

[89] Ta-chung Tsai, Alejandro Russo, and John Hughes. A library for secure multi-threaded information flow in Haskell. In *Proceedings of the 2007 IEEE Computer Security Foundations Symposium*, CSF '07, 2007. 95

[90] Steven Van Acker and Andrei Sabelfeld. Javascript sandboxing: Isolating and restricting client-side JavaScript. FOSAD '16, 2016. 93

[91] Mathy Vanhoef, Willem De Groef, Dominique Devriese, Frank Piessens, and Tamara Rezk. Stateful declassification policies for event-driven programs. In *Proceedings of the 2014 IEEE Computer Security Foundations Symposium*, CSF '14, 2014. 2, 6, 8, 9, 13, 14, 22, 29, 45, 68

[92] Jeffrey A Vaughan and Stephen Chong. Inference of expressive declassification policies. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, 2011. 4

[93] Dennis Volpano and Geoffrey Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(2-3), 1999. 95

[94] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3), 1996. 1

[95] Dennis M. Volpano. Safety versus secrecy. In *Proceedings of the 6th International Symposium on Static Analysis*, SAS '99, 1999. 2, 5, 8, 11, 69, 70, 72, 89, 95

[96] MDN web docs. Event.istrusted, 2023. [Online; accessed 9-January-2023]. 41

[97] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. Precise, dynamic information flow for database-backed applications. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, PLDI '16, 2016. 7

[98] A. Zakinthinos and E. S. Lee. The composability of non-interference [system security]. In *Proceedings of Computer Security Foundations Workshop*, CSFW '95, 1995. 5

[99] Dante Zanarini, Mauro Jaskelioff, and Alejandro Russo. Precise enforcement of confidentiality for reactive systems. In *Proceedings of the 2013 IEEE Computer Security Foundations Symposium*, CSF '13, 2013. 7

[100] Steve Zdancewic. Challenges for information-flow security. In *Proceedings of the 1st International Workshop on the Programming Language Interference and Dependence*, PLID '04, 2004. 93

[101] Steve Zdancewic and Andrew C Myers. Robust declassification. In *Proceedings of Computer Security Foundations Workshop*, CSFW '01, 2001. 2, 11, 15, 32, 34, 47

[102] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proceedings of the USENIX conference on Operating Systems Design and Implementation*, OSDI '06, 2006. 95