Knowledge-based Security of Dynamic Secrets for Reactive Programs

McKenna McCall, Hengruo Zhang, and Limin Jia
Carnegie Mellon University
Pittsburgh, USA
{mckennak, hengruoz, liminjia}@andrew.cmu.edu

***Abstract*—Scripts on webpages could steal sensitive user data. Much work has been done, both in modeling and implementation, to enforce information flow control (IFC) of webpages to mitigate such attacks. It is common to model scripts running in an IFC mechanism as a reactive program. However, this model does not account for dynamic script behavior such as user action simulation, new DOM element generation, or new event handler registration, which could leak information. In this paper, we investigate how to secure sensitive user information, while maintaining the flexibility of declassification, even in the presence of *active attackers*—those who can perform the aforementioned actions. Our approach extends prior work on secure-multi-execution with stateful declassification by treating script-generated content specially to ensure that declassification policies cannot be manipulated by them. We use a knowledge-based progress-insensitive definition of security and prove that our enforcement mechanism is sound. We further prove that our enforcement mechanism is precise and has robust declassification (i.e. active attackers cannot learn more than their passive counterpart).**

I. Introduction

Users are becoming increasingly accustomed to web services such as banking, social media, email, and shopping. Accessing these services often requires sensitive personal information, such as email addresses, phone numbers, passwords, credit card numbers, or even social security numbers. As a result, it would be profitable for web applications or third-party scripts to access this information. Indeed, web attackers are known to steal sensitive user data [23].

There has been much work on the development of information flow control (IFC) mechanisms in the browser context to mitigate such attacks. In the theoretical domain, reactive and interactive models for client-side scripts [13], [18], detailed models for the DOM [1], [28], and new definitions for security properties that suit such programming models have been proposed [12], [18], [31]. On the systems side, several projects have modified existing browsers, browser components, or implemented extensions to enforce IFC [1], [7]–[9], [17], [28], [30].

One of the challenges of IFC is dealing with *declassification*. How can sensitive information be intentionally released while maintaining a provably secure system. Allowing principled declassification is particularly important in the browser context, 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 [7], [9], or used a simplified model that is missing some dynamic Javascript features that could leak information [31].

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* [26], in which an active attacker can abuse declassification components and trick the system into leaking more information than intended. Another 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* [3], 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 paper 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 [31] 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.

To the best of our knowledge, our paper is the first to study the interaction between these dynamic script features and declassification. Our results are one more step toward enforcing IFC in real browsers.

The rest of this paper is organized as follows. We briefly review systems and concepts that our work builds on in Section II, and present examples where dynamic features interfere with declassification in Section III. In Section IV, we introduce our dynamic reactive program model and introduce declassification. Our SME system and its formal properties are presented in Section V. We discuss specific aspects of our system in Section VI and related work in Section VII.

Detailed definitions, lemmas, and proofs can be found in our companion technical report [25].

II. Background

In this section, we briefly review the reactive program model, secure multi-execution, and stateful declassification to set up the background for our work.

Reactive programs have been used to model event-driven programs, such as scripts on webpages [13]. In the reactive model, a program is a set of event handlers. The top-level event loop is single-threaded and each event handler only executes when a corresponding event is triggered. In this model, only one event handler executes at a time and events waiting to be processed stay in an event queue. To manage the single-threaded event loop, the runtime keeps track of system state (*consumer* or *producer* state). In the consumer state, a new event can be processed. Once an event handler executes, the system enters the producer state. The system stays in the producer state until the current event handler finishes, at which point, the system switches back to the consumer state to process the next event. This model is a nice and clean 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 [10], [29].

Secure multi-execution (SME) was introduced as an information flow control (IFC) mechanism for JavaScript on web pages [17], [18]. A copy of the script runs at each security level. Consider a two point security lattice with labels L and H and partial order $L \sqsubseteq H$ as an example. The copy that runs at security level H receives input from both H and L, outputs to H channels, and its output to L channels is thrown away. On the other hand, the copy that runs at security level L receives only L inputs and outputs to L channels. The H inputs are replaced with default values, and the H outputs are suppressed. This way, potential information leaks from H inputs to L outputs are stopped.

To allow scripts that depend on approximated or aggregated secret values (e.g. analytical scripts) to run correctly in SME, Vanhoef et al. proposed an approach to implementing stateful declassification policies [31]. 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.

III. Dynamic Features Leak Information

We illustrate potential security problems caused by interactions between dynamic features of scripts and declassification and demonstrate how knowledge-based noninterference is used in our setting.

