Compositional System Security in the Presence of Interface-Confined Adversaries

Deepak Garg, Jason Franklin, Dilsun Kaynar, Anupam Datta

February 19, 2010
CMU-CyLab-10-004

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213
Compositional System Security in the Presence of Interface-Confined Adversaries

Abstract—This paper presents a formal framework for compositional reasoning about secure systems. A key insight is to view a trusted system in terms of the interfaces that the various components expose: larger trusted components are built by combining interface calls in known ways; the adversary is confined to the interfaces it has access to, but may combine interface calls without restriction. Compositional reasoning for such systems is based on an extension of rely-guarantee reasoning for system correctness [1, 2] to a setting that involves an adversary whose exact program is not known. At a technical level, the paper presents an expressive concurrent programming language with recursive functions for modeling interfaces and trusted programs, and a logic of programs in which compositional reasoning principles are formalized and proved sound with respect to trace semantics. The methods are applied to representative examples of web-based systems and network protocols.

I. INTRODUCTION

Compositional security is a recognized central scientific challenge for trustworthy computing (see, for example, Bellovin [3], Mitchell [4], Wing [5]). Contemporary systems are built up from smaller components. However, even if each component is secure in isolation, the composed system may not achieve the desired end-to-end security property: an adversary may exploit complex interactions between components to compromise security. Such attacks have shown up in the wild in many different settings, including web browsers and infrastructure [6–10], network protocols and infrastructure [5, 11–14], and application and systems software [15, 16]. While there has been progress on understanding secure composition in specific settings, such as information flow control for non-interference-style properties [17–19] and cryptographic protocols [20–25], a systematic understanding of the general problem of secure composition has not emerged yet.

This paper makes a contribution in this space. We present a formal framework for compositional reasoning about secure systems, incorporating two main insights. First, we posit that a general theory of secure composition should enable one to flexibly model and parametrically reason about different classes of adversaries. This is critical because while specific domains may have a canonical adversary model (e.g., the standard network adversary model for cryptographic protocols), it is unreasonable to expect that a standard adversary model can be developed for all of system security. Indeed, an adversary against web browser mechanisms has capabilities that are very different from a network adversary or an adversary against a trusted computing system. To develop such a theory we view a trusted system in terms of the interfaces that the various components expose: larger trusted components are built by combining interface calls in known ways; the adversary is confined to the interfaces it has access to, but may combine interface calls without restriction. Such interface-confined adversaries are precisely modeled in our framework and provide a generic way to model different classes of adversaries. For example, in virtual machine monitor-based secure systems, we can model an adversarial guest operating system by confining it to the interface exposed by the virtual machine monitor (VMM). Similarly, adversary models for web browsers, such as the gadget adversary [7], can be modeled by confining the adversary to the read and write interfaces for frames guarded by the same origin policy as well as the frame navigation policies.

Second, we develop compositional reasoning principles for such systems by extending ideas from rely-guarantee reasoning [1, 2]. While rely-guarantee reasoning was developed for proving correctness properties of known concurrent programs, we extend it to soundly reason about system security in the presence of interface-confined adversaries. These principles generalize prior work on compositional logics for network protocol analysis [23, 26–28] and secure systems analysis [29] and are also related to a recently proposed type system for modularly checking interfaces of security protocols [24] (see Section II for a detailed comparison).

At a technical level, the paper presents an expressive concurrent programming language with recursive functions for modeling system interfaces and interface-confined adversaries. Specifically, the programming language is based on an untyped, first-order, concurrent version of the lambda-calculus with side-effects (Section III presents more details). Security properties are specified in a logic of programs (described in Section III). Our primary focus is on security properties that can be cast as safety properties [30]. Compositional reasoning principles are codified in the proof system for the logic of programs to support modular reasoning about program specifications (Section IV-A), trusted programs whose programs are known (Section IV-B) and interface-confined adversarial (untrusted) code (Section IV-C). We present the formal semantics for the logic of programs.
and the main technical result of the paper—a proof of the soundness of the proof system with respect to the trace semantics of the logic (Section V). Finally, we describe how the proof rules support rely-guarantee reasoning in the presence of adversaries (Section VI).

While the focus of this paper is on the technical foundations of this theory, we illustrate the methods by applying them to representative examples of web-based systems and network protocols. As a running example, we consider a web mashup system and present a modular proof of its integrity in the presence of a class of interface-confined adversaries. The interface-based view is useful both in modeling the browser interfaces that are composed in known ways by the mashup and in flexibly modeling the adversary. Furthermore, the proof exercises all the compositional reasoning principles developed in this paper except rely-guarantee reasoning. To demonstrate the latter as well as the generality of our methods we present a modular proof of symmetric key Kerberos V5 in the presence of a symbolic adversary. The proof shows that the reasoning principles for secrecy in Protocol Composition Logic [27] are instances of the general rely-guarantee reasoning principles developed in this paper. Concluding remarks and directions for future work appear in Section VII.

II. RELATED WORK

We discuss below closely related work on logic-based and language-based approaches for compositional security. Orthogonal approaches to secure composition of cryptographic protocols include work on identifying syntactic conditions that are sufficient to guarantee safe composition [20, 25], which are not based on rely-guarantee reasoning. Another orthogonal approach to secure composition is taken in the universal composability or reactive simulatability [21, 22] projects. These simulation-based definitions when satisfied can provide strong composition guarantees. However, they are not based on rely-guarantee reasoning and have been so far applied primarily to cryptographic primitives and protocols.

**Compositional Logics of Security:** The framework presented in this paper is inspired by and generalizes prior work on logics of programs for network protocol analysis [23, 26–28] and secure systems analysis [29]. At a conceptual level, a significant new idea is the use of interface-level abstractions to modularly build trusted systems and flexibly model adversaries with different capabilities by confining them to stipulated interfaces. In contrast, prior work lacked the interface abstraction and had a fixed adversary. Also, the actions (side-effects) in the language were fixed in prior work to communication actions, cryptographic operations, and certain operations on shared memory. On the other hand, our programming model and logic are parametric in actions. One advantage of this generality is that the compositional reasoning principles (proof rules) are action-independent and can be applied to a variety of systems, thus getting at the heart of the problem of compositional security (see Section III-A for details of the parametrization).

We expect domain-specific reasoning to be codified using axioms; thus, the set of axioms for reasoning about network protocols that use cryptographic primitives will be different from those for reasoning about trusted computing platforms. The treatment of rely-guarantee reasoning in the presence of adversaries generalizes the invariant rule schemas for authentication [23], integrity [29], and secrecy [27] properties developed earlier.

**Refinement types for verifying protocols:** Recently, Bhargavan et al. have developed a type system to modularly check interfaces of security protocols, implemented it, and applied it to analysis of secrecy properties of cryptographic protocols [24]. Their approach is based on refinement types, i.e., ordinary types qualified with logical assertions, which can be used to specify program invariants and pre- and post-conditions. Programmers annotate various points in the model with assumed and asserted facts. The main safety theorem states that all programmer defined assertions are implied by programmer assumed facts. However, a semantic connection between the program state and the logical formulas representing assumed and asserted facts is missing. Consequently, there are no semantic end-to-end guarantees and a protocol is proved correct only to the extent that annotations in its code are meaningful. In contrast, we prove that the inference system of our logic of programs is sound with respect to trace semantics of the programming language. Therefore, properties verified for a system will hold on any trace obtained from the system’s execution without relying on (possibly incorrect) programmer-provided annotations. Our logic of programs may provide a semantic foundation for the work of Bhargavan et al. and, dually, the implementation in that work may provide a basis for mechanizing the formal system presented in this paper. Bhargavan et al.’s programming model is more expressive than ours because it allows higher-order functions. We intend to add higher-order functions to our framework in the near future.

III. PROGRAMMING MODEL AND SECURITY PROPERTIES

In this section, we describe our formalism for modeling systems composed of both trusted and untrusted programs and a logic of programs for specifying and reasoning about their security properties. Both the language and the logic generalize prior work on security analysis of protocols [23, 26–28] and trusted computing platforms [29]. In Section III-A, we describe a concurrent programming language for modeling systems and its operational semantics. Among other things, the language supports recursive functions that can be used to define interfaces to which
adversaries are confined. Use of the language to model secure systems and adversary-accessible interface is illustrated through an example in Section III-B. The same example is also used to illustrate reasoning principles in Section IV. Section III-C presents the logic of programs that is used to express security properties of systems modeled in the programming language.

A. Programming Model

We model a system as a set of remote or co-located concurrent threads, each of which executes a sequential program. The program of each thread may either be available for analysis, in which case we call the thread trusted, or it may be unknown, in which case we call the thread untrusted or adversarial. The program of a thread consists of atomic steps called actions and control constructs like conditionals and sequencing. Actions model all operations other than control flow including side-effect free operations like encryption, decryption, and cryptographic signature creation and verification as well as inter-thread interaction through network message sending and receiving, shared-memory reading and writing, etc. In the formal description of our programming model, its operational semantics, and reasoning principles, we treat actions abstractly, denoting them with the letter \( a \) in the syntax of programs and representing their behavior with sound axioms in the logic of programs. The soundness theorem presented in this paper is general, and applies whenever axioms chosen to codify properties of actions are sound.

In order to reason about properties of untrusted threads, the programming model limits their interaction with other threads to stipulated interfaces (see Section IV for details). An interface is a function \( f(x) = e \), with name \( f \), argument \( x \), and body \( e \). The body may include calls to \( f \) as well as other functions, thus allowing for both recursion and mutual recursion. Recursive interfaces are important in many settings including web browsers, e.g., to model event handlers that respond to input from the user, remote servers, and other browser frames.

Formally, the sequential program of each thread is described by an expression \( e \) in the following language of program expressions. \( t \) denotes a term that can be passed as arguments, and over which variables \( x \) range. We do not stipulate a fixed syntax for terms; they may include integers, Booleans, keys, signatures, tuples, etc. However, our language is first-order, so terms may not contain expressions. To simplify reasoning principles, the language is interpreted functionally: all expressions and actions must return a term to the calling site and variables bind to terms. Mutable program variables, if needed, may be modeled as a separate syntactic entity whose contents can be updated through read and write actions, as illustrated in an example later in this section.

Expressions

\[
e \ ::= \ t \mid \text{act}\ a \mid \text{let}(e_1,x,e_2) \mid \text{if}(b,e_1,e_2) \mid \text{call}(f,t)
\]

Function defs

\[
f(x) \triangleq e
\]

The expression \( t \) returns term \( f \) to the caller. \( \text{act}\ a \) evaluates the action \( a \), potentially causing side-effects. \( \text{let}(e_1,x,e_2) \) is the sequential execution construct: it executes \( e_1 \), binds the term obtained from its evaluation to the variable \( x \), and evaluates \( e_2 \). \( \text{if}(b,e_1,e_2) \) evaluates \( e_1 \) if \( b \) is true and evaluates \( e_2 \) otherwise. \( \text{call}(f,t) \) calls function \( f \) with argument \( t \): if \( f(x) = e \) then \( \text{call}(f,t) \) evaluates to \( e\{t/x\} \). \( \Xi(t/x) \) denotes the usual capture-avoiding substitution of the term \( t \) for the variable \( x \) in the syntactic entity \( \Xi \).

Operational Semantics: The operational semantics of our programming language define how a configuration, the collection of all simultaneously executing threads of the system and shared state, reduces one step at a time. Formally, a configuration \( C \) contains a set of threads \( T_1,\ldots,T_n \) and a shared state \( \sigma \). Although we treat the state abstractly in our formal specification, it may be instantiated to model shared memory as well as the network which holds messages in transit. The state may change when threads perform actions, e.g., a send action by one thread may add an undelivered message to the network part of the state. Such interaction between the state and actions is captured in the reduction rule for actions, as described below.

A thread is a triple \( I;K;e \) containing a unique thread identifier \( I \), an execution stack \( K \) and an expression \( e \) that is currently executing (also called active) in the thread. The execution stack records the calling context of the currently executing program as a sequence of frames.

<table>
<thead>
<tr>
<th>Thread id</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame</td>
<td>( F ::= x.e )</td>
</tr>
<tr>
<td>Stack</td>
<td>( K ::=</td>
</tr>
<tr>
<td>Thread</td>
<td>( T ::= I;K;e )</td>
</tr>
<tr>
<td>Configuration</td>
<td>( C ::= \sigma \succ T_1,\ldots,T_n )</td>
</tr>
</tbody>
</table>

Selected reduction rules for our language are shown in Figure 1 (The remaining rules are listed in Appendix A.) The rules rely on judgments for evaluating actions \( a \) and terms \( t \), both of which are treated abstractly. The judgment \( \text{eval}\ t\ t' \) means that term \( t \) evaluates completely to the term \( t' \). For example, in a model that includes standard arithmetic, a concrete instance of the judgment would allow \( \text{eval}\ (3+5)\ 8 \). The judgment \( \sigma;I \succ a \rightarrow \sigma';I \succ t \) means that action \( a \) when executed in thread \( I \) updates the shared state \( \sigma \) to \( \sigma' \) and returns term \( t \). As an example, in a system with shared memory, \( \sigma \) may be a map from locations \( l \) to terms \( t \), the action \( \text{write}\ l,t \) may change contents of location \( l \) to \( t \), and the following judgment may hold: \( \sigma;I \succ \text{write}\ l,t \rightarrow \sigma[l \mapsto t];I \succ t \).

Our primary reduction relation, that for configurations, has the form \( \sigma \rightarrow \tau \). It interleaves reductions of individual
threads in the configuration in an arbitrary order (rule `red-config`). Reductions of threads are formalized by a reduction relation for threads: \( \sigma \triangleright T \triangleleft \sigma' \triangleright T' \). This reduction relation is standard for functional programs with side-effects and we omit its description. Reductions caused by rules other than (red-act) are called administrative reductions, whereas those caused by (red-act) are called effectual reductions.

Our reasoning principles establish properties of traces. A trace \( T \) is a finite sequence of reductions \( C \rightarrow C' \) starting from an initial configuration. We associate a time point \( u \) with every reduction on a trace. A time point is either a real number or \(-\infty\) or \(\infty\). The only constraint on time points associated with a trace is that they be monotonically increasing along the trace. Diagrammatically, we represent a trace as follows (\( u_i \) is the time at which \( C_{i-1} \) reduces to \( C_i \))

\[
\frac{u_0}{C_0} \quad \frac{u_1}{C_1} \quad \cdots \quad \frac{u_n}{C_n}
\]

We assume that individual reduction steps happen instantaneously at the time associated with them on the trace and that the state \( \sigma \) on the trace does not change between two consecutive reductions. Specifically, in the above illustration, if \( C_i = \sigma_i \triangleright T_{i,1} \triangleright \cdots \triangleright T_{i,n} \), then we assume that the state of the trace is \( \sigma_i \) in the interval \( [u_i, u_{i+1}) \). \( u_0 \) is called the start time of the trace. In addition to the sequence of reductions and time points, a trace may also include auxiliary information, such as contents of memory at the start time.

**B. An Extended Example**

We introduce an illustrative, running example that is inspired by web mashups that aggregate financial information, e.g., http://www.mint.com. The web mashup consists of three trusted frames running concurrently in a web browser window. Other malicious frames, constrained by the interfaces described below, may also be running simultaneously. Two trusted frames belong to bank1.com and bank2.com with whom the user of the mashup has accounts. The third trusted frame is an aggregator frame belonging to aggregator.com. It waits for the user to send it a signal (say, by pressing a button), then communicates with the two other frames through messages (described below) asking them to provide

```
Figure 2: A Web Mashup Example
```

```
Interfaces
send_auth(i,m) = send i, self, m
write_acc(c,v) =
  if ACM(self,c,write)
    then write c, v
  else ()
read_acc(c) =
  if ACM(self,c,read)
    then read c
  else ()

Access control matrix
ACM(User,total_box,read)
ACM(Agg,total_box,write)
ACM(User,pass_box,write)
ACM(Agg,pass_box,read)

Known Programs
agg_loop() =
  _ = recv User;
  u_pass = read_acc(pass_box);
  nonce = new;
  send_auth(Bank1, (u_pass, nonce));
  send_auth(Bank2, (u_pass, nonce));
  (v1, nonce') = recv Bank1;
  (v2, nonce'') = recv Bank2;
  if (nonce = nonce' = nonce'')
    then write_acc(total_box, v1 + v2)
  else ();
  agg_loop()

bank1_loop() =
  (u_pass, nonce) = recv Agg;
  page = get bank1.com/balance, u_pass;
  v = parse_balance(page);
  send_auth(Agg, (v, nonce));
  bank1_loop()

bank2_loop() =
  (u_pass, nonce) = recv Agg;
  page = get bank2.com/balance, u_pass;
  v = parse_balance(page);
  send_auth(Agg, (v, nonce));
  bank1_loop()

agg_body = agg_loop();
bank1_body = bank1_loop();
bank2_body = bank2_loop();
```
the user’s account balances in the respective banks. It then computes the total of the two balances and displays them to the user.

**Communication Model and Actions:** We model frames as threads, calling the three threads Bank1, Bank2, and Agg (for aggregator). Following standard browser interfaces [7], we assume that the native communication between frames is message-passing: the action send $I, J, m$ sends message $m$ to thread $I$, pretending that it comes from thread $J$. Note that the actual sender may or may not be $J$. Thus, the communication model allows sender-id spoofing for messages. Dually, the action recv $I$ returns a message $m$ purportedly sent by thread $I$ to the calling thread, if such a message exists, else it blocks. In addition, we allow shared mutable cells with access control. Cells may model text boxes on the screen as well as shared memory. The actions read $c$ and write $c, t$ read the cell $c$ and write it to, respectively. For this example, we need two cells: pass_box through which the user provides its password to the aggregator, and total_box where the aggregator displays the total to the user. Finally, we assume the action get $url, p$ that retrieves the document at the URL $url$ using credential $p$ for logging in, and the action new that generates a fresh nonce.

**Interfaces:** The actions send, read, and write described above do not enforce any security. For instance, the action send allows sender spoofing. Similarly, write allows any thread to write any location. Clearly, any browser that allows direct access to these actions will not be able to provide security guarantees for its applications. Accordingly, we assume that these actions are not directly accessible to threads running in the browser. Instead, threads are allowed access to interfaces (functions) that layer security over these actions. These browser interfaces are defined in Figure 2 and are named send_auth, read_acc and write_acc. For better readability, we omit the keywords act and call and write let$(e_1, e_2)$ as $(x = e_1); e_2$. The send_auth function sends a message to another frame, setting its sender to the calling thread (named by the metavariable self). Similarly, read_acc and write_acc check a predicate ACM$(I, c, p)$ which is true if thread $I$ is allowed to perform operation $p \in \{\text{read, write}\}$ on cell $c$. The predicate is also defined in Figure 2. The integrity of the mashup is contingent on the assumption that malicious threads running concurrently with the mashup have access to the actions send, read, and write only through these interfaces.

**Programs:** Figure 2 lists the programs agg_body, bank1_body and bank2_body of Agg, Bank1, and Bank2, respectively. Agg waits for a signal from the user (modeled as another thread User, whose program is irrelevant). It then reads the user’s password from the cell pass_box, generates a new nonce, and sends the password and nonce to the two banks (for simplicity, we assume that the user’s password in both the banks is the same). It waits for replies from the two banks, checks that the nonces returned in the replies equal the one it sent, adds the values returned by the banks, puts the sum in total_box for the user to see, and loops back to the waiting state.

The two bank frames run similar programs. Each frame waits for a password and nonce from the aggregator, then retrieves the balance of the user from its remote server and sends the balance and nonce to the aggregator. It then returns to its waiting state.

**Security Property:** We are interested in proving an integrity property of the mashup, namely, that if the user sees a total in total_box, then that total was obtained by adding balances in the user’s bank accounts at bank1.com and bank2.com that were obtained after the user signaled the aggregator frame. (This property is formalized in Section III-C after we introduce our logic, and proved in Section IV-C.) This integrity property requires that only the aggregator may write to the cell total_box, and that any message purportedly sent by either bank and received by the aggregator actually have been sent by the respective bank. These properties are enforced by the interfaces accessible to programs and, in particular, to malicious threads. Indeed, if a malicious thread could execute either the write action or the send action directly, it could violate this integrity property by either writing an incorrect value in total_box or by sending the aggregator an incorrect value as coming from a bank.

**C. Security Properties**

We represent security properties as formulas of a logic whose syntax is shown below. We are primarily interested in reasoning about safety properties, and accordingly, our reasoning principles are designed to support only such properties.

\[
\varphi, \psi ::= p \circledast u \mid b \mid t = t' \mid u_1 \leq u_2 \mid \varphi \land \psi \mid \varphi \lor \psi \mid \neg \varphi \mid T \mid \bot \mid \forall x. \varphi \mid \exists x. \varphi \mid \varphi \circledast u
\]

The logic includes only one modal formula, $p \circledast u$, which means that the atomic formula $p$ holds at time $u$ (on the trace against which the formula is being interpreted). $p \circledast u$ is a hybrid modality [31, 32]. It is known that all operators of linear temporal logic (LTL) can be expressed using $p \circledast u$ and quantifiers, so this logic is at least as expressive as LTL. In addition to increasing expressiveness, using a hybrid logic instead of LTL allows us to express order between actions in security properties in an intuitive manner, facilitates reasoning about invariants of programs modularly (Section IV) and also facilitates rely-guarantee reasoning without the need for additional induction principles (Section VI).

$b$ denotes a Boolean term from the syntax of the language. Atomic formulas $p$ are terms applied to predicates, which may represent either the execution of certain actions (corresponding to the language’s actions $a$) or properties of state. For example, a predicate Mem$(l, t)$ that checks contents of
memory may be defined by saying that $T \models \text{Mem}(l, t) @ u$ if and only if, in trace $T$, memory location $l$ contains term $t$ at time $u$. Similarly, in an analysis of security protocols, $T \models \text{Enc}(I, t, k) @ u$ may hold if and only if thread $I$ in trace $T$ encrypts term $t$ with key $k$ at time $u$.

**Example III.1.** We formalize in our logic the integrity property described informally at the end of the example in Section III-B. Assume that the predicate $\text{Mem}(c, v) @ u$ means that cell $c$ contain value $v$ at time $u$, $\text{Recv}(I, J, v) @ u$ means that thread $I$ receives at time $u$ a message sent previously by thread $J$, $\text{Read}(I, c, v) @ u$ means that thread $I$ reads value $v$ from cell $c$ at time $u$, and $\text{Get}(I, url, p, v) @ u$ means that thread $I$ retrieves page $v$ from URL $url$ using credential $p$ at time $u$. Then the integrity property in Section III-B may be expressed as follows:

$$(\text{Mem}(\text{total}\_\text{box}, v) @ u_1 \land$$
$$\text{Mem}(\text{total}\_\text{box}, v) @ u_2 \land$$
$$(u_1 < u_2) \land (v' \neq v) \land$$
$$\exists_{v_1, u_y, v_p, u_z}.$$}

The conditions of the implication in the above property state that $\text{total}\_\text{box}$ contains a different value $v$ at time $u_2$ than at time $u_1$. The conclusion means that, under these conditions, the user must have requested a total at time $u_y$, and subsequently, bank balances must have been fetched from bank1.com and bank2.com, and that $v$ equals their sum. It is instructive to observe the use of the modality $@$ to represent a relative order among events in this example.

**Logic of Programs:** On top of the temporal logic described above, we build a logic of programs to reason about properties of traces obtained by executing known or interface-confined programs. Prior experience with security analysis of protocols [23, 26–28] and trusted computing platforms [29] shows that in addition to standard pre- and post-conditions, analysis of secure systems often requires reasoning that properties hold throughout the execution of a program. We call such properties invariants and introduce a novel construct to represent them in the logic of programs.

Specifically, we express pre- and post-conditions as well as invariants using six kinds of assertions, generically denoted $\mu, \nu$.

**Assertions $\mu, \nu := [e](u_b, u_e, i, x)\varphi \mid [e](u_b, u_e, i)\varphi \mid$$
$$[f](y, u_b, u_e, i, x)\varphi \mid [f](y, u_b, u_e, i)\varphi \mid$$
$$[a](u_b, u_e, i, x)\varphi$$

In these assertions, $u_b, u_e, i, x, y$ are bound variables whose scope is $\varphi$. (Recall that $\varphi$ denotes a formula in the temporal logic described earlier.) The intuitive meanings of the six assertions are listed below; formal semantics are postponed to Section V.

- $[e](u_b, u_e, i, x)\varphi$: If in a trace $T$, the active expression of thread $I$ at time $u'_b$ is $e$, and this expression returns term $t$ to its caller at time $u'_e$, then $\varphi(u'_b/u_b)\{u'_e/u_e\}(I/i\{t/x\})$ holds.
- $[e](u_b, u_e, i)\varphi$: If in a trace $T$, the active expression of thread $I$ at time $u'_b$ is $e$, and this expression does not return to its caller until time $u'_e$, then $\varphi(u'_b/u_b)\{u'_e/u_e\}(I/i)$ holds.
- $[f](y, u_b, u_e, i, x)\varphi$: If in a trace $T$, thread $I$ calls function $f$ with argument $t'$ at time $u'_b$, and this function returns term $t$ to its caller at time $u'_e$, then $\varphi(t'/y)\{u'_b/u_b\}\{u'_e/u_e\}(I/i\{t/x\})$ holds.
- $[f](y, u_b, u_e, i)\varphi$: If in a trace $T$, thread $I$ calls function $f$ with argument $t'$ at time $u'_b$, and this function does not return to the caller until time $u'_e$, then $\varphi(t'/y)\{u'_b/u_b\}\{u'_e/u_e\}(I/i)$ holds. (Recall from Section III-A that an effectual reduction is the reduction of an action, as opposed to the reduction of a control flow construct.)
- $[a](u_b, u_e, i, x)\varphi$: If in a trace $T$, the active expression of thread $I$ at time $u'_b$ is act $a$, and this expression returns term $t$ to its caller at time $u'_e$, then $\varphi(u'_b/u_b)\{u'_e/u_e\}(I/i\{t/x\})$ holds.

Assertions $[e](u_b, u_e, i, x)\varphi$ and $[f](y, u_b, u_e, i, x)\varphi$ specify the behavior of a program that completes execution. They are generalizations of partial correctness assertions from other program logics like Hoare logic. We do not need to specify pre- and post-conditions separately because we can encode them in $\varphi$ using the construct $p @ u$. For example, consider the function $f(x) \triangleq 1 + \text{let}(\text{act read I}, z, \text{act write l, z+x})$ that increments the contents of the private memory location $l$ by its argument $x$. We can specify this function in our logic of programs as $[f](y, u_b, u_e, i, x) \forall z. (\text{Mem}(l, z) @ u_b \supset \text{Mem}(l, y + z) @ u_e)$ and $x = ()$. A salient difference from other program logics like Hoare logic is that variables bound in an expression $e$ or the body of a function $f$ cannot appear in its specification. In the example above, the variable $x$ in the body of the function differs from the $x$ in its specification. In the former case, the variable is the argument of the function whereas in the latter case it is the result of the function. This is unsurprising since our treatment of variables is functional; other related formalisms like Hoare-Type Theory [33] follow similar conventions.

The assertions $[e](u_b, u_e, i)\varphi$ and $[f](y, u_b, u_e, i)\varphi$ specify invariants of programs – $\varphi$ holds while the program $(e$ or $f$) is executing. Prior work on Protocol Composition Logic [23, 26–28] and the Logic of Secure Systems [29]
encode invariants using standard partial correctness assertions for programs without conditionals, function calls, and recursion. Our treatment is novel, strictly more general, and it allows for modular analysis of invariants of programs with arbitrary control flow constructs.

Whereas Section IV presents a proof system for establishing the four assertions $\{e\} (u_b, u_c, i, x) \varphi$, $\{e\} (u_b, u_c, i) \varphi$, $\{f\} (y, u_b, u_c, i, x) \varphi$, and $\{f\} (y, u_b, u_c, i) \varphi$, we do not stipulate rules for establishing the remaining two assertions, $\lbrack (u_b, u_c, i) \varphi \rbrack$ and $\lbrack a \rbrack (u_b, u_c, i, x) \varphi$ that specify properties of administrative reductions and actions, respectively. Instead, these two assertions must be established through sound axioms that would depend on the representation of state and actions chosen; the proof rules for establishing the other four assertions use these two assertions as black-boxes. The soundness theorem of our proof system applies whenever the sender-id of the message to $i'$.

\begin{align*}
\{\text{bank1}_\text{body}\} &\langle u_b, u_c, i \rangle \\
\forall u, j, i', x. (u_b < u < u_c) \supset &
\text{Send}(i, j, i', x) \oplus u \\
\exists u, i, u_p. (u_b < u < u_p) \wedge &
\exists i', \text{nonce}, u_{\text{pass}}'. \\
\text{Recv}(i, \text{Agg}, \langle u_{\text{pass}}, \text{nonce} \rangle) &\oplus u_i \wedge \\
\text{Get}(i, \text{bank1}/\text{balance}, u_{\text{pass}}, v_i') &\oplus u_{p1} \wedge \\
x &\equiv \text{parse_balance}(v_i'), \text{nonce}'
\end{align*}

This invariant states that while a thread $i$ executes the program $\text{bank1}_\text{body}$, if $i$ sends a message $x$, then $x$ must be a pair containing a balance obtained from $\text{bank1}/\text{balance}$ and a nonce obtained from $\text{Agg}$. Because we need this property to hold irrespective of how far $\text{bank1}_\text{body}$ executes, the property must be stated and proved as an invariant, not a partial correctness assertion.

IV. COMPOSITIONAL REASONING PRINCIPLES

Next, we codify in a proof system compositional reasoning principles for establishing assertions about programs as well as security properties. In addition to standard rules for proving temporal formulas and syntax-directed rules for proving assertions about programs and functions, our proof system includes two rules of inference that allow deduction of properties of threads from invariants of their programs. We call a thread trusted if the program it executes is known, else we call the thread untrusted or adversarial. Using the first rule (Section IV-B), we may combine knowledge that a particular thread is executing a known program with any invariant of the program to deduce that the formula in the invariant holds forever. The second rule (Section IV-C) embodies our central idea of reasoning about unknown, potentially adversarial, code by confining it to interfaces: if we know that an adversarial thread has access only to certain interfaces, then under certain conditions, we can show that a common invariant of all those interfaces always holds in the system, regardless of the manner in which the adversarial thread uses those interfaces. For instance, in the example of Section III-B, we use this reasoning principle to establish that whenever a thread writes to the location total_box, then that thread must be the aggregator.

Formally, proofs establish one of two hypothetical judgments: $\Sigma, \Gamma \vdash \varphi$ and $\Sigma; \Delta \vdash \mu$. In both judgments $\Sigma$ is a set of first-order variables that may occur free in the rest of the judgment, $\Gamma$ is a list of assumed formulas of the temporal logic and $\Delta$ contains assumed specifications of functions. A proof establishing either $\Sigma, \Gamma \vdash \varphi$ or $\Sigma; \Delta \vdash \mu$ is parametric in all variables in $\Sigma$, i.e., it holds for all ground instances of the variables.

$$
\begin{align*}
\Sigma &::= \cdot | \Sigma, x \\
\Gamma &::= \cdot | \Gamma, \varphi \\
\Delta &::= \cdot | \Delta, \{f\} (y, u_b, u_c, i, x) \varphi \mid \Delta, \{f\} (y, u_b, u_c, i) \varphi
\end{align*}
$$

The judgment $\Sigma; \Gamma \vdash \varphi$ coincides with the standard hypothetical judgment of first-order classical logic with equality
Assertions manifest in the judgment \(\Sigma; \Gamma; \Delta \vdash \mu\) are established by an analysis of the program in \(\mu\) through rules described in Section IV-A. Additionally, there are inference rules to combine reasoning in the temporal logic with reasoning about assertions. For instance, if \(\{e\} \langle u_b, u_e, i, x \rangle \varphi \) and \(\varphi \models \varphi'\), then one of the rules of inference allows deduction of \(\{e\} \langle u_b, u_e, i, x \rangle \varphi'\). Such rules are common in program logics; we allow several such rules that are listed in Appendix B. As mentioned earlier, Sections IV-B and IV-C describe the rule for proving properties of trusted and untrusted threads, respectively.

A. Reasoning About Specifications of Programs

Specifications of a program are proved through syntax-directed analysis of the program. Selected rules for establishing program specifications are shown in Figure 3; the remaining rules are shown in Appendix B. As mentioned in Section III-C, the rules of our proof system rely on the abstract judgments \([u]\langle u_b, u_e, i, x \rangle \varphi\) and \(\langle u_b, u_e, i \rangle \varphi\) (e.g. rule (PA)). The rules are modular: specifications of a program are established by combining specifications of sub-programs. For instance, we may justify the rule (PL) as follows. In the conclusion of the rule we wish to establish a partial correctness assertion of the expression \(\text{let}\langle e_1, y, e_2\rangle\). If this expression is active in thread \(i\) at time \(u_b\) and returns value \(x\) at time \(u_e\), then through an analysis of the operational semantics it follows that at some time \(u_m\) after \(u_b\), \(e_1\) must have become active, then at a later time \(u_m'\), \(e_1\) would have returned some value \(y\) to \(e_2\), which would have become active, and finally at time \(u_e\), \(e_2\) would have returned \(x\). So if \(\langle u_b, u_m, i \rangle \varphi_1\), \(\langle u_m, u_m', i, y \rangle \varphi_2\), and \(\langle u_e, i, x \rangle \varphi_3\) hold (as in the premises of the rule), then \(\exists y. \exists u_m. \exists u_m'. ((u_b < u_m < u_m' < u_e) \land \varphi_1 \land \varphi_2 \land \varphi_3)\) must hold. Observe that it is necessary to existentially quantify the variables \(y\), \(u_m\), and \(u_m'\) in the conclusion because during reasoning, we cannot determine their exact values. This justifies the conclusion of the rule. Other rules for establishing partial correctness assertions can be justified similarly.

Rules for establishing invariance assertions are more involved, but are also modular. We illustrate their justification through the rule (IL). In the conclusion of the rule we wish to establish an invariant that holds while \(\text{let}\langle e_1, y, e_2\rangle\) executes. If this expression starts executing in thread \(i\) at time \(u_0\) but does not return until time \(u_e\), then there are only three possibilities: (a) \(e_1\) does not start executing until time \(u_e\), (b) \(e_1\) started executing at some time \(u_m\), but does not return until time \(u_e\), or (c) \(e_1\) starts executing at time \(u_m\), returns at time \(u_m'\), \(e_2\) starts executing at time \(u_m'\), but does not return until time \(u_e\). If we can show that \(\varphi\) holds in each of these three cases, then \(\varphi\) is in invariant of \(\text{let}\langle e_1, y, e_2\rangle\). The premises of the rule account for exactly these three cases: the first premise accounts for case (a), premises 2–4 account for case (b), and the remaining premises account for case (c).

Rules (PF) and (IF) for proving partial correctness assertions and invariants of functions check the corresponding specification on the bodies of the respective functions. In order to account for the possibility of recursion, we also assume the function’s specification when we check the body of the function by adding it to the context \(\Delta\) in the premises. It is not obvious that this approach is sound and accounting for it complicates our proof of soundness (see Section V).

Example IV.1. The invariant of Example III.2 can be established using the rules of Figure 3 and some straightforward axioms for relevant actions. For instance, we need to assume that \([\text{send}\ j, i', m]\langle u_b, u_e, i, x \rangle \text{Send}(j, i', i, m) @ u_e \land \forall u \in \langle u_b, u_e \rangle. \neg \text{Send}(j, i', i, m') @ u\). This axiom as well as others needed to prove this particular invariant are derived from prior work on Protocol Composition Logic and the Logic of Secure Systems [26, 29], where they were shown to be sound.

B. Reasoning About Trusted Threads

In this section, we present a rule to prove properties of trusted threads from knowledge of their programs. Formally, in the logic, we say that HonestThread\((I, e)\) if thread \(I\) executes program expression \(e\) only. Let Start\((I) @ u\) hold if at time \(u\), thread \(I\) is ready to execute, but has not performed any reduction. The following rule, based on the Honesty rule in prior work on Protocol Composition Logic [26], allows us to prove a property of thread \(I\) from an invariant of \(e\) if HonestThread\((I, e)\).

\[
\Sigma; \Gamma; \vdash \{e\} \langle u_b, u_e, i \rangle \varphi(u_b, u_e, i) \\
\Sigma; \Gamma; \vdash \text{HonestThread}(I, e) \\
\Sigma; \Gamma; \vdash \text{Start}(I) @ u \\
\text{HONTH}
\]

The justification for this rule follows: since HonestThread\((I, e)\) and Start\((I) @ u\), it must be the case that \(e\) is the active expression in \(I\) at time \(u\). Further, since \(e\) is the top-level program of \(I\), it can never return. Hence, by the definition of \(\{e\} \langle u_b, u_e, i \rangle \varphi(u_b, u_e, i)\), \(\varphi(u, u', I)\) must hold for any \(u' > u\). (We do not stipulate rules for proving either HonestThread\((I, e)\) or Start\((I) @ u\) since they codify system-specific assumptions. Instead such formulas must be explicit hypotheses in \(\Gamma\).)