A. Scripts Interfering with Declassification

One of the drawbacks of the reactive programming model from the prior work discussed in Section II 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 do not model event bubbling, preemptive events, or DOM element removal, but plan to extend our model to address these in 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 shouldn't be affected by script operations. Allowing scripts to control what is declassified violates the principle of *robust declassification* [33], 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.

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

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 [17] (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 need 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.

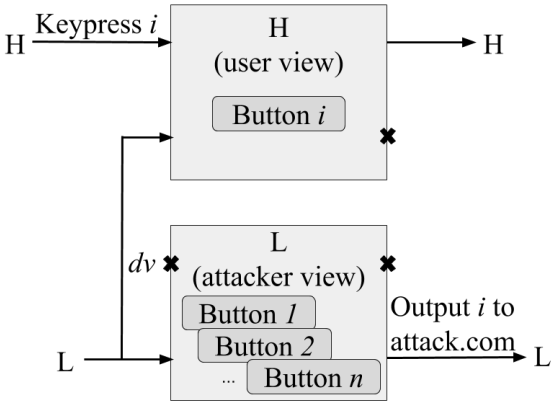

Figure 1. The high execution receives the real keypress, so generates only one button with id id_i. The low execution receives the default value, so generates all n buttons.

To remove this restriction, we give each execution its own copy of the DOM. However, if we freely allow the high execution to add new 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 branch.

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

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 1. The user never sees the buttons from the low execution and the attacker doesn't see which button was generated for the user, but when the 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 V, we show how to stop leaks through dynamically generated elements. Next, we show informally that this example violates a knowledge-based security property.

B. Knowledge-Based Security

We review knowledge-based noninterference and gradual release, and provide some intuition for how gradual release is useful to our dynamic program model.

For explanatory purposes, we write t to denote an execution trace, τ to denote input/output sequences, and $\mathcal{L}$ to denote a label context that maps events to security labels.

The secrets in our system are sequences of user inputs. Let us write $\tau \approx_L^{\mathcal{L}} \tau'$ to denote that two traces are observationally equivalent at the level L given the label context $\mathcal{L}$. The $\approx_L^{\mathcal{L}}$ relation is standard: $\tau \approx_L^{\mathcal{L}} \tau'$ if removing all secret events (those which are not observable from L) from τ and τ' results in the same trace. $\mathcal{L}$ is formally defined in Section IV-C.

An attacker's knowledge, written $\mathcal{K}(\tau, \sigma_0, \mathcal{L})$, is the set of possible input sequences that could produce an output trace that is observationally equivalent at L to τ from the initial configuration σ_0 given the context $\mathcal{L}$.

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.

$$\mathcal{K}(\tau, \sigma_0, \mathcal{L}) = \{\tau_i \mid \exists t \in \mathsf{runs}(\sigma_0), \mathsf{out}(t) \approx_L^{\mathcal{L}} \tau \\ \wedge \tau_i = \mathsf{in}(t)\}$$

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. $\tau \cdot \alpha \in \mathsf{runs}(\sigma_0)$, $\mathcal{K}(\tau, \sigma_0, \mathcal{L}) \subseteq_\preceq \mathcal{K}(\tau \cdot \alpha, \sigma_0, \mathcal{L})$.*

Here, $S_1 \subseteq_\preceq S_2$ means that every element in S_1 is a prefix of an element in S_2. This is a gradual release property [3]. It is weaker than the standard noninterference property, 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 have to be associated with existing elements, which reduces the number of possible inputs.

Let's revisit the example in Section III-A. Let's assume the attacker knows the program and that the secret value is between 1 and 8. The attacker now knows that $id.ev(9)$ is not a possible input. We allow the attacker to know this type of information, even though it refines their knowledge. After the first input and before seeing $id_2.\mathsf{click}(v)$, the knowledge of the argument of the first input event could be any integer begin 1 and 8:

$$\mathcal{K}([id.ev(2)], \sigma_0, \mathcal{L}) = \{[id.ev(1)], \cdots, [id.ev(8)]\}$$

After observing $id_2.\mathsf{click}(v)$, every possible input except 2 is eliminated.

$$\mathcal{K}([id.ev(2), id_2.onClick(...)], \sigma_0, \mathcal{L}) \\ = \{[id.ev(2), id_2.onClick(...)]\}$$

Here, not all knowledge of the shorter trace is a prefix of the knowledge of the longer trace:

$$\{[id.ev(1)], \cdots, [id.ev(8)]\} \\ \not\subseteq_\preceq \{[id.ev(2), id_2.onClick(...)]\}$$

The program is not secure using our definition.

We will present the formal definitions in Section V.

IV. 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.

A. 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 [13] 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.

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

Command c includes the following actions: $\mathsf{output}\ ch\ e$ evaluates e and sends the result to URL ch, $\mathsf{trigger}\ id.ev(e)$ 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 also allow multiple event handlers to be registered for one 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, denoted 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 a number of 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.

B. Operational Semantics

To define the operational semantics for our language, we first introduce a few runtime constructs. We write E to denote the set of events generated by the event handlers. As we discussed in Section III, 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, 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).

events	E	::=	$\cdot \mid E, id.ev(v)$
non-silent actions	a	::=	$id.ev(v) \mid ch(v)$
actions	α	::=	$a \mid \bullet$
execution state	s	::=	$P \mid C \mid LC$
configurations	κ	::=	σ, c, s, E
action traces	τ	::=	$\cdot \mid \tau\, \alpha$
execution traces	t	::=	$\kappa \mid \kappa \xrightarrow{\alpha} t$

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 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 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.

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

$$\frac{}{\sigma, \mathsf{skip}; c \xrightarrow{\bullet} \sigma, c, \cdot}\ \textsc{skip}$$

$$\frac{\sigma, c_1 \xrightarrow{\alpha} \sigma', c_1', E}{\sigma, c_1; c_2 \xrightarrow{\alpha} \sigma', c_1'; c_2, E}\ \textsc{seq}$$

$$\frac{[\![e]\!]_\sigma = v}{\sigma, x := e \xrightarrow{\bullet} \sigma[x \mapsto v], \mathsf{skip}, \cdot}\ \textsc{assign}$$

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

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

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

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

$$\frac{[\![e]\!]_\sigma = v}{\sigma, \mathsf{output}\ ch\ e \xrightarrow{ch(v)} \sigma, \mathsf{skip}, \cdot}\ \textsc{output}$$

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

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

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

Figure 2. Operational Semantics of Commands

We summarize the operational semantic rules for event loops in Figure 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

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

$$\frac{}{\sigma, \mathsf{skip}, P, \cdot \xrightarrow{\bullet} \sigma, \mathsf{skip}, C, \cdot}\ \textsc{PtoC}$$

$$\frac{E \neq \cdot}{\sigma, \mathsf{skip}, P, E \xrightarrow{\bullet} \sigma, \mathsf{skip}, LC, E}\ \textsc{PtoLC}$$

$$\frac{\sigma(id.ev(v)) = c}{\sigma, \mathsf{skip}, C, \cdot \xrightarrow{id.ev(v)} \sigma, c, P, \cdot}\ \textsc{CtoP-Usr-Input}$$

$$\textsc{CtoP-Script-Input}\ \frac{\sigma(id.ev(v)) = c}{\sigma, \mathsf{skip}, LC, (id.ev(v), E) \xrightarrow{\bullet} \sigma, c, P, E}$$

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

Figure 3. Operational Semantics for Event Loop

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.

C. Security Labels

Before introducing declassification policies, we define our security lattice. Figure 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 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 affect on low outputs, it is possible that we will change the behavior of otherwise benign programs. This affects how we reason about the *precision* of our enforcement mechanism, which says that the semantics of good programs should not be altered. See Section V-C for information about our precision theorem and Section VI for further discussion.

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

Figure 4. Constructs for Defining Security Policies

We use a label context $\mathcal{L}$ 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 $\mathcal{L}$. The judgment $\mathcal{L} \vdash a : \ell$ means that a non-silent action a has security label ℓ with regard to the label context $\mathcal{L}$. It is defined as follows:

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

$$\frac{m_l(ch, v) = \ell}{(\Gamma, m_l) \vdash ch(v) : \ell}$$

To decide the label of an event $id.ev(v)$, 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: $m_l(ev, v)$. Instead of using the judgment, we write $\mathcal{L}(a)$ to denote the security label of a given $\mathcal{L}$. For instance, a label context $\mathcal{L}$ 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 $\mathcal{L}$ in the example in Section III-A, we have $\mathcal{L}(button_0.\mathsf{click}(v)) = H$ and $\mathcal{L}(id_1.\mathsf{click}(v)) = H_\Delta$.

D. 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 [31], 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 [6], 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 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.

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

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

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, except the CToP-User-Input rule. The resulting set of rules may be found in Section V, Figure 6 (they will be re-used for defining SME rules in that section).