Example IV.2. In the web mashup example of Section III-B, the thread Bank1 is known to execute the program bank1_body. Hence, by definition, HonestThread\((Bank1, bank1_body)\). Further, we may assume that the thread exists in the trace from the

\[\text{bank1_body} \ldots @ u\]
\[
\Sigma; \Gamma; \Delta \vdash [a](u_b, u_e, i, x) \varphi_{PA}\]
\[
\Sigma; \Gamma; \Delta \vdash \{act a\}(u_b, u_e, i, x) \varphi_{IA}\]
\[
\Sigma; \Gamma; \Delta \vdash [[(u_b, u_m, i)] \varphi_{1} \quad \Sigma; \Gamma; \Delta \vdash [e_1](u_m, u_m', i, y) \varphi_{2} \quad \Sigma, y; \Delta \vdash [e_2](u_m, u_e, i, x) \varphi_{3} \\
\Sigma, y; \Delta \vdash \{act a\}(u_b, u_e, i, x) \varphi_{3}\]
\[
\Sigma, y; \Delta \vdash \{act a\}(u_b, u_e, i) \varphi_{3}\]
\[
\Sigma; \Gamma; \Delta \vdash \{act a\}(u_b, u_e, i) \varphi_{3}\]
\[
\Sigma; \Gamma; \Delta \vdash \{act a\}(u_b, u_e, i) \varphi_{3}\]
\[
\Sigma; \Gamma; \Delta \vdash \{act a\}(u_b, u_e, i) \varphi_{3}\]

**Figure 3:** Selected modular rules for establishing program specifications

beginning, so Start(Bank1) @ -∞. In Example III.2, we listed an invariant of bank1_body. Applying the rule (HONTH) to that invariant we may conclude that:

∀u' > -∞. ∀u, j, i', x. (∞ < u < u') ⊃
\[
\text{Send(Bank1, j, i', x) @ u ⊃} \]
\[
\exists u_{i,1}, u_{i,1}. (u_{i,1} < u_{i,1}' < u) \land \exists j, i, 'u_{i,1}, u_{pass_1}.'
\]
\[
\text{Recv(Bank1, Agg, (u_{pass_1}', nonce_1')) @ u_{i,1} \land} \]
\[
\text{Get(Bank1, bank1.com/balance, u_{pass_1}', v_1') @ u_{i,1} \land} \]
\[
x = (\text{parse_balance}(v_1'), nonce_1')\]

Instantiating with u' = ∞ and simplifying, we obtain:

∀u, j, i, x. Send(Bank1, j, i, x) @ u ⊃
\[
\exists u_{i,1}, u_{i,1}. (u_{i,1} < u_{i,1}' < u) \land \exists j, i, 'u_{i,1}, u_{pass_1}.'
\]
\[
\text{Recv(Bank1, Agg, (u_{pass_1}', nonce_1')) @ u_{i,1} \land} \]
\[
\text{Get(Bank1, bank1.com/balance, u_{pass_1}', v_1') @ u_{i,1} \land} \]
\[
x = (parse\_balance(v_1'), nonce_1')\]

This formula states a universal property of the thread Bank1 (as opposed to the program expression bank1_body), namely, that whenever Bank1 sends out a message x, x is a pair consisting of a nonce obtained from Agg and a balance obtained from bank1.com/balance. This property is crucial in proving the integrity property of Example III.1.

C. Reasoning About Interface-Conﬁned Untrusted Threads

As opposed to trusted threads, whose security properties may be established by analysis of their programs, the programs of untrusted or adversarial threads are not known, so proving their security properties may seem impossible. Yet, in practice, security of systems often relies on conﬁnement of behavior of untrusted threads. For instance, the integrity property presented at the end of Section III-B holds only if we can show that an untrusted thread cannot write to the cell total_box. The latter holds because all threads are forced to use the interface write_acc in order to write a cell and that interface prohibits any thread except Agg from writing total_box. In this section, we develop reasoning principles that allow us to infer properties from the knowledge that an untrusted thread has been restricted, or conﬁned, to certain known interfaces. We deﬁne an interface, denoted F, as a set of functions. In general, an adversary that is conﬁned to F may construct a new set of functions G that call functions of F and themselves and combine calls to functions of F and G in any way it chooses. To formally represent such an adversary, we need a few deﬁnitions.

**Deﬁnition IV.3 (F-confined expressions).** Given an interface F, we call an expression e F-confined if the following hold: (a) All occurrences of call in e have the form call(f, l), where f ∈ F, and (b) act does not occur in e.

**Deﬁnition IV.4 (F-limited functions).** Given an interface F, we call a set of functions G = {g_k | g_k(y) = e_k} F-limited if the body e_k of each function is (F ∪ G)-confined.

**Deﬁnition IV.5 (F-confined thread).** A (untrusted) thread I is said to be F-confined if I executes a program e and there is a F-limited interface G such that e is (F ∪ G)-conﬁned. The predicate Confined(I, F) holds iff I is F-confined.\(^2\)

**Deﬁnition IV.6 (Compositional formula).** A formula \(\varphi(u_b, u_e, i)\), possibly containing the free variables \(u_b, u_e, i\), is called compositional if \(\forall u_b, u_m, u_e, i. ((u_b < u_m < u_e) \land \}

\(f(z) \overset{e}{=} e, \Sigma; y; \Delta, \{f\}(y, u_b, u_e, i, x) \varphi \vdash \{e\}(y/z)\langle u_b, u_e, i, x) \varphi_{IF}\)

\(f(z) \overset{e}{=} e, \Sigma; y; \Delta, \{f\}(y, u_b, u_e, i, x) \varphi \vdash \{e\}(y/z)\langle u_b, u_e, i, x) \varphi_{IF}\)
\(\varphi(u_b, u_m, i) \land \varphi(u_m, u_e, i) \supset \varphi(u_b, u_e, i)\).

Roughly, a formula \(\varphi(u_b, u_e, i)\) describing some property over an interval \((u_b, u_e]\) is compositional if whenever the formula holds on two adjoining intervals, it also holds on the union of the intervals. In general, if \(\varphi(u_b, u_e, i)\) encodes the fact that a safety property holds throughout the interval \((u_b, u_e]\), then \(\varphi(u_b, u_e, i)\) will be compositional.

We codify our reasoning principles for untrusted, interface-confined threads in the following rule:

\[
\begin{align*}
& (\varphi(u_b, u_e, i) \text{ (compositional)}) \\
& \forall f \in \mathcal{F}. \quad (; \Gamma; \vdash \langle f \rangle(y, u_b, u_e, i) \varphi(u_b, u_e, i)) \\
& \forall f \in \mathcal{F}. \quad (; \Gamma; \vdash \langle f \rangle(y, u_b, u_e, i) \varphi(u_b, u_e, i)) \\
& \Sigma, \Gamma, \Gamma' \vdash \forall u_e. \varphi(-\infty, u_e, I) \\
& \text{RES}
\end{align*}
\]

The informal justification for the rule (RES) is that, owing to its confinement to \(\mathcal{F}\), the reduction of \(I\) up to any time point \(u_e\) can be split into calls to functions in \(\mathcal{F}\) interspersed with administrative reductions of the adversary’s choosing. Since \(\varphi\) is a partial correctness assertion of all functions in \(\mathcal{F}\) (fourth premise) and administrative reductions (second premise), as well as invariant of all functions in \(\mathcal{F}\) (third premise), it must hold over all these splits. Therefore, due to the compositionality of \(\varphi\) (first premise), \(\varphi(-\infty, u_e, I)\) must hold. The formal justification of this rule is a non-trivial part of the soundness theorem (Section V) because we must consider all \(\mathcal{F}\)-confined programs that the thread \(I\) may execute.

**Example IV.7.** An important property needed in the proof of the integrity property of Example III.1 is the following:

\[\forall i, v, u. \text{Write}(i, \text{total\_box}, v) @ u \supset i = \text{Agg}\]

Since this property speaks of all threads \(i\), it requires reasoning about untrusted threads. Here, we show that if all threads are confined to the interfaces \(\text{send\_auth}, \text{write\_acc}, \text{read\_acc}\) and the following two interfaces that mimic the \(\text{recv}\) and \(\text{get}\) actions, then this property can be established using the rule (RES).

\[\text{recv\_i}(x) \triangleq \text{recv} x \quad \text{get\_i}(x) \triangleq \text{get} x\]

Define the set \(\mathcal{F} = \{\text{send\_auth}, \text{write\_acc}, \text{read\_acc}, \text{recv\_i}, \text{get\_i}\}\). Assume that \(\forall i. \text{Confined}(i, \mathcal{F})\). Due to the fact that the access control matrix allows only \(\text{Agg}\) to write cell \(\text{total\_box}\), it can be shown that the following hold:

\[
\begin{align*}
& \forall f \in \mathcal{F}. \quad \langle f \rangle(y, u_b, u_e, i) \\
& \forall v, u. \left(\left(u_b < u \leq u_e\right) \land \text{Write}(i, \text{total\_box}, v) @ u\right) \\
& \quad \supset i = \text{Agg} \\
& \forall f \in \mathcal{F}. \quad \langle f \rangle(y, u_b, u_e, i, x) \\
& \forall v, u. \left(\left(u_b < u \leq u_e\right) \land \text{Write}(i, \text{total\_box}, v) @ u\right) \\
& \quad \supset i = \text{Agg} \\
& \langle (u_b, u_e, i) \rangle \\
& \forall v, u. \left(\left(u_b < u \leq u_e\right) \land \text{Write}(i, \text{total\_box}, v) @ u\right) \\
& \quad \supset i = \text{Agg}
\end{align*}
\]

Further, \(\forall i, u. \left(\left(u_b < u \leq u_e\right) \land \text{Write}(i, \text{total\_box}, v) @ u\right) \\
\supset i = \text{Agg}\) is compositional by definition. Therefore, by rule (RES) we may derive that

\[
\forall i, u. \left(\left(-\infty < u \leq u_e\right) \land \text{Write}(i, \text{total\_box}, v) @ u\right) \\
\supset i = \text{Agg}
\]

Choosing \(u_e = \infty\) and simplifying, we obtain the required property \(\forall i, v, u. \text{Write}(i, \text{total\_box}, v) @ u \supset i = \text{Agg}\). The reader may observe that if untrusted threads are not confined to the interface set \(\mathcal{F}\), then this property may not hold. For instance, if an untrusted thread has access to the \(\text{write}\) action through some other interface that does not check the access control matrix, then the invariants in the premise of the (RES) rule may not hold.

**Proof of Integrity Property of Example III.1:** The integrity property from Example III.1 can be proved using the inference rules presented in this section, assuming that all untrusted threads are confined to the interface set \(\mathcal{F}\) defined in Example IV.7. Details of the proof are in Appendix F. Briefly, the proof relies on the property of the trusted thread Bank1 derived in Example IV.2 using the (HONTH) rule, and similarly derived properties about the threads Bank2 and Agg. We also need the property of untrusted threads derived in Example IV.7 using the (RES) rule and a similarly derived property that any message purportedly sent by thread \(I\) was actually sent by thread \(\hat{I}\), i.e, the absence of sender-id spoofing. This property is a consequence of the fact that the only function in \(\mathcal{F}\) that contains a \(\text{send}\) action, namely \(\text{send\_auth}\), ensures that the sender-id in an outgoing message is set correctly to the actual sender.

**V. SEMANTICS AND SOUNDNESS THEOREM**

In this section, we formally define the trace semantics of temporal formulas \(\varphi\) and assertions \(\mu\) and show that our proof rules are sound, i.e., any formula or assertion proved using the rules is valid in the semantics. This provides foundational justification for the reasoning principles of Section IV.

**Semantics:** Since our programming model and the logic of programs are parametric in the syntax of terms and predicates, we assume that interpretations of these entities are given. Let \([t]\) denote the semantic interpretation of the term \(t\) in some domain and let \(\models\) denote equality in the domain. For interpreting atomic formulas, we assume the existence of a Boolean valued function \(V(T, u, p)\) \((T\) is a trace, \(u\) is a ground time point, and \(p\) is a ground atomic formula) such that \([t] \models [t']\) implies \(V(T, u, p(t/x)) = V(T, u, p(t'/x))\). Given these assumptions, we may define the semantics \(\mathcal{M} \models \varphi\) of ground temporal formulas \(\varphi\) in a standard manner (see Appendix C for details).

In order to define semantics of assertions, we need a notion of the suffix of a trace, also called a subtrace.

**Definition V.1 (Subtraces).** Let \(T\) be the trace

\[u_0 \rightarrow C_0 \rightarrow u_1 \rightarrow C_1 \rightarrow \cdots \rightarrow u_n \rightarrow C_n\]
For any $k \geq 0$, we define the truncation of $T$ to $k$, written $\text{trunc}(T, k)$ as the trace which contains only the last $k+1$ configurations of $T$. If $k > n$ then $\text{trunc}(T, k) = T$.

Semantics of ground assertions $\mu$ are represented through the judgment $T, n \vdash \mu$, which roughly means that the assertion $\mu$ holds in the subtrace $\text{trunc}(T, n)$. The additional information $n$ is needed to prove soundness for recursive functions. One representative clause of the definition of $T, n \vdash \mu$ is shown below; others are listed in Appendix C.

- $T, n \vdash \{e\}(u_b, u_e, i, x) \varphi$ holds iff $T \vdash \varphi\{u'_b/u_b\}\{u'_e/u_e\}\{I/i\}$ whenever the following pattern matches the subtrace $\text{trunc}(T, n)$ for some $u' < u'_b'$, there is no reduction in thread $I$ in the interval $(u'_e, u'_b')$, and the stack of $I$ has suffix $K$ in the interval $(u''_b, u''_e)$.

Finally, we define semantics of hypothetical judgments of the proof system.

- $T \vdash (\Sigma; \Gamma \vdash \varphi)$ if for every grounding substitution $\theta$ with domain $\Sigma$, $T \vdash \varphi\theta$ implies $T \vdash \varphi\theta$.
- $T \vdash (\Sigma; \Gamma \vdash \Delta \vdash \mu)$ if for every grounding substitution $\theta$ with domain $\Sigma$ and every $n, T \vdash \Gamma\theta$ and $T, n \vdash \Delta\theta$ imply $T, n \vdash \mu\theta$.

**Soundness:** Assuming that the axioms chosen to reason about the assertions $\{a\}(u_b, u_e, i, x) \varphi$ and $\{u_b, u_e, i\} \varphi$ are valid in the semantics, we can show that any hypothetical judgment established using the proof system of Section IV is semantically valid. As a result, any instance of our reasoning principles is sound if we choose sound axioms for actions and administrative reductions.

**Theorem V.2** (Soundness). Suppose that each assumed axiom (e.g., about the assertions $\{a\}(u_b, u_e, i, x) \varphi$ and $\{u_b, u_e, i\} \varphi$) is sound. Then for every $T$,

1. $\Sigma; \Gamma \vdash \varphi$ implies $T \vdash (\Sigma; \Gamma \vdash \varphi)$.
2. $\Sigma; \Gamma \vdash \Delta \vdash \mu$ implies $T \vdash (\Sigma; \Gamma \vdash \Delta \vdash \mu)$.

The proof of soundness proceeds by a lexicographic induction, first on the maximum number of (RES) rules in any path in the given derivation, and then on the depth of the derivation. A simpler, more obvious induction on the depth of the derivation does not work because in the (RES) rule the proof that the program $e$ being executed by the thread $I$ satisfies invariant $\varphi$ may be arbitrarily deeper than the proofs of the premises. Another technical difficulty arises due to the possibility of recursive functions: for the rules (PF) and (IF) of Figure 3, we must subinduction on the number $n$ in the definition of $T, n \vdash \mu$. Appendix D shows the proof of soundness for these critical cases.

**VI. RELY-GUARANTEE REASONING**

Often, a security property relies on specific behavior of threads that can be ascertained only if the security property itself holds. For instance, in the HRU model of access control [34], where an access control matrix prevents its own modification, the property that a certain access is not allowed may rely on the matrix preventing untrusted threads from creating a corresponding entry in it. Similarly, the analysis of secrecy of keys in security protocols often relies both on the keys having remained secret in the past and on trusted threads performing only stipulated actions [27].

The rely-guarantee method is a general technique for proving such properties for concurrently executing threads [1, 2, 35]. Summarily, suppose $\varphi$ is a property of state. The rely-guarantee method envisages that in order to show that $\varphi$ holds in all states of the system’s execution, it suffices to prove the following three properties:

(A) $\varphi$ holds initially.

(B) There is a class of properties $\psi(i)$, indexed by threads $i$, such that for any action that $i$ may perform, if $\varphi$ holds in the state preceding the action, then $\psi(i)$ holds immediately after $i$ executes the action.

(C) If $\varphi$ holds in a state and $\psi(i)$ holds in the next state for all $i$, then $\varphi$ holds in the next state.

Here, we show how, for a wide class of properties, the rely-guarantee technique is a special case of the reasoning principles presented in Section IV. Suppose $\varphi(u)$ is a property that we wish to establish for all time points $u$. Assume that there is a set of threads identified by the predicate $\iota(i)$ and a thread-specific property $\psi(u, i)$ such that the following analogues of the properties (A)-(C) hold:

1. $\varphi(\infty)$
2. $\forall i, u. (\iota(i) \land \forall u' < u. \varphi(u')) \supset \psi(u, i)$
3. $(\varphi(u_1) \land \neg\varphi(u_2) \land (u_1 < u_2)) \supset \exists i, u_3. (u_1 < u_3 \leq u_2) \land \iota(i) \land \neg\psi(u_3, i) \land$ $\forall u_4 \in \{u_1, u_3\}, \varphi(u_4)$

Then, we can prove in the proof system of Section IV that $\forall u. \varphi(u)$.

**Theorem VI.1** (Rely-guarantee). Conditions (1)-(3) as above imply $\forall u. \varphi(u)$ in the proof system of Section IV. **Proof:** The proof follows by a straightforward analysis in the temporal logic. See Appendix E for details.

In general, condition (2) is analogous to property (B) and it may be established through invariants. For instance, if $\iota(i) = i \in I$, where $I$ is a set of trusted threads, then by rule (HONTH), condition (2) holds if the following assertion holds for all programs $e$ that threads in $I$ may execute.

$\{e\}(u_b, u_e, i) \forall u \in \{u_b, u_e\}. (\forall u' < u. \varphi(u')) \supset \psi(u, i)$

Condition (3) means that if there is violation of $\varphi$ at time $u_2$ but this was not the case at time $u_3$ then is must be a first violation at time $u_3$ that is caused due to a violation of the thread specific property by some thread satisfying $\iota$. This is stronger than property (C), but holds for state properties that satisfy (C). In practice, condition (3) may be established either as a sound axiom, or through other proof rules.
Analysis of secrecy in the Kerberos V5 protocol: We have used the rely-guarantee method outlined here to prove secrecy of the shared key generated by the authentication server during the Kerberos V5 protocol. Our proof is similar to a prior proof of the same property written in Protocol Composition Logic (PCL) [27]. The PCL proof uses a secrecy induction through a rule called (NET), which is an instance of our rely-guarantee technique if we choose \( \varphi \) to be the PCL predicate \( \text{SafeMsg} \) and \( \psi \) to be the PCL predicate \( \text{SendSafeMsg} \). Consequently, the proof shows that secrecy analysis in PCL is a special case of rely-guarantee reasoning in our framework. Details of the analysis are in Appendix G.