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

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

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

$$\frac{\begin{array}{c}\sigma(id.ev(v)) = c \\ \mathcal{L}(id.ev(v)) = H \qquad \mathcal{R} = (\rho, \mathcal{D}) \\ \mathcal{D}(id.ev(v)) = (r, \_, \rho') \qquad d' = \mathsf{update}(d, r)\end{array}}{\mathcal{L} \vdash \mathcal{R}, d, \sigma, \mathsf{skip}, C, \cdot \xrightarrow{id.ev(v)} (\rho', \mathcal{D}), d', \sigma, c, P, \cdot} \text{ IN-H}$$

$$\frac{d, \kappa \xrightarrow{\alpha} \kappa'}{\mathcal{L} \vdash \mathcal{R}, d, \kappa \xrightarrow{\alpha} \mathcal{R}, d, \kappa'} \text{ OUT}$$

Our release function is only applied to events that are labeled H. Therefore, the runtime state includes the label context $\mathcal{L}$. 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.

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 V-C, 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.

V. Secure Multi-Execution

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.

A. 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.

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

We write $\mathcal{L} \vdash \Sigma \overset{\alpha}{\Longrightarrow} \Sigma'$ 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 7 and 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. 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 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 5.

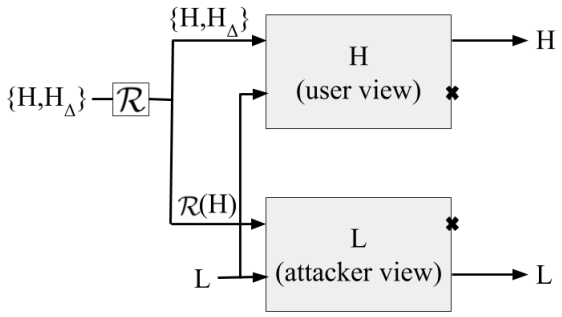

Figure 5. The high execution receives all inputs, unchanged. The low execution receives L inputs and released H inputs through the release module. Note that the release module throws out the H_Δ inputs so that they don't interfere with the declassification policy.

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 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.

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 III-A, 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

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

$$\frac{E \neq \cdot}{d, \sigma, \mathsf{skip}, P, E \xrightarrow{\bullet} \sigma, \mathsf{skip}, LC, E} \textsc{PtoLC}$$

$$\frac{}{d, \sigma, \mathsf{skip}, P, \cdot \xrightarrow{\bullet} \sigma, \mathsf{skip}, C, \cdot} \textsc{PtoC}$$

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

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

Figure 6. Operational Semantic Rules for Single Execution

$\boxed{\mathcal{L} \vdash \Sigma \xRightarrow{\alpha} \Sigma'}$

$$\frac{\begin{array}{c}\mathcal{L}(id.ev(v)) = H \\ \mathcal{D}(\rho, id.ev(v)) = (r, \mathsf{emp}, \rho') \\ d' = \mathsf{update}(d, r) \qquad \sigma_H(id.ev(v)) = c_H\end{array}}{\begin{array}{l}(\rho, \mathcal{D}), d; \sigma_L, \mathsf{skip}, C, \cdot; \sigma_H, \mathsf{skip}, C, \cdot \\ \mathcal{L} \vdash \xRightarrow{id.ev(v)} (\rho', \mathcal{D}), d'; \sigma_L, \mathsf{skip}, C, \cdot; \\ \qquad\qquad \sigma_H, c_H, P, \cdot\end{array}} \textsc{SmeI-NR1}$$

$$\frac{\mathcal{L}(id.ev(v)) = H_\Delta \qquad \sigma_H(id.ev(v)) = c_H}{\begin{array}{l}\mathcal{R}, d; \sigma_L, \mathsf{skip}, C, \cdot; \sigma_H, \mathsf{skip}, C, \cdot \\ \mathcal{L} \vdash \xRightarrow{id.ev(v)} \mathcal{R}, d; \sigma_L, \mathsf{skip}, C, \cdot; \\ \qquad\qquad \sigma_H, c_H, P, \cdot\end{array}} \textsc{SmeI-NR2}$$

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

$$\frac{\begin{array}{c}\mathcal{L}(id.ev(v)) = L \\ \sigma_L(id.ev(v)) = c_L \qquad \sigma_H(id.ev(v)) = c_H\end{array}}{\begin{array}{l}\mathcal{R}, d; \sigma_L, \mathsf{skip}, C, \cdot; \sigma_H, \mathsf{skip}, C, \cdot \\ \mathcal{L} \vdash \xRightarrow{id.ev(v)} \mathcal{R}, d; \sigma_L, c_L, P, \cdot; \\ \qquad\qquad \sigma_H, c_H, P, \cdot\end{array}} \textsc{SmeI-L}$$

Figure 7. SME Input Rules

on the "agree to share GPS location" button will 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.