VII. Future Work

This paper makes significant progress towards developing a systematic foundation for compositional system security. We plan to extend this work in several directions. So far, we have considered reasoning principles for first-order programs where code cannot be passed as arguments or returned from expressions. However, many systems rely on passing code either as data or through pointers. To model and to establish security properties of such applications, we propose to extend the formalism with higher-order constructs and develop associated compositional reasoning principles. While this paper has focused on the technical foundations of the theory, we plan to apply this framework to develop a systematic basis for web security, to formalize attacker models for web browsers proposed in the literature [7] and develop new ones, and to build an understanding of relevant security policies, end-to-end security properties, attacks in the wild, and ways to defend and prove web applications secure against these attacks.

Acknowledgments: This work was partially supported by the U.S. Army Research Office contract on Perpetually Available and Secure Information Systems (DAAD19-02-1-0389) to CMUs CyLab, the NSF Science and Technology Center TRUST, and the NSF CyberTrust grant “Realizing Verifiable Security Properties on Untrusted Computing Platforms”. Jason Franklin is supported in part by an NSF Graduate Research Fellowship.

REFERENCES

Our main semantic relations take the form $T \vdash \varphi$ and $T, n \vdash \mu$. The former means that the ground ordinary formula $\varphi$ holds on trace $T$. The $T, n \vdash \mu$ is explained later. We define these relations by induction on $\varphi$ and $\mu$. Semantics for formulas with free variables are a special case of semantics.

**APPENDIX B.**

**Proof System for the Logic of Programs**

The proof system of the logic of programs is summarized in Figures 5–7.

**APPENDIX C.**

**Semantics of Formulas and Assertions**

Our main semantic relations take the form $T \models T'$ and $T, n \models \mu$. The former means that the ground ordinary formula $\varphi$ holds on trace $T$. The $T, n \models \mu$ is explained later. We define these relations by induction on $\varphi$ and $\mu$. Semantics for formulas with free variables are a special case of semantics.

**Figure 4:** Operational semantics

\[
\sigma \triangleright T \iff \sigma' \triangleright T' \\
\text{eval } t \ x \ y \ z \ \text{red-term} \quad \sigma \triangleright I; (x, e) :: K; x \leftarrow \sigma \triangleright I; K; e\{t'/x\} \\
\sigma; I \triangleright a \iff \sigma' \triangleright I \triangleright t \quad \text{eval } t \ x \ y \ z \ \text{red-act} \\
\sigma \triangleright I; (x, e) :: K; \text{act } a \leftarrow \sigma' \triangleright I; K; e\{t'/x\} \\
\sigma \triangleright I; K; \text{let}(e_1, x, e_2) \leftarrow \sigma \triangleright I; (x, e_2) :: K; e_1 \\
\sigma \triangleright I; K; \text{if}(b, e_1, e_2) \leftarrow \sigma \triangleright I; K; e_1 \\
\sigma \triangleright I; K; \text{call}(f, t_2) \leftarrow \sigma \triangleright I; K; e\{t'_2/x\} \\
\sigma \triangleright T_1, \ldots, T_n \rightarrow \sigma' \triangleright T'_1, \ldots, T'_n \\
\sigma \triangleright \cdot \rightarrow \sigma' \triangleright \cdot \\
\sigma \triangleright \cdot \rightarrow \sigma' \triangleright \cdot \\
\sigma \triangleright \cdot \rightarrow \sigma' \triangleright \cdot \]

**APPENDIX A.**

**Operational Semantics**

The operational semantics of the programming language are shown in Figure 4.
for hypotetical judgments, which are explained later in this section.

**Definition C.1 (Subtraces).** Let \( T \) be the trace
\[
\overset{u_0}{C_0} \overset{u_1}{C_1} \ldots \overset{u_n}{C_n}
\]
For any \( k \geq 0 \), we define the truncation of \( T \) to \( k \), written \( \text{trunc}(T, k) \) as the trace which contains only the last \( k + 1 \) configurations of \( T \). More precisely,
- If \( k \geq n \), then \( \text{trunc}(T, k) \) equals \( T \).
- If \( k < n \), then \( \text{trunc}(T, k) \) is the trace:
\[
\overset{u_{n-k}}{C_{n-k}} \overset{u_{n-k+1}}{C_{n-k+1}} \ldots \overset{u_n}{C_n}
\]
Note that \( \text{trunc}(T, k) \) is not a trace in the strictest sense since it does not have the auxiliary information that may be associated with \( T \). It is only a sequence of actions, and may not be used to interpret formulas.

**Semantics of terms and predicates:** Since the language and the logic are parametric in the syntax of terms and predicates, we must make some assumptions about interpretations of these entities. Let \([t]\) denote the semantic interpretation of the term \( t \). Let \( t \equiv t' \) mean that \( t \) and \( t' \) are syntactically equal.

- Every closed term \( t \) evaluates (in the semantics) to a closed term \([t]\). The following are assumed to hold:
  - If \( \text{eval} \ t \ t' \), then \([t]\) \( \equiv \) \([t']\). Note that given a semantic definition of \([t]\), this clause imposes a constraint on \( \text{eval} \).
  - \([t_1]\) \( \equiv \) \([t_2]\), then \([t_1/x]\) \( \equiv \) \([t_2/x]\).
- There is a Boolean-valued function \( V(T, u, p) \) which given a trace \( T \), a time point \( u \), and a ground atomic formula \( p \), states whether \( p \) holds on \( T \) at \( u \) or not. We require the following:
  - If \( \sigma = \tau\), then \( V(T, u, p{r/x}) = V(T, u, p{r/x}) \).

**Semantics of temporal formulas:**

- \( T \models t_1 = t_2 \) iff \([t_1]\) \( \equiv \) \([t_2]\).
- \( T \models u_1 \leq u_2 \) iff \( u_1 \) is less than or equal to \( u_2 \) in the usual order on real numbers.
- \( T \models \varphi \land \psi \) iff \( T \models \varphi \) and \( T \models \psi \).
- \( T \models \varphi \lor \psi \) iff \( T \models \varphi \) or \( T \models \psi \).
- \( T \models \varphi \Rightarrow \psi \) iff \( T \models \varphi \Rightarrow \psi \).
- \( T \models \bot \) always.
- \( T \models \top \) never.
- \( T \models \forall x. \varphi \) iff for every ground term \( t \), \( T \models \varphi \).
- \( T \models \exists x. \varphi \) iff there is a ground term \( t \), \( T \models \varphi \).

**Semantics of assertions:** The semantic relation for assertions has the form \( T, n \models \mu \). \( n \) restricts matching to the subtrace containing the last \( n + 1 \) states of the trace only. This is needed to inductively establish soundness for recursive functions.

- \( T, n \models [\mu] \varphi \) iff whenever in subtrace \( \text{trunc}(T, n) \) thread \( I \) has no effectual reduction in the interval \( \langle u_e, u_e' \rangle \), it is the case that \( T \models \varphi \langle u_e'/u_e \rangle \langle u_e'/u_e \rangle \langle \mu(\tau) \rangle \).
- \( T, n \models [\sigma] \varphi \) iff whenever in subtrace \( \text{trunc}(T, n) \) the active expression \( I \) in time \( u_e' \) is act \( a \), and at time \( u_e' \) this action \( a \) reduces returning term \( t \) to its continuation, it is the case that \( T \models \varphi \langle u_e'/u_e \rangle \langle u_e'/u_e \rangle \langle \mu(\tau) \rangle \langle \tau \rangle \). More precisely, \( T, n \models [\sigma] \varphi \) holds iff \( T \models \varphi \langle u_e'/u_e \rangle \langle u_e'/u_e \rangle \langle \mu(\tau) \rangle \langle \tau \rangle \).
Partial correctness assertions

\[ \Sigma; \Gamma; \Delta \vdash \emptyset \langle u_b, u_e, i \rangle \varphi \]
\[ \Sigma; \Gamma; \Delta \vdash \emptyset \langle u_b, u_e, i, x \rangle \varphi \land (x = t) \]
\[ \Sigma; \Gamma; \Delta \vdash \emptyset \langle a \rangle \langle u_b, u_e, i, x \rangle \varphi \]

\[ \Sigma; \Gamma; \Delta \vdash \emptyset \langle u_b, u_e, i \rangle \varphi \]
\[ \Sigma; \Gamma; \Delta \vdash \emptyset \langle u_b, u_e, i, x \rangle \varphi \]
\[ \Sigma; \Gamma; \Delta \vdash \emptyset \langle u_b, u_e, i, x \rangle \varphi \land (x = t) \]
\[ \Sigma; \Gamma; \Delta \vdash \emptyset \langle a \rangle \langle u_b, u_e, i, x \rangle \varphi \]

PT

PA

PL

PC

PF

Invariance assertions

\[ \Sigma; \Gamma; \Delta \vdash \emptyset \langle u_b, u_e, i \rangle \varphi \]
\[ \Sigma; \Gamma; \Delta \vdash \emptyset \langle u_b, u_e, i, x \rangle \varphi \]
\[ \Sigma; \Gamma; \Delta \vdash \emptyset \langle u_b, u_e, i \rangle \varphi \]
\[ \Sigma; \Gamma; \Delta \vdash \emptyset \langle u_b, u_e, i, x \rangle \varphi \]

IT

IA

IL

II

IC

IF

Trusted and Untrusted Threads

\[ \Sigma; \Gamma; \vdash \emptyset \langle u_b, u_e, i \rangle \varphi \]
\[ \Sigma; \Gamma; \vdash \emptyset \langle u_b, u_e, i \rangle \varphi \]
\[ \Sigma; \Gamma; \vdash \emptyset \langle a \rangle \langle u_b, u_e, i \rangle \varphi \]
\[ \Sigma; \Gamma; \vdash \emptyset \langle u_b, u_e, i \rangle \varphi \]
\[ \Sigma; \Gamma; \vdash \emptyset \langle u_b, u_e, i \rangle \varphi \]
\[ \Sigma; \Gamma; \vdash \emptyset \langle u_b, u_e, i \rangle \varphi \]
\[ \Sigma; \Gamma; \vdash \emptyset \langle u_b, u_e, i \rangle \varphi \]
\[ \Sigma; \Gamma; \vdash \emptyset \langle u_b, u_e, i \rangle \varphi \]

\[ \Sigma; \Gamma; \vdash \emptyset \langle u_b, u_e, i \rangle \varphi \]
\[ \Sigma; \Gamma; \vdash \emptyset \langle u_b, u_e, i \rangle \varphi \]
\[ \Sigma; \Gamma; \vdash \emptyset \langle u_b, u_e, i \rangle \varphi \]
\[ \Sigma; \Gamma; \vdash \emptyset \langle u_b, u_e, i \rangle \varphi \]

\[ \Sigma; \Gamma; \vdash \emptyset \langle u_b, u_e, i \rangle \varphi \]
\[ \Sigma; \Gamma; \vdash \emptyset \langle u_b, u_e, i \rangle \varphi \]
\[ \Sigma; \Gamma; \vdash \emptyset \langle u_b, u_e, i \rangle \varphi \]
\[ \Sigma; \Gamma; \vdash \emptyset \langle u_b, u_e, i \rangle \varphi \]

HONT

RES

Figure 6: Proof System, part 2

pattern matches the subtructe \texttt{trunc}(T, n), and there is no reduction in thread I in the interval \((u_b', u_e')\).

\[ \sigma \triangleright I : (x.e) :: K ; \text{act } a \]
\[ u_e' \rightarrow \sigma' \triangleright I : K ; e\{t/x\} \]

\[ \cdots \]
General rules for assertions

$(\Sigma; \Gamma; \Delta \vdash \mu) \text{ is an axiom}$

$\Sigma, \Sigma' ; \Gamma, \Gamma' ; \Delta, \Delta' \vdash \mu \text{ axM}$

$\Sigma, \Gamma \vdash \varphi \quad \Sigma, \Gamma ; \varphi \vdash \mu' \text{ cutOM}$

$\Sigma, \Gamma \vdash \mu \quad \Sigma, \Gamma, \Delta, \mu \vdash \mu' \text{ hypM}$

$\Sigma ; \Gamma ; \Delta, \mu \vdash \mu \text{ cutMM}$

$\Sigma ; \Gamma ; \Delta \vdash [a] \langle u_b, u_e, i \rangle \varphi \quad \Sigma, u_b, u_e, i ; \Gamma \vdash \varphi' \text{ GE}$

$\Sigma ; \Gamma ; \Delta \vdash [e] \langle u_b, u_e, i \rangle \varphi \quad \Sigma, u_b, u_e, i ; \Gamma \vdash \varphi' \text{ GEP}$

$\Sigma ; \Gamma ; \Delta \vdash \{e\} \langle u_b, u_e, i \rangle \varphi \quad \Sigma, u_b, u_e, i ; \Gamma \vdash \varphi' \text{ GEI}$

$\Sigma ; \Gamma ; \Delta \vdash \{f\} \langle y, u_b, u_e, i \rangle \varphi \quad \Sigma, y, u_b, u_e, i ; \Gamma \vdash \varphi' \text{ GFP}$

$\Sigma ; \Gamma ; \Delta \vdash \{f\} \langle y, u_b, u_e, i \rangle \varphi \quad \Sigma, y, u_b, u_e, i ; \Gamma \vdash \varphi' \text{ GFI}$

Figure 7: Proof System, part 3

- $T, n \models [e] \langle u_b, u_e, i, x \rangle \varphi$ if and only if whenever in thread $I$ in subtrace $\text{trunc}(T, n)$ the active expression at time $u'_b$ is $e$, and at time $u'_e$ this expression returns term $t$ to its continuation, it is the case that $T \models \varphi[u'_b/u_b] \{u'_e/u_e\} \{I/i\} \{t/x\}$. More precisely, $T, n \models [e] \langle u_b, u_e, i, x \rangle \varphi$ holds iff $T \models \varphi[u'_b/u_b] \{u'_e/u_e\} \{I/i\} \{t/x\}$ whenever the following pattern matches the subtrace $\text{trunc}(T, n)$ for some $u' > u'_b$ and eval $t$ $t'$, there is no reduction in thread $I$ in the interval $(u'_b, u'_e)$, and the stack of $I$ has suffix $(x, e') :: K$ in the interval $(u'_b, u'_e)$.

- $T, n \models \{e\} \langle u_b, u_e, i \rangle \varphi$ if and only if whenever in thread $I$ in subtrace $\text{trunc}(T, n)$ the active expression at time $u'_b$ is $e$, and until time $u'_e$ this expression has not returned, it is the case that $T \models \varphi[u'_b/u_b] \{u'_e/u_e\} \{I/i\} \{t/x\}$. More precisely, $T, n \models \{e\} \langle u_b, u_e, i \rangle \varphi$ holds iff $T \models \varphi[u'_b/u_b] \{u'_e/u_e\} \{I/i\}$ whenever the following pattern matches the subtrace $\text{trunc}(T, n)$ for some $u' < u'_b$, there is no reduction in thread $I$ in the interval $(u'_b, u'_e)$, and the stack of $I$ has suffix $K$ in the interval $(u'_b, u'_e)$.
\[ T \models \varphi_{\{t_2/y\}}\{u'_b/u_b\}\{u'_e/u_e\}\{I/i\} \] whenever the following pattern matches the subtrace \( \text{trunc}(T,n) \) for \( \text{eval} t_2 t'_2 \), and \( f(y) \triangleq e \), and in the interval \( (u'_b, u'_e) \) the stack of \( I \) has suffix \( K \).

\[
\vdots \quad \sigma \vdash I; K; \text{call}(f, t_2) \\
\vdash u'_b \\ \vdots
\]

Observation C.2. The following hold:
1) If \( T, n \models []\{u_b, u_e, i\}\varphi \) and \( n' \leq n \), then \( T, n' \models []\{u_b, u_e, i\}\varphi \). Similar properties hold for other prefixes \( [a] \), \( [e] \), \( [f] \), and \( [\} \).
2) It is always the case that \( T, 0 \vdash [f]\{y, u_b, u_e, i, x\}\varphi \) and \( T, 0 \vdash \{f\}\{y, u_b, u_e, i\}\varphi \) because the subtrace \( \text{trunc}(T,0) \) has only one state and therefore cannot contain a reduction; in particular, \( \text{trunc}(T,0) \) cannot contain a call to \( f \).

Semantics of sequents: We lift semantics from formulas to sequents as follows.
- \( T \models (\Sigma; \Gamma \vdash \varphi) \) if for every grounding substitution \( \theta \) with domain \( \Sigma \), \( T \models \Gamma\theta \) implies \( T \models \varphi\theta \).
- \( T \models (\Sigma; \Gamma; \Delta \vdash \mu) \) if for every grounding substitution \( \theta \) with domain \( \Sigma \) and every \( n \), \( T \models \Gamma\theta \) and \( T, n \models \Delta\theta \) imply \( T, n \models \mu\theta \).

APPENDIX D. SOUNDNESS

We prove that our logic of programs is sound with respect to the semantics of Section C if we assume that all axioms are also sound.

Lemma D.1 (Closure under evaluation). \( \text{eval} t t' \) implies \( (T \models \varphi_{\{t/x\}} \iff T \models \varphi_{\{t'/x\}}) \).

Proof: Since \( \text{eval} t t' \), \( [t] \cong [t'] \) by assumption. It therefore suffices to show that \( [t] \cong [t'] \) implies \( (T \models \varphi_{\{t/x\}} \iff T \models \varphi_{\{t'/x\}}) \). This follows by induction on \( \varphi \). For the base case where \( \varphi = p \otimes u \), we appeal to the assumption that \( [t] \cong [t'] \) implies \( V(T, u, p\{t/x\}) = V(T, u, p\{t'/x\}) \).

Next, we need a few definitions and lemmas about \( F \)-confined expressions and \( F \)-limited interfaces. (These terms are defined in Section IV-C.)

Definition D.2 \( (F\text{-invariant}) \). Given an interface \( F \), a formula \( \varphi(u_b, u_e, i) \) containing only the free variables \( u_b, u_e, \) and \( i \) is called an \((F, \Gamma, \Delta)\)-invariant if the following hold for every \( \Sigma \):
- \( \varphi(u_b, u_e, i) \) is compositional (see Section IV-C for a definition of compositional)
- For every \( f \in F \), \( \Sigma; \Gamma; \Delta \models \{f\}\{y, u_b, u_e, i\}\varphi(u_b, u_e, i) \)
- For every \( f \in F \), \( \Sigma; \Gamma; \Delta \models []\{y, u_b, u_e, i\}\varphi(u_b, u_e, i) \)
- For every \( f \in F \), \( \Sigma; \Gamma; \Delta \models [f]\{y, u_b, u_e, i\}\varphi(u_b, u_e, i) \)
- For every \( f \in F \), \( \Sigma; \Gamma; \Delta \models [\text{call}(f, t)]\{y, u_b, u_e, i\}\varphi(u_b, u_e, i) \)

Lemma D.3 (Invariance of \( F \)-confined expressions). Let \( e \) be \( F \)-confined, and let \( \varphi(u_b, u_e, i) \) be an \((F, \Gamma, \Delta)\)-invariant. Then,
1) \( \Sigma; \Gamma; \Delta \models []\{u_b, u_e, i\}\varphi(u_b, u_e, i) \)
2) \( \Sigma; \Gamma; \Delta \models [e]\{u_b, u_e, i\}\varphi(u_b, u_e, i) \)

Proof: We prove (1) by induction on \( e \), i.e., we show that \( \Sigma; \Gamma; \Delta \models [e]\{u_b, u_e, i\}\varphi(u_b, u_e, i) \). We proceed by a case analysis on \( e \), and show some representative cases below.