B. Soundness

In Section III-B, 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 $\mathsf{iruns}(\sigma_0, \mathcal{L}, \mathcal{R})$ 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 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, $\mathcal{K}(T, \sigma_0, \mathcal{L}, \mathcal{R})$ is the set of possible input traces that could produce an execution trace that is observationally equivalent to T.

$$\mathcal{K}(T, \sigma_0, \mathcal{L}, \mathcal{R}) = \{\tau_i \mid \exists T' \in \mathsf{iruns}(\sigma_0, \mathcal{L}, \mathcal{R}), T \approx_L^{\mathcal{L}} T' \wedge \tau_i = \mathsf{in}(T')\}$$

$\boxed{\mathcal{L} \vdash \Sigma \xRightarrow{\alpha} \Sigma'}$

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

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

$$\frac{\neg\mathsf{consumer}(\kappa_H) \qquad \mathsf{consumer}(\kappa_L) \qquad d, \kappa_H \xrightarrow{\alpha} \kappa_H' \qquad \mathcal{L}(\alpha) = H}{\mathcal{L} \vdash \mathcal{R}, d; \kappa_L; \kappa_H \xRightarrow{\alpha} \mathcal{R}, d; \kappa_L; \kappa_H'} \textsc{SmeO-HH}$$

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

Figure 8. SME Output Rules

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 $\mathcal{L}$, denoted $T \Downarrow_L^{\mathcal{L}}$, 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

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

$$\frac{}{(\cdot) \Downarrow_L^{\mathcal{L}} = \cdot}$$

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

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

$$\frac{T' \in \mathsf{runs}(\Sigma', \mathcal{R}', \mathcal{L}) \qquad \Sigma \approx_L \Sigma'}{(\mathcal{L} \vdash \Sigma \xRightarrow{\alpha} T') \Downarrow_L^{\mathcal{L}} = T' \Downarrow_L^{\mathcal{L}}}$$

Figure 9. Projection of Traces

inputs, the low inputs, and the low outputs. Formally, $T \Downarrow_L^{\mathcal{L}}$ is defined in Figure 9. Here :: denotes concatenation, and $\mathsf{runs}(\Sigma, \mathcal{R}, \mathcal{L})$ is the set of execution traces beginning from Σ with release policy $\mathcal{R}$ and label context $\mathcal{L}$.

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

Definition 2 is progress sensitive. For instance, if a confidential value determines whether or not 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 2. To prove this statement for our rules, we want to show that all of the shorter traces in $\mathcal{K}(T, \sigma_0, \mathcal{L}, \mathcal{R})$ are prefixes of longer traces in $\mathcal{K}(T \overset{\alpha}{\Longrightarrow} \Sigma, \sigma_0, \mathcal{L}, \mathcal{R})$. 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 $\mathcal{K}(T \overset{\alpha}{\Longrightarrow} \Sigma, \sigma_0, \mathcal{L}, \mathcal{R})$.

Instead, we consider a progress-insensitive definition of security. We define a trace which can make progress as follows:

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

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

$$\begin{aligned} &\mathcal{K}_t(T, \sigma_0, \mathcal{L}, \mathcal{R}) = \\ &\quad \{\tau_i \mid \exists T' \in \mathsf{iruns}(\sigma_0, \mathcal{L}, \mathcal{R}), T \approx_L^{\mathcal{L}} T' \wedge \\ &\qquad \tau_i = \mathsf{in}(T') \wedge \mathsf{prog}(T', \mathcal{L})\} \end{aligned}$$

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

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

We prove that our SME rules are sound, formally:

Theorem 4 (Soundness)**.** *$\forall \mathcal{L}, \mathcal{R}, \sigma_0$, s.t. σ_0 is secure w.r.t. the label context $\mathcal{L}$ and release policy $\mathcal{R}$ against an attacker at level L.*

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

Lemma 5 (Strong One-step)**.** *If $T_1 = \mathcal{L} \vdash \Sigma_1 \overset{\alpha}{\Longrightarrow} \Sigma_1'$ with $\Sigma_1 \not\approx_L \Sigma_1'$, $\Sigma_1 \approx_L \Sigma_2$, and $\mathsf{prog}(\Sigma_2, \mathcal{L})$ then $\exists \Sigma_2', \tau$ s.t. $T_2 = \mathcal{L} \vdash \Sigma_2 \Longrightarrow^* \Sigma_2'$ with $T_1 \approx_L^{\mathcal{L}} T_2$ 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 6 (Eq trace Eq state)**.** *If $T_1 = \mathcal{L} \vdash \Sigma_1 \Longrightarrow^* \Sigma_1'$ and $T_2 = \mathcal{L} \vdash \Sigma_2 \Longrightarrow^* \Sigma_2'$ with $\Sigma_1 \approx_L \Sigma_2$ and $T_1 \approx_L^{\mathcal{L}} T_2$, then $\Sigma_1' \approx_L \Sigma_2'$.*

Lemma 5 is proven by examining each case of $\mathcal{E} :: \mathcal{L} \vdash \Sigma_1 \overset{\alpha}{\Longrightarrow} \Sigma_1'$, where α is observable, and showing that the second trace can take an equivalent step. Lemma 6 is proven by induction over the length of the trace T_1. A detailed proof of Theorem 4, as well as supporting lemmas may be found in our companion technical report [25].

C. 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 [31].

Definition 7 (Compatibility)**.** *We say that a state σ is compatible with a release policy $\mathcal{R}$ and label context $\mathcal{L}$, when for all τ $\mathcal{L} \vdash d_0, \mathcal{R}; \kappa \overset{\tau}{\longrightarrow}^* d', \mathcal{R}'; \kappa'$ iff $\kappa \overset{\tau}{\longrightarrow}^* \kappa'$ where d_0 is the initial release channel, $\kappa = (\sigma, \mathsf{skip}, C, \cdot)$.*

We use the judgement $\kappa \longrightarrow^* \kappa'$ to denote program execution without SME. Definition 7 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 $\tau|_\ell^{\mathcal{L}}$ to denote the projection of an action sequence to label ℓ under the label context $\mathcal{L}$, and $\mathcal{R}_{\mathcal{L}}^*(\tau)$ to denote repeatedly applying $\mathcal{R}$ to each input event in τ with label context $\mathcal{L}$.

Definition 8 (No leak outside declassification)**.** *We say that a state σ is has no leak outside declassification, if for all label context $\mathcal{L}$, release policies $\mathcal{R}$, $\mathcal{R}'$, $\mathcal{R}_1$, $\mathcal{R}'_1$ and traces τ_{i1}, τ_{i2}, s.t. $\mathcal{R}^*_{\mathcal{L}}(\tau_{i_1}) = \mathcal{R}'^*_{\mathcal{L}}(\tau_{i_2})$, for all τ_1 and τ_2 $\mathcal{L} \vdash t_1 = d_0, \mathcal{R}; \kappa \longrightarrow^* d', \mathcal{R}_1; \kappa'$ and $\mathcal{L} \vdash t_2 = d_0, \mathcal{R}'; \kappa \longrightarrow^* d', \mathcal{R}'_1; \kappa'$, and $\mathsf{in}(t_1) = \tau_{i1}$, $\mathsf{in}(t_2) = \tau_{i2}$, it is the case that $\mathsf{out}(t_1)|^{\mathcal{L}}_L = \mathsf{out}(t_2)|^{\mathcal{L}}_L$.*

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

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

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

Theorem 9 (Precision)**.** *For all $\mathcal{L}$, $\mathcal{R}$, σ and κ_1, $\kappa_1 = (\sigma, \mathsf{skip}, C, \cdot)$, σ is compatible with $\mathcal{L}$ and $\mathcal{R}$, and does not leak outside declassification, then for all complete runs T and t s.t. $\mathcal{L} \vdash T = \Sigma_1 \Longrightarrow^* \Sigma_2$, $t = \kappa_1 \longrightarrow^* \kappa_2$, and $\Sigma_1 = d, \mathcal{R}; \kappa_1; \kappa_1$, and $\mathsf{in}(T) = \mathsf{in}(t)$ imply $\mathsf{out}(T)|^{\mathcal{L}}_H = \mathsf{out}(t)|^{\mathcal{L}}_H$ and $\mathsf{out}(T)|^{\mathcal{L}}_L = \mathsf{out}(t)|^{\mathcal{L}}_L$.*

Proof details may be found in the full version of our paper [25], and uses similar techniques as prior work [31]. 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 VI, we show that programs are not precise using a stronger definition due to dynamic features.

D. 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. The formal definition may be found in our companion technical report [25].