Case. \( e = t \). To show: \( \Sigma; \Gamma; \Delta \models [t]\{u_b, u_e, i\}\varphi(u_b, u_e, i) \).

We have:
1) \( \Sigma; \Gamma; \Delta \models [t]\{u_b, u_e, i\}\varphi(u_b, u_e, i) \) (Defn. of \( (F, \Gamma, \Delta)\)-invariant)
2) \( \Sigma; \Gamma; \Delta \models [\text{let}(e_1, y, e_2)]\{u_b, u_e, i, x\}\varphi(u_b, u_e, i) \) (Rule PT on 1)
3) \( \Sigma; \Gamma; \Delta \models [\text{let}(e_1, e_2)]\{u_b, u_e, i, x\}\varphi(u_b, u_e, i) \) (Thm. in predicate logic)
4) \( \Sigma; \Gamma; \Delta \models [\text{let}(e_1, y, e_2)]\{u_b, u_e, i, x\}\varphi(u_b, u_e, i) \) (Rule GEP on 2, 3)

Case. \( e = \text{act} a \). This does not apply due to definition of \( F \)-confined.

Case. \( e = \text{call}(f, t) \). To show: \( \Sigma; \Gamma; \Delta \models [\text{call}(f, t)]\{u_b, u_e, i, x\}\varphi(u_b, u_e, i) \).

We define by \( F \)-confined, \( f \in F \). We have:
1) \( \Sigma; \Gamma; \Delta \models [\text{call}(f, t)]\{u_b, u_e, i, x\}\varphi(u_b, u_e, i) \) (Defn. of \( (F, \Gamma, \Delta)\)-invariant)
2) \( \Sigma; \Gamma; \Delta \models [\text{call}(f, t)]\{u_b, u_e, i, x\}\varphi(u_b, u_e, i) \) (Defn. of \( (F, \Gamma, \Delta)\)-invariant)
3) \( \Sigma; \Gamma; \Delta \models [\text{call}(f, t)]\{u_b, u_e, i, x\}\varphi(u_b, u_e, i) \) (Rule PC on 1, 2)
4) Σ, u_b, u_m, u_e, i; Γ, u_b < u_m ≤ u_e, ϕ(u_b, u_m, i) ⪰ ϕ(u_m, u_e, i) (Defn. of (F,Γ,Δ)-invariant)
5) Σ, u_b, u_e, i, x; Γ, ≥ u_m, (u_b < u_m < u_e) ∧ ϕ(u_b, u_e, i) ∧ ϕ(u_m, u_e, i)) ⪰ ϕ(u_e, u_e, i) (Thm. in predicate logic using 4)
6) Σ; Γ; Δ ⊢ \{ call(f, t) \} (u_b, u_e, i, x) ϕ(u_b, u_e, i) (Rule GEP on 3.5)

Next, we prove (2) by induction on e and case analysis of its structure. The proof is similar to that of (1), so we show only the interesting case of e = call(f, t).

Case. e = call(f, t). To show: Σ; Γ; Δ ⊢ {call(f, t)} (u_b, u_e, i) ϕ(u_b, u_e, i)

1) Σ; Γ; Δ ⊢ {call(f, t)} (u_b, u_e, i) ϕ(u_b, u_e, i) (Defn. of (F,Γ,Δ)-invariant)
2) Σ; Γ; Δ ⊢ {call(f, t)} (u_b, u_m, i) ϕ(u_b, u_m, i) (Defn. of (F,Γ,Δ)-invariant)
3) Σ; Γ; Δ ⊢ {call(f, t)} (u_b, u_e, i) ϕ(u_m, u_e, i) (Defn. of (F,Γ,Δ)-invariant)
4) Σ; Γ; Δ ⊢ {call(f, t)} (u_b, u_m, u_e, i) ϕ(u_m, u_e, i) (Defn. of (F,Γ,Δ)-invariant)
5) Σ; Γ; Δ ⊢ {call(f, t)} (u_b, u_e, i) ϕ(u_b, u_e, i) (Rule IC on 1-4)

Lemma D.4 (Invariance of F-limited functions). Let G be a set of F-limited functions and let ϕ(u_b, u_e, i) be an (F,Γ,Δ)-invariant. Then ϕ(u_b, u_e, i) is an (F ∪ Γ, Δ, Δ')-invariant.

Proof: Let G = \{ g_k : k = 1, \ldots, n \}. Define:

\[ μ_k = [g_k](y, u_b, u_e, i) \]
\[ ν_k = [g_k](y, u_b, u_e, i) \]
\[ Δ' = μ_1, ν_1, \ldots, μ_n, ν_n \]

Observe that by definition, and due to weakening and rule hypM, ϕ(u_b, u_e, i) is an (F ∪ Γ, Γ, Δ, Δ')-invariant. By definition of F-limited, it follows that the body e_k of each function in G is (F ∪ G)-confined. Hence by Lemma D.3, we have:

- y; Γ; Δ, Δ' ⊢ [ek](u_b, u_e, i, x) ϕ(u_b, u_e, i) for each k
- y; Γ; Δ, Δ' ⊢ [ek](u_b, u_e, i) ϕ(u_b, u_e, i) for each k

Equivalently, we have

A0. y; Γ; Δ, μ_1, ν_1, \ldots, μ_n, ν_n ⊢ [ek](u_b, u_e, i) ϕ(u_b, u_e, i) for each k
B0. y; Γ; Δ, μ_1, ν_1, \ldots, μ_n, ν_n ⊢ [ek](u_b, u_e, i) ϕ(u_b, u_e, i) for each k

Choosing \[ k = 1 \] in (A0), we get y; Γ; Δ, μ_1, ν_1, \ldots, μ_n, ν_n ⊢ [e1](u_b, u_e, i) ϕ(u_b, u_e, i).

By rule PF, we have: y; Γ; Δ, μ_1, ν_1, \ldots, μ_n, ν_n ⊢ [g1](y, u_b, u_e, i, x) ϕ(u_b, u_e, i) or equivalently that:

\[ Γ; Δ, ν_1, \ldots, μ_n, ν_n ⊢ μ_1 \] using cutMM on this and (A0) and (B0), we get:

A0'. y; Γ; Δ, ν_1, \ldots, μ_n, ν_n ⊢ [ek](u_b, u_e, i) ϕ(u_b, u_e, i) for each k
B0'. y; Γ; Δ, ν_1, \ldots, μ_n, ν_n ⊢ [ek](u_b, u_e, i) ϕ(u_b, u_e, i) for each k

Choosing \[ k = 1 \] in (B0'), we get y; Γ; Δ, ν_1, \ldots, μ_n, ν_n ⊢ [e1](u_b, u_e, i) ϕ(u_b, u_e, i).

By rule IF, we have: y; Γ; Δ, ν_1, \ldots, μ_n, ν_n ⊢ [g1](y, u_b, u_e, i) ϕ(u_b, u_e, i) or equivalently that:

\[ Γ; Δ, ν_1, \ldots, μ_n, ν_n ⊢ μ_1 \] using rule cutMM on this and (A0') and (B0'), we get:

A1. y; Γ; Δ, ν_1, \ldots, μ_n, ν_n ⊢ [ek](u_b, u_e, i, x) ϕ(u_b, u_e, i) for each k
B1. y; Γ; Δ, ν_1, \ldots, μ_n, ν_n ⊢ [ek](u_b, u_e, i) ϕ(u_b, u_e, i) for each k

Repeating the previous two step n − 1 times each, we eventually get:

A. y; Γ; Δ ⊢ [ek](u_b, u_e, i, x) ϕ(u_b, u_e, i) for each k
Bn. y; Γ; Δ ⊢ [ek](u_b, u_e, i) ϕ(u_b, u_e, i) for each k

Weakening, we get:

C. y; Γ; Δ, μ_k ⊢ [ek](u_b, u_e, i, x) ϕ(u_b, u_e, i) for each k
D. y; Γ; Δ, ν_k ⊢ [ek](u_b, u_e, i) ϕ(u_b, u_e, i) for each k

Applying rules PF and IF to (C) and (D) respectively, we get:

1) y; Γ; Δ ⊢ [gk](y, u_b, u_e, i, x) ϕ(u_b, u_e, i) for each k
2) y; Γ; Δ ⊢ [gk](y, u_b, u_e, i) ϕ(u_b, u_e, i) for each k

Further from the definition of (F,Γ,Δ)-invariant we have:

3. y; Γ; Δ ⊢ [ek](u_b, u_e, i) ϕ(u_b, u_e, i)
4. y; Γ; Δ ⊢ [ek](u_b, u_e, i) ϕ(u_b, u_e, i) for each k
5. y; Γ; Δ ⊢ [gk](y, u_b, u_e, i, x) ϕ(u_b, u_e, i) for each f ∈ F
6. y; Γ; Δ ⊢ [gk](y, u_b, u_e, i) ϕ(u_b, u_e, i) for each f ∈ F

(1)-(6) imply that ϕ(u_b, u_e, i) is an (F ∪ Γ, Δ, Δ')-invariant.

This lemma is also important in practice. Suppose an adversary is given an interface F such that ϕ(u_b, u_e, i) is an (F, Γ, Δ) invariant. Suppose the adversary uses the functions F to define a new set of functions G. By construction, G is F-limited. Lemma D.4 tells us that ϕ(u_b, u_e, i) is an (F ∪ Γ, Δ, Δ')-invariant. Hence the adversary cannot break system invariants simply by creating new functions from existing ones. Although intuitive, formally establishing this property is non-trivial as the proofs above show.

Theorem D.5 (Soundness). Suppose that each assumed axiom (e.g., about the assertions \[ a(u_b, u_e, i, x) \] ϕ and \[ a(u_b, u_e, i) \] ϕ) is sound. Then for every T,

1. y; Γ ⊢ ϕ implies T ⊢ y; Γ ⊢ ϕ
2. y; Γ ⊢ μ implies T ⊢ y; Γ ⊢ μ

In general, the proof of soundness proceeds by induction on the depth of the given derivations in the proof system. The proof is complicated due to two reasons:
In order to correctly account for recursion, we have to subinduct on the number of steps in a trace when proving soundness of rules (IF) and (PF).

For the rule (RES), we use Lemmas D.2 and D.4, but these Lemmas do not set a bound on the depth of the derivations they produce. Consequently, if the proof is only by induction on the depth of derivations, then we cannot apply the i.h. to derivations obtained from Lemmas D.2 and D.4. The observation that lets us proceed is that the derivations constructed in these Lemmas do not use the rule (RES). As a result, if we induct lexicographically, first on the maximum number of (RES) rules in any path of the derivation, and then on the derivation’s depth, then the induction succeeds.

**Definition D.6** (RES-depth). Define the RES-depth of a derivation as the maximum number of (RES) rules on any path in the derivation starting at its root and ending at a leaf.

**Proof of soundness.** We prove (1) and (2) simultaneously by a lexicographic induction, first on the RES-depth of the given derivation, and then on its depth. We show the interesting cases of the rules (PF) and (RES) here.

**Case.**

\[
\begin{align*}
\sigma & \triangleright I; (x.e') :: K; \text{call}(f, t_2) \\
\longrightarrow & \sigma \triangleright I; (x.e') :: K; e\{t_2/z\} \\
\vdots \\
\end{align*}
\]

It now suffices to show that \( T \models \varphi\{t_2/z\}\{u_b'/u_b\}\{u_c'/u_c\}\{I/i\}\{t/x\} \). Consider the subtrace of \( T \) that starts at \( u_b' \). Observe that this trace is strictly shorter than \( \text{trunc}(T, k' + 1) \) since it lacks at least the state containing \( \sigma \triangleright I; (x.e') :: K; \text{call}(f, t_2) \) in \( I \). So suppose that this subtrace is \( \text{trunc}(T, k'') \), where \( k'' \leq k' \). Now by our assumption we have \( T, n \models \Delta \theta \), and hence Observation C.2(1) implies \( T, k'' \models \Delta \theta \). By the subinduction hypothesis, we also have \( T, k'' \models [f](u_b, u_c, i, x)\varphi \theta \). Combining, we have:

\[ T, k'' \models (\Delta, [f])(y, u_b, u_c, i, x)\varphi \theta \]

Since we also assumed that \( T \models \Gamma \theta \), induction hypothesis on the premise of the rule PF implies that:

\[ T, k'' \models [\varphi\theta\{y/z\}\{t_0/y\}\{u_b, u_c, i, x\}\varphi \theta\{t_0/y\}] \] 

for any \( t_0 \).

Simplifying and choosing \( t_0 = t_2' \), we get \( T, k'' \models [\varphi\theta\{t_2'/z\}\{u_b, u_c, i, x\}\varphi \theta\{t_2'/y\}] \). Since \( \varphi \) is the body of \( f \), it may contain only one free variable \( z \), and therefore \( \varphi \theta = \varphi \), which gives us \( T, k'' \models [\varphi\theta\{t_2'/z\}\{u_b, u_c, i, x\}\varphi \theta\{t_2'/y\}] \).

Now observe that by construction, \( \text{trunc}(T, k'') \) matches the pattern:

\[ 
\end{align*}
\]

By definition of \( T, k'' \models [\varphi\theta\{t_2'/z\}\{u_b, u_c, i, x\}\varphi \theta\{t_2'/y\}] \), we therefore have \( T \models [\varphi\theta\{t_2'/y\}\{u_b'/u_b\}\{u_c'/u_c\}\{I/i\}\{t/x\}] \). Since \text{eval} \( t_2 \) \( t_2' \), Lemma D.1 implies \( T \models \varphi\theta\{t_2'/y\}\{u_b'/u_b\}\{u_c'/u_c\}\{I/i\}\{t/x\} \), as required.

\[ (\varphi(u_b, u_c, i) \text{ compositional}) \]

\[ \forall f \in F, (\cdot; \Gamma; \triangleright \{f\}(y, u_b, u_c, i)\varphi \theta \{t_2'/y\}) \]

\[ \forall f \in F, (\cdot; \Gamma; \triangleright \{f\}(y, u_b, u_c, i)\varphi \theta \{u_b, u_c, i\}) \]

\[ \vdots \]

**Case.**

\[ \Sigma; \Gamma \models \text{Confined}(F) \]

\[ \Sigma; \Gamma; \Gamma' \models \forall u_b. \varphi(\varnothing, u_b, I) \]

Pick a trace \( T \) and a substitution \( \theta \) for \( \Sigma \) such that \( T \models (\Gamma, \Gamma') \theta \). Since \( \Gamma \) must be closed, it follows that \( T \models \Gamma \). It suffices to show that for any ground \( u_c', T \models \varphi(\varnothing, u_c', I) \theta \). From the first
four premises, \( \varphi(u_b, u_c, i) \) is an \((\mathcal{F}, \Gamma, \cdot)-\)invariant, so 
\( \varphi(\omega, u', I\theta) = \varphi(-\infty, u', 10) \). By i.h. on the fifth premise, \( T \models \text{Confined}(I\theta, \mathcal{F}) \). By definition of the latter, it follows that there is a \( \mathcal{F}\)-limited set \( \mathcal{G} \) such that the program of \( I\theta \) in its first state, say \( e \), is \((\mathcal{F} \cup \mathcal{G}, \Gamma, \cdot)-\)invariant. By Lemma D.3, \( \vdash \{ e \} \varphi(u_b, u_e, i) \varphi(u_b, u_e, i) \) and 
\( \vdash \{ e \} \varphi(u_b, u_e, i) \varphi(u_b, u_e, i) \). We distinguish multiple subcases:

Subcase. \( I\theta \) does not exist in \( T \). From the soundness of the fourth premise (via the i.h.), we get 
\( T, \omega \models \{ e \} \varphi(u_b, u_e, i) \varphi(u_b, u_e, i) \), which immediately implies 
\( \varphi(-\infty, u', 10) \), since clearly \( I\theta \) performs no effectual reduction in the interval \((-\infty, u')\).

Subcase. \( I\theta \) first appears in \( T \) at time \( u_b' \) and \( u_e' \leq u_b' \). Again, from the soundness of the fourth premise (via the i.h.), we directly have \( T \models \varphi(-\infty, u', 10) \), since clearly \( I\theta \) performs no effectual reduction in the interval \((-\infty, u')\).

Subcase. \( I\theta \) first appears in \( T \) at time \( u_b' \) and \( u_e' > u_b' \). As reasoned before the case analysis, 
\( \vdash \{ e \} \varphi(u_b, u_e, i) \varphi(u_b, u_e, i) \), where \( e \) is the initial program of thread \( I\theta \). Further, the RES-depth of this derivation is no more than that of premises 1–4, which is one less than that of the whole derivation. Hence by i.h., this sequent is sound, which implies \( T, \omega \models \{ e \} \varphi(u_b, u_e, i) \varphi(u_b, u_e, i) \).\(^3\)

By definition of the semantics of modal formulas, we must have 
\( T \models \varphi(u_b', u_e', 10) \). (Observe that \( e \) cannot return since it must start with an empty stack.) Further, since \( I\theta \) performs no effectual reduction in the interval \((-\infty, u_b')\), and the fourth premise is sound by the i.h., we also get 
\( T \models \varphi(-\infty, u', 10) \). Using the soundness of the first premise (choosing \( u_b = -\infty, u_m = u_b', u_e = u_e', i = 10 \)) we combine 
\( T \models \varphi(-\infty, u', 10) \) and 
\( T \models \varphi(u_b', u_e', 10) \) to obtain 
\( T \models \varphi(-\infty, u', 10) \).

\[ \square \]

**APPENDIX E. PROOF OF THEOREM VI.1**

This Appendix proves Theorem VI.1. Suppose that the conditions (1)–(3) of the statement of the theorem hold.

1. \( \varphi(-\infty) \)
2. \( \forall i, u, (i(i) \land \forall u' < u, \varphi(u')) \supset \psi(u, i) \)
   \( (\varphi(u_1) \land \neg \varphi(u_2) \land (u_1 < u_2)) \supset \psi(u, i) \)
3. \( \exists i, u_3. (u_1 < u_3 \leq u_2) \land (i(i) \land \neg \psi(u_3, i) \land \forall u_4 \in (u_1, u_3), \varphi(u_4)) \)

We wish to show that \( \forall u, \varphi(u) \). We reason by contradiction. Suppose for some \( u, \neg \varphi(u) \). (By (1) and (3) we can deduce that there are \( i \) and \( u_3 \) such that \( (u_3 \leq u) \land (i(i) \land \neg \psi(u_3, i) \land \forall u_4 < u_3, \varphi(u_4)) \). From \( (i(i) \land \forall u_4 < u_3, \varphi(u_4)) \), and (2) we deduce \( \psi(u_3, i) \). This contradicts the previously derived fact \( \neg \psi(u_3, i) \).