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:

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

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

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

$$\frac{\alpha = ch(v)}{\mathsf{dom}(\alpha) = \{\ \}}$$

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 10 (Robust Declassification)**.** *$\forall \sigma_1, \sigma_2, \mathcal{L}, \mathcal{R}$ s.t. $\sigma_1 <_A \sigma_2$, and $\forall T_1 \in \mathsf{iruns}(\sigma_1, \mathcal{L}, \mathcal{R})$ s.t. T_1 is a complete run, $\forall T_2 \in \mathsf{iruns}(\sigma_2, \mathcal{L}, \mathcal{R})$ s.t. T_2 is a complete run, with $\tau_i = \mathsf{in}(T_1)$, $\mathsf{in}(T_2) = \tau_i \bowtie \tau_\Delta$, $\mathsf{dom}(\tau_\Delta) \cap \Gamma = \emptyset$, $\mathcal{K}(T_1, \sigma_1, \mathcal{L}, \mathcal{R}) \subseteq_{\ll} \mathcal{K}(T_2, \sigma_2, \mathcal{L}, \mathcal{R})$ 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. Details about this proof as well as supporting lemmas may be found in our companion technical report [25].

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 or not it makes progress.

Going back to our example in Section III-A, 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.

VI. 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 taken into account. 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 [15], [26], [33]. We face two challenges when incorporating integrity in our system. SME provides a clean and intuitive mechanism for enforcing confidentiality which may be expanded to include integrity, but it is not clear how to do this without sacrificing performance since every additional label requires another execution. Indeed, previous work which implements SME does not consider integrity. Similarly, our knowledge-based security property is a natural way to reason about confidentiality, as it is precisely the attacker's knowledge we wish to restrict but, a knowledge-based interpretation of integrity has not been studied. We intend to consider integrity in future work.

VII. Related Work

Enforcing IFC in JavaScript Much work has been done on information flow control enforcement in JavaScript [4], [16], [19]–[24]. Because of the dynamic nature of JavaScript, all of the above mentioned projects use runtime enforcement mechanisms to enforce information flow control. Austin and Flanagan [4] present a system which simulates different executions for different security levels using *faceted values* to keep track of the high and low versions of data. SME and faceted execution are similar in principle. Their similarities and differences are well studied [11]. Our system allows user access to the elements present in the high execution, which is what scripts in the high execution see. Scripts in the low execution see an alternative set of elements. This is reminiscent of multi-faceted value, except that the entire DOM, not individual elements, is faceted.

Enforcing IFC on web scripts Several projects have developed tools for enforcing information flow control on web scripts by modifying browser components [7]–[9], [17], [30], [31]. Methods used by these projects include taint tracking, compartmentalization, and secure multi-execution, which was introduced by Devriese and Piessens [18]. SME has since been extended to be more precise [32] and to deal with declassification [14], [27], [31]. Our paper builds on Vanhoef et al.'s work on SME with stateful declassification [31]. Two forms of declassification are considered in this paper: event projection (which returns any information relating to the event which is public, or declassified) and information release (which contains aggregate information, made public periodically). We also introduce dynamic script features and prove a robust declassification theorem.

DOM event handling logic is quite complex and can be used to leak information [28]. 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, the interactions between event bubbling order and pre-emptive event scheduling and declassification policies can be very tricky.

Knowledge-Based Information Flow Security Balliu defined abstract knowledge-based security for distributed programs [5] and studied the relationship between knowledge-based definitions and various trace-equivalence-based definitions. Our knowledge-based security definition is based on the concept of *gradual release*, which was introduced by Askarov and Sabelfeld [3]. The gradual release property enforces that knowledge stays constant outside of intentional releases (declassification). Our definition of security is a gradual release property, except that in most cases, gradual release has been applied to knowledge of possible initial configurations, while ours reasons about possible input sequences. 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 [6]. They also present a type system for enforcing knowledge-based security, which is defined as a conditional gradual release property. Askarov and Chong [2] 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. The concept of robust declassification was introduced by Zdancewic et al. to ensure low integrity attackers cannot manipulate declassification operations [33]. Later work develops a type system for enforcing robust declassification and *qualified robustness* [26]. 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.

VIII. Conclusion

In this paper, 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. As future work, we plan to implement our SME rules in a research prototype browser developed on FireFox.

Acknowledgement

This work was supported in part by the National Science Foundation via grant CNS1704542.

References

[1] A. G. Almeida Matos, J. F. Santos, and T. 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)*, 2014.

[2] A. Askarov and S. Chong. Learning is change in knowledge: Knowledge-based security for dynamic policies. In *Proceedings of the 2012 IEEE Computer Security Foundations Symposium (CSF)*, 2012.

[3] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy (SP)*, 2007.

[4] T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *Proceedings of the ACM Principles of Programming Languages (POPL)*, 2012.

[5] M. Balliu. A logic for information flow analysis of distributed programs. In *Proceedings of the Nordic Conference on Secure IT Systems (NordSec)*. Springer, 2013.