**APPENDIX F. PROOF OF INTEGRITY PROPERTY OF EXAMPLE IN SECTION III-B**

In Section III-B, we presented an example inspired by web mashups and informally stated a relevant integrity property for it. In Example III.1, we formalized the property in our temporal logic and, in Section IV-C, we described a high-level outline of its proof. In this appendix we present details of that proof.

First, we codify basic properties of the actions and memory I/O in the example’s model (see Section III-B for details of the actions and memory) as axioms. Selected such axioms are shown in Figure 8. It can be shown easily by induction on traces that the axioms are sound. Hence, by the soundness theorem (Theorem D.5), the proof system of our logic extended with these axioms is sound, so we may use the proof system to prove the property of interest. Further, we also make some example-specific assumptions about trusted and untrusted threads. These are also shown in the same figure.

The property we are trying to prove, as also shown in Example III.1, is that:

\[
\begin{align*}
\text{(Mem(total_box, v) @ u_1 \land} \\
\forall i, u, (i(i) \land \forall u' < u, \varphi(u')) \supset \psi(u, i) \\
(\forall u_1 < u_2, \forall u_3, \forall u_4, (u_4 < u_3 \leq u_2) \land (\forall u_4 < u_3, \varphi(u_4)) \\
\forall u_1 < u_2, \forall u_3, \forall u_4, (\varphi(u_1) \land \neg \varphi(u_2) \land (u_1 < u_2)) \supset \psi(u, i) \\
\exists i, u_3. (u_1 < u_3 \leq u_2) \land (i(i) \land \neg \psi(u_3, i) \land \forall u_4 \in (u_1, u_3), \varphi(u_4))
\end{align*}
\]

We start from the assumption:

\[
\text{(Mem(total_box, v) @ u_1 \land} \\
\forall u_1 < u_2, \forall u_3, \forall u_4, (u_4 < u_3 \leq u_2) \land (\varphi(u_1) \land \neg \varphi(u_2) \land (u_1 < u_2)) \supset \psi(u, i) \\
\exists i, u_3. (u_1 < u_3 \leq u_2) \land (i(i) \land \neg \psi(u_3, i) \land \forall u_4 \in (u_1, u_3), \varphi(u_4))
\]

and seek to show that:

\[
\begin{align*}
\forall u_1 < u_2, \forall u_3, \forall u_4, (u_4 < u_3 \leq u_2) \land (\varphi(u_1) \land \neg \varphi(u_2) \land (u_1 < u_2)) \supset \psi(u, i) \\
\exists i, u_3. (u_1 < u_3 \leq u_2) \land (i(i) \land \neg \psi(u_3, i) \land \forall u_4 \in (u_1, u_3), \varphi(u_4))
\end{align*}
\]

From axiom (Mem) and (1), we get:

\[
\begin{align*}
\exists i, u_w \in (u_1, u_2), \text{Write}(i, \text{total_box}, v) @ u_w
\end{align*}
\]
Model-specific axioms

(Mem) \( (\text{Mem}(c, v) \land \text{Mem}(c, v') \land u' < u) \land (v \neq v') \Rightarrow \exists u_w \in (u, u'], \text{Write}(i, c, v') \land u_w \)

(Receive) \( \text{Recv}(i, j, v) \land u \land v' \land (u' < u) \land \text{Send}(j', i, j, v) \land u' \Rightarrow \)

(New1) \( (\text{New}(i, n) \land \text{Send}(i, j', v) \land u' \land n < v) \Rightarrow u < u' \)

(New2) \( (\text{New}(i, n) \land \text{Send}(i, i', j, v) \land u' \land n < v) \Rightarrow u < u' \)

(New3) \( (\text{New}(i, n) \land \text{New}(i', n) \land u' \Rightarrow (i = i') \land (u = u')) \)

(SendU) \( \text{Send}(i, j, i, j, v) \land u \land \text{Send}(i', j, i, j, v') \land u' \Rightarrow (v = v') \)

Example-specific assumptions

Let \( \mathcal{F} = \{ \text{send_auth, write_acc, read_acc, recv_i, get_i} \} \)

(Untrust) \forall i. \text{Confined}(i, \mathcal{F})

(Trust1) \text{HonestThread}(\text{Agg}, \{ \text{agg_body} \})

(Trust2) \text{HonestThread}(\text{Bank1}, \{ \text{bank1_body} \})

(Trust3) \text{HonestThread}(\text{Bank2}, \{ \text{bank2_body} \})

(Start1) \text{Start}(\text{Agg}) \land -\infty

(Start2) \text{Start}(\text{Bank1}) \land -\infty

(Start3) \text{Start}(\text{Bank2}) \land -\infty

Figure 8: Selected axioms and assumptions for the proof of integrity of the web mashup example

We wish to show that the thread \( i \) in (2) is \( \text{Agg} \). To do this, we exploit the fact that all threads are confined to the interfaces \( \mathcal{F} \) (assumption (Untrust), Figure 8) and the interfaces in \( \mathcal{F} \) only allow \( \text{Agg} \) to write to cell total_box. Formally, we prove the following using the rules of Figure 6.

\[ \forall f \in \mathcal{F}. \{ \text{send_auth, write_acc, read_acc, recv_i, get_i} \} \]

\[ (i < u \land \text{Write}(i, \text{total_box}, v) \land u) \Rightarrow (i = \text{Agg}) \]

(3)

Further, it is easy to see that the invariant \( \varphi(u_b, u_e, i) = \forall v, u. \text{Write}(i, \text{total_box}, v) \land u \land \Rightarrow (i = \text{Agg}) \) is compositional. Hence applying the rule (RES) to (3), (4), and assumption (Untrust), we obtain (for a thread parameter \( i \)):

\[ \forall v, u, (u < u \land \text{Write}(i, \text{total_box}, v) \land u) \Rightarrow (i = \text{Agg}) \]

(4)

Instantiating with \( u_e = \infty \), simplifying, and quantifying the \( i \), we get:

\[ \forall v, u. \text{Write}(i, \text{total_box}, v) \land u \Rightarrow (i = \text{Agg}) \]

(5)

Note that (5) cannot be derived if we allow threads access to the write action directly. Combining (2) and (5) we obtain:

\[ \exists u_w \in (u_1, u_2], \text{Write}(\text{Agg}, \text{total_box}, v) \land u_w \]

(6)

We continue our reasoning by examining the program of \( \text{Agg} \), which, by assumption (Trust1), we already know to be \( \text{agg_body} \). We show the following invariant of the program using the rules of Figure 6. The proof includes a proof of a similar assertion about the function \( \text{agg_loop} \) and also assertions about \( \text{send_auth, write_acc} \) and \( \text{read_acc} \), which we omit here.

\[ \{ \text{agg_body} \} \land \forall v, u. \text{Write}(i, \text{total_box}, v) \land u \Rightarrow (i = \text{Agg}) \]

(7)

Applying the rule (HONTH) to the assumptions (Trust1) and (Start1) and (7), we obtain:

\[ \forall v, u. \text{Write}(i, \text{total_box}, v) \land u \Rightarrow (i = \text{Agg}) \]

(8)

Further, it is easy to see that the invariant \( \varphi(u_b, u_e, i) = \forall v, u. \text{Write}(i, \text{total_box}, v) \land u \land \Rightarrow (i = \text{Agg}) \) is compositional. Hence applying the rule (RES) to (3), (4), and assumption (Untrust), we obtain (for a thread parameter \( i \)):

\[ \forall v, u, \text{Write}(i, \text{total_box}, v) \land u \Rightarrow (i = \text{Agg}) \]

The instantiation \( u_e = \infty \) and some simplification yield:
∀v, u. Write(i, total_box, v) @ u ⊃
∃uru, uup, un, us1, us2, u′
1,u′
2. ∃u pass, nonce, v1, v2, s.
(uru <uup <un <us1 <us2 <u′
1 ... pass′
1,v′
1) @ up1 ∧
(v1,nonce) = (parse_balance(v′
1),nonce′
1)
Combining with (14) we get:
∀v, u. ∀i, j, i'. ((−∞ < u ≤ uc) ∧ Send(i, j, i', v) @ u) ⊃ (i = i')

Instantiating with uc = ∞, simplifying, and quantifying over the parameter i, we get:
∀i, u, j, i'. Send(i, j, i', v) @ u ⊃ (i = i') (12)

Combining with axiom (Receive) yields:
∀i, j, v, u. Recv(i, j, v) @ u ⊃ ∃u'. (u' < u) ∧
Send(j, i, j, v) @ u' (13)

From (9) and (13), we get the following facts in the context of the existential quantifiers in (9):
∀u_w, (u_w < u1 ≤ u2).
∀u_w, u_up, u_n, u_s1, u_s2, u'_1, u'_2. ∀u_pass, nonce, v1, v2, s.
(u_w < u_up < u_n < u_s1 < u_s2 < u'_1 < u'_2 < u_w) ∧
Recv(Agg, User, s) @ u_w ∧
Read(Agg_pass_box, u_pass) @ u_up ∧
Recv(Agg, nonce) @ u_n ∧
Send(Agg, Bank1, Agg, (u_pass, nonce)) @ u_s1 ∧
Send(Agg, Bank2, Agg, (u_pass, nonce)) @ u_s2 ∧
Recv(Agg, Bank1, (v1, nonce)) @ u'_1 ∧
Recv(Agg, Bank2, (v2, nonce)) @ u'_2 ∧
v = v1 + v2
(8)

Next, we wish to reason from the following two lines in (9):
Recv(Agg, Bank1, (v1, nonce)) @ u'_1
Recv(Agg, Bank2, (v2, nonce)) @ u'_2

We wish to show:
∀u_w, (u_w < u1 ≤ u2).
∀u_w, u_up, u_n, u_s1, u_s2, u'_1, u'_2. ∀u_pass, nonce, v1, v2, s.
(u_w < u_up < u_n < u_s1 < u_s2 < u'_1 < u'_2 < u_w) ∧
Recv(Agg, User, s) @ u_w ∧
Read(Agg_pass_box, u_pass) @ u_up ∧
Recv(Agg, nonce) @ u_n ∧
Send(Agg, Bank1, Agg, (u_pass, nonce)) @ u_s1 ∧
Send(Agg, Bank2, Agg, (u_pass, nonce)) @ u_s2 ∧
Recv(Agg, Bank1, (v1, nonce)) @ u'_1 ∧
Recv(Agg, Bank2, (v2, nonce)) @ u'_2 ∧
v = v1 + v2
(9)

Next, we wish to reason from the following two lines in (9):
Recv(Agg, Bank1, (v1, nonce)) @ u'_1
Recv(Agg, Bank2, (v2, nonce)) @ u'_2

We wish to show:
∀u, i, j, i', v. Send(i, j, i', v) @ u ⊃ (i = i')

In order to prove this, we intend to exploit the fact that the only interface which allows sending satisfies this property. Formally, we show the following using the rules of Figure 6.
∀f ∈ F. ((f)∈F(y, u_w, u_e, i)
∀u, i, j, i'. (u_w < u ≤ u_e) ∧ Send(i, j, i', v) @ u) ⊃ (i = i') (10)
∀f ∈ F. ((f)∈F(y, u_w, u_e, i, x)
∀u, i, j, i'. (u_w < u ≤ u_e) ∧ Send(i, j, i', v) @ u) ⊃ (i = i') (11)

Observe that the invariant φ(u, u_e, i) = ∀u, j, i'. ((u_w < u ≤ u_e) ∧ Send(i, j, i', v) @ u) ⊃ (i = i') is compositional. Hence, applying the rule (RES) to (10), (11), and assumption (Untrust), we obtain the following for any thread parameter i:
∀u_w, i, j, i', (u_w < u ≤ u_e) ∧ Send(i, j, i', v) @ u) ⊃ (i = i') (12)

Instantiating with u_e = ∞, simplifying, and quantifying over the parameter i, we get:
∀i, u, j, i'. Send(i, j, i', v) @ u ⊃ (i = i') (13)

Combining with axiom (Receive) yields:
∀i, j, v, u. Recv(i, j, v) @ u ⊃ ∃u'. (u' < u) ∧
Send(j, i, j, v) @ u' (14)
∃u_1, u_1', u_1'' \ (u_1 < u_1 < u_1'' < u_1') \land
∃v_1', nonce_1', u_pass' \land
Recv(Bank1, Agg, (u_pass', nonce', nonce)) @ u_1 \land
Get(Bank1, bank1.com/balance, u_pass', v_1') @ u_1 \land
(v_1, nonce) = (parse_balance(v_1'), nonce')

Simplification using the last line yields:

∃u_1, u_1', u_1'' \ (u_1 < u_1 < u_1'' < u_1') \land
∃v_1', u_pass' \land
Recv(Bank1, Agg, (u_pass', nonce')) @ u_1 \land
Get(Bank1, bank1.com/balance, u_pass', v_1') @ u_1 \land
v_1 = parse_balance(v_1')

A similar property may be derived for Bank2. Combining both with (9), we get:

∃u_w, (u_w < u_1 \leq u_2),
∃u_r, u_up, u_n, u_1, u_2, u_1', u_2', u_r_1, u_r_2, u_p_2.
∀u_pass, nonce, v_1, v_2, s, u_pass_1, u_pass_2, v_1', v_2'.
(u_r < u_p < u_n < u_1 < u_2 < u_1' < u_2' < u_w) \land
(u_1 < u_1 < u_1') \land (u_2 < u_2' < u_2') \land
Recv(Agg, User, s) @ u_n \land
Read(Agg, pass_box, u_pass) @ u_p \land
Send(Agg, Bank1, (u_pass', nonce')) @ u_1 \land
Recv(Agg, Bank1, (v_1, nonce')) @ u_1' \land
Recv(Agg, Bank2, (v_2, nonce')) @ u_2' \land
v = v_1 + v_2' \land
Recv(Bank1, Agg, (u_pass', nonce')) @ u_1 \land
Get(Bank1, bank1.com/balance, u_pass', v_1') @ u_1 \land
v_1 = parse_balance(v_1') \land
Recv(Bank2, Agg, (u_pass_2', nonce')) @ u_2' \land
Get(Bank2, bank2.com/balance, u_pass_2', v_2') @ u_2 \land
v_2 = parse_balance(v_2')

(16)

Simplifying using the lines v_1 = parse_balance(v_1') and v_2 = parse_balance(v_2'), we get:

∃u_w, (u_w < u_1 \leq u_2),
∃u_r, u_up, u_n, u_1, u_2, u_1', u_2', u_r_1, u_r_2, u_p_2.
∀u_pass, nonce, s, u_pass_1, u_pass_2, v_1', v_2'.
(u_r < u_p < u_n < u_1 < u_2 < u_1' < u_2' < u_w) \land
(u_1 < u_1 < u_1') \land (u_2 < u_2' < u_2') \land
Recv(Agg, User, s) @ u_n \land
Read(Agg, pass_box, u_pass) @ u_p \land
New(Agg, nonce) @ u_n \land
Send(Agg, Bank1, Agg, (u_pass, nonce)) @ u_1 \land
Send(Agg, Bank2, Agg, (u_pass, nonce)) @ u_2 \land
Recv(Agg, Bank1, (parse_balance(v_1'), nonce)) @ u_1' \land
Recv(Agg, Bank2, (parse_balance(v_2'), nonce)) @ u_2' \land
v = parse_balance(v_1') + parse_balance(v_2') \land
Recv(Bank1, Agg, (u_pass', nonce')) @ u_1 \land
Get(Bank1, bank1.com/balance, u_pass', v_1') @ u_1 \land
Recv(Bank2, Agg, (u_pass_2', nonce')) @ u_2' \land
Get(Bank2, bank2.com/balance, u_pass_2', v_2') @ u_2

(17)

Next, observe that we have the facts New(Agg, nonce) @ v_n and Recv(Bank1, Agg, (u_pass', nonce')) @ u_1 in the above formula. From axiom (New1), it follows that u_n < u_1. Similarly, we may also derive that u_n < u_r_2.

Incorporating these into (17) (see the line marked * below), we get:

∃u_w, (u_w < u_1 \leq u_2).
∃u_r, u_up, u_n, u_1, u_2, u_1', u_2', u_r_1, u_r_2, u_p_2.
∀u_pass, nonce, s, u_pass_1, u_pass_2, v_1', v_2'.
(u_r < u_p < u_n < u_1 < u_2 < u_1' < u_2' < u_w) \land
(u_1 < u_1 < u_1') \land (u_2 < u_2' < u_2') \land
Recv(Agg, User, s) @ u_n \land
Read(Agg, pass_box, u_pass) @ u_p \land
New(Agg, nonce) @ u_n \land
Send(Agg, Bank1, Agg, (u_pass', nonce')) @ u_1 \land
Send(Agg, Bank2, Agg, (u_pass, nonce)) @ u_2 \land
Recv(Agg, Bank1, (parse_balance(v_1'), nonce)) @ u_1' \land
Recv(Agg, Bank2, (parse_balance(v_2'), nonce)) @ u_2' \land
v = parse_balance(v_1') + parse_balance(v_2') \land
Recv(Bank1, Agg, (u_pass', nonce')) @ u_1 \land
Get(Bank1, bank1.com/balance, u_pass', v_1') @ u_1 \land
Recv(Bank2, Agg, (u_pass_2', nonce')) @ u_2' \land
Get(Bank2, bank2.com/balance, u_pass_2', v_2') @ u_2

(18)

By (13) and the fact Recv(Bank1, Agg, (u_pass', nonce')) @ u_r_1 above, we deduce that:

∃u_r, (u_r < u_1 \land
Send(Agg, Bank1, Agg, (u_pass', nonce')) @ u_1

(19)

Due to axiom (New2), it follows that u_n < u_{r_1}. Incorporating into (19), we get

∃u_w, (u_w < u_{r_1} \land
Send(Agg, Bank1, Agg, (u_pass', nonce')) @ u_{r_1}

(20)

We may derive a similar fact about Bank2. Combining both with (18) we obtain:

∃u_w, (u_w < u_1 \leq u_2).
∃u_r, u_up, u_n, u_1, u_2, u_1', u_2', u_r_1, u_r_2, u_p_2.
∀u_pass, nonce, s, u_pass_1, u_pass_2, v_1', v_2'.
(u_r < u_p < u_n < u_1 < u_2 < u_1' < u_2' < u_w) \land
(u_1 < u_1 < u_1') \land (u_2 < u_2' < u_2') \land
Recv(Agg, User, s) @ u_n \land
Read(Agg, pass_box, u_pass) @ u_p \land
New(Agg, nonce) @ u_n \land
Send(Agg, Bank1, Agg, (u_pass, nonce)) @ u_1 \land
Send(Agg, Bank2, Agg, (u_pass, nonce)) @ u_2 \land
Recv(Agg, Bank1, (parse_balance(v_1'), nonce)) @ u_1' \land
Recv(Agg, Bank2, (parse_balance(v_2'), nonce)) @ u_2' \land
v = parse_balance(v_1') + parse_balance(v_2') \land
Recv(Bank1, Agg, (u_pass', nonce')) @ u_1 \land
Get(Bank1, bank1.com/balance, u_pass', v_1') @ u_1 \land
Recv(Bank2, Agg, (u_pass_2', nonce')) @ u_2' \land
Get(Bank2, bank2.com/balance, u_pass_2', v_2') @ u_2

(21)

Our next objective is to show that u_pass_1 = u_pass. To prove this, we first show that the aggregator never sends out
two messages with the same nonce. Then, since we already have the facts Send(Agg, Bank1, Agg, (\textit{u\_pass}, \textit{nonce})) \oplus \textit{u}'_1 and Send(Agg, Bank1, Agg, (\textit{u\_pass}', nonce)) \oplus \textit{u}'_2, we easily derive \textit{u\_pass}'_1 = \textit{u\_pass} and \textit{u\_pass}'_2 = \textit{u\_pass}. Proving that the aggregator never sends out the same message twice requires some care. Because \textit{agg\_body} is a loop, many invariants will not compose with themselves. We show that:

\newcommand{\agg_body}{\textit{agg\_body}}
\begin{equation}
\begin{aligned}
\forall u, v, i \cdot (u_0 < u \leq u_1) \land \text{Send}(i, j, i', v) \oplus u \supset \\
\exists n, x, u_n \cdot (u_0 < u_n < u) \land \text{New}(i, n) \oplus u_n \land (v = (x, n)) \land \\
\forall x', i', u' \in (u, u), (\text{Send}(i, j, i'', (x', n)) \oplus u')
\end{aligned}
\end{equation}

(22)

Applying the rule (MONTH) to assumptions (Trust1) and (Start1) and (22), we obtain:

\begin{align*}
\forall u, v, i' \cdot & ((-\infty < u < u_0) \land \text{Send}(Agg, j, i', v) \oplus u) \supset \\
\exists n, x, u_n \cdot & (-\infty < u_n < u) \land \text{New}(Agg, n) \oplus u_n \land \\
& (v = (x, n)) \land \\
& \forall x', i'', u'' \in (-\infty, u), (\neg \text{Send}(Agg, j, i'', (x', n)) \oplus u')
\end{align*}

Instantiating with \textit{u}\textsubscript{e} = \infty and simplifying, we get:

\begin{equation}
\begin{aligned}
\forall u, v, i' \cdot & \text{Send}(Agg, j, i', v) \oplus u \supset \\
\exists n, x, u_n \cdot & (-\infty < u_n < u) \land \text{New}(Agg, n) \oplus u_n \land \\
& (v = (x, n)) \land \\
& \forall x', i'', u'' < u, (\neg \text{Send}(Agg, j, i'', (x', n)) \oplus u')
\end{aligned}
\end{equation}

Using this, the fact that time points are a total order, and axiom (New3), we can easily show that:

\begin{equation}
\begin{aligned}
\forall u, u', v, x, x', n, j, i', i''. \\
(\text{New}(Agg, n) \oplus u_n \land \text{Send}(Agg, j, i', (x, n)) \oplus u \land \\
\text{Send}(Agg, j, i'', (x', n)) \oplus u') \supset (u = u')
\end{aligned}
\end{equation}

Using axiom (SendU), we derive:

\begin{equation}
\begin{aligned}
\forall u, u', v, x, x', n, j, i', i''. \\
(\text{New}(Agg, n) \oplus u_n \land \text{Send}(Agg, j, i', (x, n)) \oplus u \land \\
\text{Send}(Agg, j, i'', (x', n)) \oplus u') \supset (x = x')
\end{aligned}
\end{equation}

(23)

From (23), we obtain that in (21), \textit{u\_pass}'_1 = \textit{u\_pass} and \textit{u\_pass}'_2 = \textit{u\_pass}. Using this, we obtain:

\begin{equation}
\begin{aligned}
\exists u_w, (u_0 < u_1 \leq u_2), \\
\exists u_u, u_up, u_p1, u_p2, u_1, u_2, u_{u1}, u_{u2}, u_{p1}, u_{p2}. \\
(\text{Recv}(Agg, User, s) \oplus u_u \land \\
\text{Read}(Agg, pass\_box, u_pass) \oplus u_up \land \\
\text{Get}(Bank1, bank1, balance, u_pass, v') \oplus u_p1 \land \\
\text{Send}(Agg, Bank2, (u_pass, nonce)) \oplus u_{p2} \land \\
\text{Recv}(Bank2, Agg, (u_pass', nonce)) \oplus u'_{p2} \land \\
\text{Get}(Bank2, bank2, balance, u_pass, v'')) \oplus u_{p2}
\end{aligned}
\end{equation}

Eliminating some unnecessary lines, we obtain:

\begin{equation}
\begin{aligned}
\exists u_{u}, u_up, u_{p1}, u_{p2}. \\
\exists u\_pass, nonce, s, v\_1, v\_2. \\
(\text{Recv}(Agg, User, s) \oplus u_u \land \\
\text{Read}(Agg, pass\_box, u_pass) \oplus u_up \land \\
\text{Get}(Bank1, bank1, balance, u_pass, v') \oplus u_p1 \land \\
\text{Get}(Bank2, bank2, balance, u_pass, v'') \oplus u_{p2} \land \\
v = \text{parse\_balance}(v') + \text{parse\_balance}(v'')
\end{aligned}
\end{equation}

Renaming \textit{v\_1} \rightarrow \textit{page\_1} and \textit{v\_2} \rightarrow \textit{page\_2}, we obtain the required property:

\begin{equation}
\begin{aligned}
\exists u_{u}, u_up, u_{p1}, u_{p2}. \\
\exists u\_pass, nonce, s, page\_1, page\_2. \\
(\text{Recv}(Agg, User, s) \oplus u_u \land \\
\text{Read}(Agg, pass\_box, u_pass) \oplus u_up \land \\
\text{Get}(Bank1, bank1, balance, u_pass, page1) \oplus u_{p1} \land \\
\text{Get}(Bank2, bank2, balance, u_pass, page2) \oplus u_{p2} \land \\
v = \text{parse\_balance}(page1) + \text{parse\_balance}(page2)
\end{aligned}
\end{equation}

**APPENDIX G. PROOF OF SECRECY OF AUTHENTICATION SERVER’S KEY IN KERBEROS**

In this appendix we use our encoding of the rely-guarantee approach (Section VI) to prove the secrecy of the key generated by the key authentication server during the Kerberos V5 protocol. Our proof is similar to a prior proof of the same property in Protocol Composition Logic (PCL) [27] and the instantiation of our framework for this example employs the same actions and similar axioms as PCL. Because the latter have already been proved sound in PCL, we do not have to prove their soundness again. Our proof demonstrates that secrecy induction in PCL is a special case of rely-guarantee reasoning in our framework and also shows how PCL’s actions and axioms are an instance of our framework.
Actions, Communication Model and Predicates

For this example, we use the standard communication model for network protocols [27]: communication is based on messages, which may be intercepted and read by anyone (unless explicitly protected by encryption). There is no automatic authentication of senders, so messages have no information about source or destination. The action send $v$ sends the message $v$ on the network, while recv receives a message from the network. Correspondingly, the predicate Send$(i, v) \oplus u$ means that thread $i$ sent message $v$ at time $u$ whereas Recv$(i, v) \oplus u$ means that thread $i$ received $v$ at time $u$. Principals are denoted by terms with subscript $p$. Following PCL, we associate owning principals with each thread and state our secrecy properties in terms of what principals know. The predicate Owner$(i, t_p)$ means that thread $i$ is owned by principal $t_p$. We assume that each thread has a unique owner.

We also use PCL’s symmetric encryption framework. The term $k(t_p, t_p')$ is assumed to be a pre-shared secrecy key known only to the principals $t_p$ and $t_p'$. ENC$(k, v)$ denotes the term obtained by encrypting $v$ with the key $k$. The action enc $k, v$ encrypts $v$ with key $k$ and returns ENC$(k, v)$, while the action dec $v, k$ decrypts $v$ of the form ENC$(k, v')$ with $k$ and returns $v'$. The action new generates a new nonce; the predicate New$(i, n) \oplus u$ means that thread $i$ generates the nonce $n$ at time $u$. We assume that nonces can be used as shared keys. Realistically, this requires that the key be derived from the nonce using an algorithm but, as in PCL, we elide this detail. Our proof requires that principals and timestamps (denoted $t$) be distinguishable from nonces and keys.

We use two actions that are not present in PCL. The action pick picks a term of the appropriate type from the set of terms known to the thread. For instance, pick at type thread will choose a thread that the caller knows is executing. We use this action to allow the client and TGS to decide which other threads to initiate the communication with (see Figure 9). Note that, unlike new, pick does not generate a nonce. Further, it plays no role in our analysis.4

The action flag $t$ flags the term $t$ as being significant to an analysis. The predicate Flag$(i, t) \oplus u$ means that thread $i$ flags term $t$ at time $u$. Operationally, flag $t$ behaves like a no-op. However, it is useful for stating our secrecy property, because it allows us to represent internal variables of the program of the KAS in formulas.

Following the prior PCL proof, we include the predicate Has$(i, v) \oplus u$ which means that thread $i$ has previously (at or before time $u$) seen all components of the term $v$ in clear text. This may happen because each component was either generated by $i$ (through new) or because it received it unencrypted or because it received the component encrypted and had the relevant decryption keys. We also include another predicate from PCL: SafeMsg$(v, s, K)$, which means that each occurrence of the atomic term $s$ in the term $v$ is protected through encryption with a key in the set $K$. For precise, inductive definitions of Has$(i, v)$ and SafeMsg$(v, s, K)$, we refer the reader to the prior work on analysis of Kerberos V5 in PCL [27].

Model of Kerberos V5

The Kerberos V5 protocol consists of four roles – client, KAS (Key Authentication Server), TGS (Ticket Granting Server), and Server. The programs of these four roles are shown in Figure 9. match $v, v'; e$ is an abbreviation for if($v = v', e$, halt), where halt = infloop() for infloop($\triangleleft$) = infloop(). self_prin refers to the principal owning the thread executing the program. Every session of the protocol is initiated by the client, which also plays a central role since all other roles communicate directly with the client only. We omit a description of the protocol, referring the reader to prior work for details [27].

Observe that the only occurrence of the action flag in the four programs is in the expression marked tgs, and it is the argument AKey in this occurrence that we wish to prove secret. For the protocol to work correctly, the principals named C, Kp, Tp, and S must execute the programs client, kas, tgs, and server respectively.

Abbreviations and Definitions

In order to simplify the presentation of the secrecy property and its proof, we introduce abbreviations and definitions for formulas in temporal logic. These are shown in Figure 10. $K$ denotes a set of keys and $S$ denotes a set of principals. Informally, Honest($P_p$) means that each thread owned by $P_p$ executes one of the programs from Figure 9. Owner$(i, S)$ means that the owner of thread $i$ lies in the set $S$. OrigRes$(s, S)$ means that the principal who created the term $s$ lies in the set $S$. KeyRes$(K, S)$ means that each key in the set $K$ is known only to principals in $S$. KORes$(s, K, S)$ combines the previous two predicates, SendsSafeMsg$(i, s, K)$ means that if thread $i$ sends $s$ in a message, then all occurrences of $s$ in that message are protected by keys in $K$. SafeNet$(s, K, u)$ means that prior to time $u$, every threads protects $s$ in all messages it sends using keys in $K$, and SendsOut$(i, s)$ means that thread $i$ sends a message containing $s$, possibly encrypted. The predicates Honest, SendsSafeMsg, and SafeNet have meanings similar to predicates of the same names in PCL. Predicates OrigRes, KeyRes, and KORes are related to the PCL predicates OrigHonest, KeyHonest, and KORes, respectively. However, the PCL analogues require that the thread $i$ in the definitions of OrigRes and KeyRes be honest, not owned in a specific set $S$.  

4In PCL, parties that the client and the TGS communicate with are arguments to the programs of the client and the TGS. Our approach is equivalent.
Abbreviations

- $\phi \land \psi @ u = (\phi @ u) \land (\psi @ u)$
- $\phi \lor \psi @ u = (\phi @ u) \lor (\psi @ u)$
- $\phi @ u = (\phi @ u) \lor (\psi @ u)$
- $\neg \phi @ u = - (\phi @ u)$
- $T @ u = T$
- $\bot @ u = \bot$
- $(\forall x. \phi) @ u = \forall x. (\phi @ u)$
- $(\exists x. \phi) @ u = \exists x. (\phi @ u)$
- $(\phi @ u') @ u = \phi @ u'$

Definitions

- $\text{Honest}(P_s) = \forall i. \text{Owner}(i, P_s) \lor \text{HonestThread}(i, \{\text{client}, \text{kas}, \text{tgs}, \text{server}\})$
- $\text{OwnedIn}(i, S) = \exists p. \text{Owner}(i, p) \land p \in S$
- $\text{OrigRes}(s, S) = \forall i. \text{New}(i, s) \lor \text{OwnedIn}(i, S)$
- $\text{KeyRes}(K, S) = \forall i. (\text{Has}(i, k) \land k \in K) \lor \text{OwnedIn}(i, S)$
- $\text{KORes}(s, K, S) = \text{OrigRes}(s, S) \land \forall u. \text{KeyRes}(K, S) \land u$
- $\text{SendsSafeMsg}(i, s, K) = \text{Send}(i, v) \lor \text{SafeMsg}(v, s, K)$
- $\text{SafeNet}(s, K, u) = \forall i'. (u' \leq u) \lor \text{SendsSafeMsg}(i', s, K) \land u'$
- $\text{HasOnly}(S, s) = \forall i. \text{Has}(i, s) \lor \text{OwnedIn}(i, S)$
- $\text{SendOut}(i, s) = \exists m. \text{Send}(i, m) \land s \in m$

**Figure 10:** Abbreviations and definitions for secrecy analysis in Kerberos V5

**Secrecy Property**

The property we wish to show is that the key $\text{AKey}$ generated by the KAS (see Figure 9) in any session of the protocol becomes known only to the KAS, the client, and the TGS in that session, provided, of course, that the client, KAS, and TGS follow the programs of the protocol (they may run concurrent sessions in other roles of the protocol, but may not run any other programs). Formally, we may represent this property in the temporal logic as follows:

$$\varphi_{sec} = \forall i. \text{akey}(i, k_p, t_p, u) . (\text{Flag}(i, \text{akey}(i, c_p, t_p)) \lor \text{Owned}(i, k_p) \land \text{Honest}(k_p) \land \text{Honest}(c_p)) \lor \forall i'. \text{HasOnly}(k_p, c_p, t_p, i', \text{akey}) @ u'$$

**Axioms**

The axioms needed to prove the above secrecy property are shown in Figure 11. All axioms are adaptations of similar axioms or rules in PCL [26, 27]. (Receive) means that if a thread receives a message then some thread must have sent the message earlier. (New3) means that a nonce $n$ can only be generated once. (KeyR) states that if $I_p$ and $J_p$ are honest principals, then their shared key $k(I_p, J_p)$ is known only to them. (SendN) means that in the beginning of a trace,
or before it, there cannot be any send action (as per our definition of traces, there cannot be any action prior to the first state of a trace).

The axiom (NET) is based on a rule of the same name in PCL. Although our version is seemingly more general than the PCL version, its soundness follows in the same way as that of the PCL rule (NET). In PCL, the (NET) rule facilitates secrecy induction. Similarly, in our framework, the (NET) axiom facilitates the rely-guarantee method for secrecy. The (POS) axiom is a translation of an axiom of the same name in PCL.

**Proof of Secrecy Property**

To prove the secrecy property $\varphi_{sec}$, we choose parameters $I_0$, $AKey_{e0}$, $C_{p0}$, $K_{p0}$, $T_{p0}$, and $U_0$ for the variables $i$, $akey$, $cp$, $kp$, and $tp$, and in the formula $\varphi_{sec}$ that we want to show and want that:

$\forall u'.\text{HasOnly}(\{K_{p0},C_{p0},T_{p0}\}, AKey_{e0})@u'$

Define $S_0 = \{K_{p0},C_{p0},T_{p0}\}$. Then, it suffices to show that

$\varphi_0 = \forall u'.\text{HasOnly}(S_0, AKey_{e0})@u'$

follows from the assumptions

$\Gamma_0 = \text{Flag}(I_0, (AKey_{e0}, C_{p0}, T_{p0}))@U_0 \land \text{Owner}(I_0, K_{p0}) \land \text{Honest}(K_{p0}) \land \text{Honest}(C_{p0}) \land \text{Honest}(T_{p0})$

(24)

We now take $\Gamma_0$ as our set of assumptions. In the sequel, whenever we derive a modal formula $\psi$ or assertion $\mu$, what we really mean is $\Sigma; \Gamma_0 \vdash \psi \lor \Sigma; \Gamma_0 \vdash \mu$ for an appropriate $\Sigma$ that includes at least $I_0$, $AKey_{e0}$, $C_{p0}$, $K_{p0}$, $T_{p0}$, and $U_0$. (What we eventually prove is that $\Sigma; \Gamma_0 \vdash \varphi_0$, from which $\varphi_{sec}$ follows immediately.)

**Preliminary deductions:** Let $K_0 = \{k(T_{p0}, K_{p0}), k(C_{p0}, K_{p0})\}$. Our first step is to prove the following property:

$\forall i, c_p, t_p, k_p, u, akey.
(\text{Flag}(i, (akey, c_p, t_p)))@U \land \text{Owner}(i, k_p) \land \text{Honest}(k_p) \vdash
\exists u'.\text{New}(i, akey, k(t_p, k_p), c(k_p, k_p))@u'$

(25)

This property is equivalent to showing that

$\forall c_p, t_p, u, akey.\text{Flag}(i, (akey, c_p, t_p))@U \vdash
\exists u'.\text{New}(i, akey, k(t_p, k_p), c(k_p, k_p))@u'$

(26)

We prove the above using the rule (HNOTH). We show using the rules of Figure 6 that for $E_0 = \{client, kas, tgs, server\}$, it is the case that:

$\forall u_c \in E_0.\{u_c, u_e, i\}$. Then, it is the case that:

(27)

Choosing $u_e = \infty$ and simplifying, we obtain (26). Thus (26) holds. Hence (25) also holds. We now instantiate (25) by choosing $i = I_0, c_p = C_{p0}, t_p = T_{p0}, k_p = K_{p0}$, and $akey = AKey_{e0}$ to get:

$\forall u'.\text{SendSafeMsg}(i, akey, k(t_p, k_p), c(k_p, k_p))@u'$

(27)
The conditions of the implication in (27) are assumed to hold in $\Gamma_0$ (see (24)). Therefore, it follows that the consequent of the implication, i.e., the following must also hold:

$$\exists u'. \ New(I_0, AKey_{y0}) \land u' \land 
\neg \text{SendOut}(I_0, AKey_{y0}) \circ (u', U_0) \land 
\forall u'. \ SendsSafeMsg(I_0, AKey_{y0}, K_0) \circ u'$$  (28)

Next, we wish to prove that $\text{OrigRes}(AKey_{y0}, S_0)$. To do this, we prove the stronger statement: $\forall u, i. \ New(i, AKey_{y0}) \circ u \supset i = I_0$. To prove the latter, assume that $\New(i, AKey_{y0}) \circ u$. By axiom (New3) and $\New(I_0, AKey_{y0}) \circ u'$ in (28) we obtain $i = I_0$ as required. Hence, $\forall u, i. \ New(i, AKey_{y0}) \circ u \supset i = I_0$. This implies $\text{OrigRes}(AKey_{y0}, S_0)$. (Note that $\text{Owner}(I_0, K_{i0})$ in $\Gamma_0$.)