[6] A. Banerjee, D. A. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy (SP)*, 2008.

[7] L. Bauer, S. Cai, L. Jia, T. Passaro, M. Stroucken, and Y. Tian. Run-time monitoring and formal analysis of information flows in Chromium. In *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS)*, 2015.

[8] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Information flow control in WebKit's JavaScript bytecode. In *Proceedings of the International Conference on Principles of Security and Trust (POST)*, 2014.

[9] A. Bichhawat, V. Rajani, J. Jain, D. Garg, and C. Hammer. WebPol: Fine-grained information flow policies for web browsers. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2017.

[10] N. Bielova, D. Devriese, F. Massacci, and F. Piessens. Reactive non-interference for a browser model. In *Proceedings of the International Conference on Network and System Security, (NSS)*, 2011.

[11] N. Bielova and T. Rezk. Spot the difference: Secure multi-execution and multiple facets. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2016.

[12] A. Bohannon and B. C. Pierce. Featherweight Firefox: Formalizing the core of a web browser. In *Proceedings of the 2010 USENIX Conference on Web Application Development (WebApps)*, 2010.

[13] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic. Reactive noninterference. In *Proceedings of the 2009 ACM Conference on Computer and Communications Security (CCS)*, 2009.

[14] I. Boloteanu and D. Garg. Asymmetric secure multi-execution with declassification. In *Proceedings of the International Conference on Principles of Security and Trust (POST)*, 2016.

[15] E. Cecchetti, A. Myers, and O. Arden. Nonmalleable information flow control. In *Proceedings of the 2017 ACM Conference on Computer and Communications Security (CCS)*, 2017.

[16] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[17] W. De Groef, D. Devriese, N. Nikiforakis, and F. 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)*, 2012.

[18] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP)*, 2010.

[19] M. Dhawan and V. Ganapathy. Analyzing information flow in JavaScript-based browser extensions. In *Proceedings of the 2009 Computer Security Applications Conference (ACSAC)*, 2009.

[20] D. Hedin, L. Bello, and A. Sabelfeld. Information-flow security for JavaScript and its APIs. *Journal of Computer Security (JCS)*, 24(2), 2016.

[21] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking information flow in JavaScript and its APIs. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, 2014.

[22] D. Hedin and A. Sabelfeld. Information-flow security for a core of JavaScript. In *Proceedings of the 2012 IEEE Computer Security Foundations Symposium (CSF)*, 2012.

[23] D. Jang, R. Jhala, S. Lerner, and H. 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)*, 2010.

[24] S. Just, A. Cleary, B. Shirley, and C. Hammer. Information flow analysis for JavaScript. In *Proceedings of the ACM Workshop on Programming Language and Systems Technologies for Internet Clients (PLASTIC)*, 2011.

[25] M. McCall, H. Zhang, and L. Jia. Knowledge-based security of dynamic secrets for reactive programs. Technical Report CMU-CyLab-18-001, Cylab, Carnegie Mellon University, March 2018.

[26] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. In *Proceedings of Computer Security Foundations Workshop (CSFW)*, 2004.

[27] W. Rafnsson and A. Sabelfeld. Secure multi-execution: Fine-grained, declassification-aware, and transparent. *Journal of Computer Security (JCS)*, 24(1), 2016.

[28] V. Rajani, A. Bichhawat, D. Garg, and C. Hammer. Information flow control for event handling and the DOM in web browsers. In *Proceedings of the 2015 IEEE Computer Security Foundations Symposium (CSF)*, 2015.

[29] B. Reynders, D. Devriese, and F. 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, part of SPLASH*, 2014.

[30] D. Stefan, E. Z. Yang, B. Karp, P. Marchenko, A. Russo, and D. Mazières. Protecting users by confining JavaScript with COWL. In *Proceedings of the USENIX conference on Operating Systems Design and Implementation (OSDI)*, 2014.

[31] M. Vanhoef, W. De Groef, D. Devriese, F. Piessens, and T. Rezk. Stateful declassification policies for event-driven programs. In *Proceedings of the 2014 IEEE Computer Security Foundations Symposium (CSF)*, 2014.

[32] D. Zanarini, M. Jaskelioff, and A. Russo. Precise enforcement of confidentiality for reactive systems. In *Proceedings of the 2013 IEEE Computer Security Foundations Symposium (CSF)*, 2013.

[33] S. Zdancewic and A. C. Myers. Robust declassification. In *Proceedings of Computer Security Foundations Workshop (CSFW)*, 2001.