Further the fact $\forall u. \ KeyRes(K_0, S_0) \circ u$ follows immediately from axiom (KeyR) and the assumptions $\text{Honest}(K_{i0})$, $\text{Honest}(C_{i0})$, and $\text{Honest}(T_{i0})$ in $\Gamma_0$. Combining with $\text{OrigRes}(AKey_{y0}, S_0)$, which was derived earlier, we deduce that:

$$\text{KORes}(AKey_{y0}, K_0, S_0)$$  (29)

**Key Step: Rely-Guarantee:** The key step in our proof is an instance of the rely-guarantee technique of Section VI. Using this technique we establish that $\forall u. \ \text{SafeNet}(AKey_{y0}, K_0, u)$. In order to apply this technique, we instantiate the framework of Section VI by choosing:

$$\varphi(u) = \text{SafeNet}(AKey_{y0}, K_0, u)$$

$$\epsilon(i) = \text{Owner}(i, S_0)$$

$$\psi(u, i) = \text{SendsSafeMsg}(i, AKey_{y0}, K_0) \circ u$$

In order to apply the method of Section VI, we must show that the following hold for $\varphi$, $\epsilon$, and $\psi$ as defined above:

1. $\varphi(-\infty)$
2. $\forall i, u, \ (\epsilon(i) \land \forall u' < u. \ \varphi(u')) \supset \psi(u, i)$
   $$(\varphi(u_1) \land \neg\varphi(u_2) \land (u_1 < u_2)) \supset (3) \ \exists i, u_3. \ (u_1 < u_3 \leq u_2) \land \epsilon(i) \land \neg\psi(u_3, i) \land 
   \forall u_4 \in (u_1, u_3). \ \varphi(u_4)$$

(1) follows immediately from axiom (SendN) and the definition $\varphi(u) = \text{SafeNet}(AKey_{y0}, K_0, u)$. The proof of (2), which is the critical step here, and uses reasoning about programs, is described later. To prove (3), we instantiate the axiom (NET) by choosing $s = AKey_{y0}$, $K = K_0$, and $S = S_0$ to obtain:

$$(\text{KORes}(AKey_{y0}, K_0, S_0) \land \text{SafeNet}(AKey_{y0}, K_0, u_1) \land 
-\text{SafeNet}(AKey_{y0}, K_0, u_2) \land (u_1 < u_2)) \supset 
\exists i, u_3. \ (u_1 < u_3 \leq u_2) \land \text{Owner}(i, S_0) \land 
-\text{SendsSafeMsg}(i, S_0, K_0) \circ u_3 \land 
\forall u_4 \in (u_1, u_3). \ \text{SafeNet}(AKey_{y0}, K_0, u_4)$$

Since we already know $\text{KORes}(AKey_{y0}, K_0, S_0)$ from (29), we may eliminate that condition from the above formula to obtain:

$$(\text{SafeNet}(AKey_{y0}, K_0, u_1) \land \neg\text{SafeNet}(AKey_{y0}, K_0, u_2) \land 
\exists i, u_3. \ (u_1 < u_3 \leq u_2) \land \text{Owner}(i, S_0) \land 
-\text{SendsSafeMsg}(i, S_0, K_0) \circ u_3 \land 
\forall u_4 \in (u_1, u_3). \ \text{SafeNet}(AKey_{y0}, K_0, u_4))$$

This is precisely the statement of (3) above. Hence, using Theorem VI.1 we deduce that $\forall u. \ \varphi(u)$, i.e.,

$$\forall u. \ \text{SafeNet}(AKey_{y0}, K_0, u)$$  (30)

Next, we fix any time parameter $u'_0$, and try to show that $\text{HasOnly}(S_0, AKey_{y0}) \circ u'_0$. Following the definition of $\text{HasOnly}$ assume that for some $i$, $\text{Has}(i, AKey_{y0}) \circ u'_0$. It suffices to show that $\text{Owner}(i, S_0)$. From (30), the assumption $\text{Has}(i, AKey_{y0}) \circ u'_0$, and axiom (POS), we obtain:

$$(\exists u'. \ (u' < u'_0) \land \text{New}(i, AKey_{y0}) \circ u') \lor 
(\exists k. \ (k \in K_{i0}) \land \text{Has}(i, k) \circ u'_0)$$

We case analyze these two disjuncts. If $\exists u'. \ (u' < u'_0) \land \text{New}(i, AKey_{y0}) \circ u'$, then using axiom (New3) and (28) we obtain $i = I_0$. Since $\Gamma_0$ contains the assumption $\text{Owner}(I_0, K_{i0})$ and $K_{i0} \in S_0$, $\text{Owner}(i, S_0)$ follows from definition of $\text{OwnerIn}$.

If $\exists k. \ (k \in K_{i0}) \land \text{Has}(i, k) \circ u'_0$, then from fact (29), i.e., $\text{KORes}(AKey_{y0}, K_0, S_0)$, we immediately obtain $\text{Owner}(i, S_0)$.

Since, in both case analyses we obtain $\text{Owner}(i, S_0)$, it follows that $\text{HasOnly}(S_0, AKey_{y0}) \circ u'_0$ for any $u'_0$. Since $u'_0$ is a parameter, this implies $\forall u'. \ \text{HasOnly}(S_0, AKey_{y0}) \circ u'$, which is the $\varphi_0$ that we wanted to prove. Hence, $\varphi_{sec}$, our secrecy property, holds.

**Proof of Condition (2) of Rely-Guarantee:** It only remains to show that condition (2) in the use of the rely-guarantee method holds. Expanding out the condition using earlier definitions of $\varphi(u), \epsilon(i)$, and $\psi(u, i)$, we must show that:

$$\forall i, u. \ \text{Owner}(i, S_0) \land \forall u' < u. \ \text{SafeNet}(AKey_{y0}, K_0, u')) \supset \text{SendsSafeMsg}(i, AKey_{y0}, K_0) \circ u$$

This formula is equivalent to

$$\forall i. \ \text{Owner}(i, S_0) \land \forall u'. (\forall u' < u. \ \text{SafeNet}(AKey_{y0}, K_0, u')) \supset \text{SendsSafeMsg}(i, AKey_{y0}, K_0) \circ u$$

To prove this formula, pick a parameter $i_0$, and assume $\text{Owner}(i_0, S_0)$. It suffices to show $\forall u. \ (\forall u' < u. \ \text{SafeNet}(AKey_{y0}, K_0, u')) \supset \text{SendsSafeMsg}(i_0, AKey_{y0}, K_0) \circ u$.

The assumption $\text{Owner}(i_0, S_0)$ implies that there is an $i_p$ such that $\text{Owner}(i_p, i_p)$ and $i_p \in S_0$. From the assumptions $\text{Honest}(K_{i0}), \text{Honest}(T_{i0})$, and $\text{Honest}(C_{i0})$ in $\Gamma_0$, we obtain $\text{HonestThread}(i_0, S_0)$. Hence, if we can prove the assertion below, then by rule (HONTH), we
would obtain ∀u. ((∀u′< u. SafeNet(AKey0,K0,u′)) ⊃ SendsSafeMsg(i0, AKey0,K0) @ u) as required.

∀e ∈ E0. ({e}⟨u0,ue,i⟩) ∀u ∈ (u,ue]. (∀u'<u. SafeNet(AKey0,K0,u')) ⊃ SendsSafeMsg(i, AKey0,K0) @ u) (31)

Therefore, we only need show the above assertion for each e ∈ E0. So, we must show:

{client}⟨u0,ue,i⟩
∀u ∈ (u,ue]. (∀u'<u. SafeNet(AKey0,K0,u')) ⊃ SendsSafeMsg(i, AKey0,K0) @ u (32)

{kas}⟨u0,ue,i⟩
∀u ∈ (u,ue]. (∀u'<u. SafeNet(AKey0,K0,u')) ⊃ SendsSafeMsg(i, AKey0,K0) @ u (33)

{tgs}⟨u0,ue,i⟩
∀u ∈ (u,ue]. (∀u'<u. SafeNet(AKey0,K0,u')) ⊃ SendsSafeMsg(i, AKey0,K0) @ u (34)

We prove these formulas one by one.

Proof of assertion (31): We are trying to show that:

{client}⟨u0,ue,i⟩
∀u ∈ (u,ue]. (∀u'<u. SafeNet(AKey0,K0,u')) ⊃ SendsSafeMsg(i, AKey0,K0) @ u (35)

Using the rules of Figure 6, we show that:

{client}⟨u0,ue,i⟩ ∀u ∈ (u,ue]. ∀x. Send(i,x) @ u ⊃ 
(∃p, cp, n1, u, uc, (u < uc < u) ∧ 
New(i, n1) @ u ∧ 
¬Flag(i, (n1, −, −)) ⊗ (u, uc) ∧ 
\( x = (cp, tp, n1) \) ∨ 
(∃gt, cp, aKey, sp, n2, u2, ur, (u < ur < u2 < uc) ∧ 
New(i, n2) @ u2 ∧ 
¬Flag(i, (n2, −, −)) ⊗ (u, uc) ∧ 
Recv(i, gt) @ u2 ∧ 
x = (gt, ENC(aKey, cp), cp, sp, n2) \) ∨ 
(∃st, cp, t, u, sKey, (u < u < u2) ∧ 
Recv(i, st) @ u ∧ 
x = (st, ENC(sKey, (cp, t))) (36)

Now we argue that the temporal formula in the assertion (35) implies that in the assertion (31), which, by rule (GEI), would mean that (31) holds. We use a method of contradiction. Assume that the formula in (35) holds. Further assume that the condition of the implication in (31) holds, but the conclusion does not. So we must have for some u ∈ (u0, uc] that:

∀u'<u. SafeNet(AKey0,K0,u') (37)

¬SendsSafeMsg(i, AKey0,K0) @ u (38)

Expanding the definition of SendsSafeMsg in (37), we further obtain that there is a message x such that:

Send(i, x) @ u ∧ ¬SafeMsg(x, AKey0, K0) (39)

From Send(i, x) @ u in (38) and (35) it follows that one of the disjuncts in (35) must hold. We case analyze the three possible disjuncts and show that, in each case, a contradiction can be derived.

Case. The following disjunct holds:

\( \exists t, cp, n1, u, uc, (u < u_1 < u) \) ∧ 
New(i, n1) @ u ∧ 
¬Flag(i, (n1, −, −)) ⊗ (u, uc) ∧ 
\( x = (cp, tp, n1) \) (40)

The last line of (39) and (38) together yield

¬SafeMsg((cp, tp, n1), AKey0, K0) (41)

By definition of SafeMsg, it follows that at least one of the following three must hold:

¬SafeMsg(cp, AKey0, K0) (42)

¬SafeMsg(tp, AKey0, K0) (43)

However, (41) and (42) cannot hold because cp and tp are principals, and, hence, cannot contain AKey0. Therefore, (43) must be true. This forces n1 = AKey0. From this and (39) we deduce:

New(i, AKey0) ⊢ u (44)

We already know from (28) that there is a u' < U0 such that New(I0, AKey0) ⊢ u'. Using axiom (New3) we deduce:

\( (i = I_0) ∧ (u' = u_1) \) (45)

From (45) and (39) we deduce:

\( (u < u' < u) \) ∧ ¬Flag(I0, (AKey0, −, −)) ⊗ (u, uc) (46)

From (28) we also know that

\( \text{Flag}(I0, (AKey0, C_{p0}, T_{p0})) @ U0 \) (47)

Using (46), (47), and u' < U0 we obtain:

\( U_0 > u_e \) (48)

Since u' < u ≤ u_e, we also get:

\( u' < u < U_0 \) (49)
Hence from (28) we obtain \( \neg \text{SendOut}(I_0, AKey_0) \circ (u', u) \) which implies
\[
\neg \text{SendOut}(I_0, AKey_0) @ u
\]  
(50)

From (38), (39), (45), and \( n_1 = AKey_0 \), we also deduce \( \text{Send}(I_0, (c_p, t_p, AKey_0)) @ u \), which contradicts (50).

**Case.** The following disjunct holds:
\[
\exists tgt, c_p, aKey, s_p, n_2, u_2, u_r, (u_b < u_r < u_2 < u) \land
\neg \text{New}(i, n_2) @ u_2 \land
\neg \text{Flag}(i, (n_2, u_r)) @ (u_b, u_e) \land
x = (tgt, EN(cKey, p), c_p, s_p, n_2)
\]  
(51)

The last line of (51) and (38) together yield
\[
\neg \text{SafeMsg}((tgt, EN(cKey, c_p), c_p, s_p, n_2), AKey_0, K_0)
\]  
(52)

By definition of SafeMsg, if follows that at least one of the following must hold:
\[
\neg \text{SafeMsg}(c_p, AKey_0, K_0)
\]  
(53)
\[
\neg \text{SafeMsg}(s_p, AKey_0, K_0)
\]  
(54)
\[
\neg \text{SafeMsg}(n_2, AKey_0, K_0)
\]  
(55)
\[
\neg \text{SafeMsg}(tgt, AKey_0, K_0)
\]  
(56)

However, (53) and (54) cannot hold because \( c_p \) and \( s_p \) are principals, and, hence, cannot contain \( AKey_0 \). Therefore, either (55) or (56) must be true. If (55) holds, then we derive a contradiction as in the previous case (replacing \( n_1 \) with \( n_2 \) and \( n_1 \) with \( u_2 \)). So it suffices to show that (56) entails a contradiction. So assume that (56) holds. From (51) we know that there is a \( u_r \) such that:
\[
(u_r < u) \land \text{Recv}(i, tgt) @ u_r
\]  
(57)

Using axiom (Receive), we obtain a \( u_s \) and a \( j \) such that:
\[
(u_s < u_r < u) \land \text{Send}(j, tgt) @ u_s
\]  
(58)

From (56) and \( u_s < u \) in above formula, we also deduce that \( \neg \text{SafeMsg}(j, AKey_0, K_0) @ u_s \). So, from (58) we obtain SafeMsg((j, AKey_0, K_0)). This contradicts the assumption (56).

**Case.** The following disjunct holds:
\[
\exists st, c_p, t, u_r, scKey, (u_b < u_r < u) \land
\text{Recv}(i, st) @ u_r \land
x = (st, ENC(scKey, (c_p, t)))
\]  
(59)

The last line of (59) and (38) together yield
\[
\neg \text{SafeMsg}((st, ENC(scKey, (c_p, t))), AKey_0, K_0)
\]  
(60)

By definition of SafeMsg, if follows that at least one of the following must hold:
\[
\neg \text{SafeMsg}(c_p, AKey_0, K_0)
\]  
(61)
\[
\neg \text{SafeMsg}(t, AKey_0, K_0)
\]  
(62)
\[
\neg \text{SafeMsg}(st, AKey_0, K_0)
\]  
(63)

However, (61) and (62) cannot hold because \( c_p \) and \( t \) are a principal and a timestamp, respectively and, hence, cannot contain \( AKey_0 \). Therefore, (63) must be true. In that case, we derive a contradiction as in the previous case (by substituting \( tgt \) with \( st \)).

**Proof of assertion (32):** We are trying to show that:
\[
\{\text{kas}\} (u_b, u_e, i) \forall u \in (u_b, u_e). \{\forall u'. u. \text{SafeNet}(AKey_0, K_0, u')\}
\]
\[
\supset \text{SendsSafeMsg}(i, AKey_0, K_0) @ u
\]

Using the rules of Figure 6, we show that
\[
\{\text{kas}\} (u_b, u_e, i) \forall u \in (u_b, u_e). \forall x. \text{Send}(i, x) @ u \supset
(\exists c_p, t_p, u_1, u_n, u_f, a, k_p, n_1.
(u_b < u_r < u_n < u_f < u \leq u_e) \land \text{Owner}(i, k_p) \land
\text{Recv}(i, (c_p, t_p, n_1)) @ u_e \land
\neg \text{New}(i, a) \land u_n \land
\text{Flag}(i, (a, c_p, t_p)) @ u_f \land
x = (c_p, ENC(k(t_p, k_p), (a, c_p)),
ENC(k(c_p, k_p), (a, n_1, t_p)))
\]  
(64)

Now we argue that the temporal formula in the assertion (64) implies that in the assertion (32), which, by rule (GEI), would mean that (32) holds. We use a method of contradiction. Assume that the formula in (64) holds. Further assume that the condition of the implication in (32) holds, but the conclusion does not. So we must have for some \( u \in (u_b, u_e) \) that:
\[
\forall u'. u. \text{SafeNet}(AKey_0, K_0, u')
\]
\[
\neg \text{SendsSafeMsg}(i, AKey_0, K_0) @ u
\]  
(65)

Expanding the definition of SendsSafeMsg in (66), we further obtain that there is a message \( x \) such that:
\[
\text{Send}(i, x) @ u \land \neg \text{SafeMsg}(x, AKey_0, K_0)
\]  
(67)

From \( \text{Send}(i, x) @ u \) in (67) and (64) it follows that the conclusion of (64), i.e. the following, holds.
\[
(\exists c_p, t_p, u_r, a, k_p, n_1.
(u_b < u_r < u_n < u_f < u \leq u_e) \land \text{Owner}(i, k_p) \land
\text{Recv}(i, (c_p, t_p, n_1)) @ u_e \land
\neg \text{New}(i, a) \land u_n \land
\text{Flag}(i, (a, c_p, t_p)) @ u_f \land
x = (c_p, ENC(k(t_p, k_p), (a, c_p)),
ENC(k(c_p, k_p), (a, n_1, t_p)))
\]  
(68)

The last line of (68) and (67) together yield
By definition of SafeMsg, if follows that at least one of the following must hold:

\[ \neg \text{SafeMsg}(c_p, AKey_0, \mathcal{K}_0) \] (70)

\[ \neg \text{SafeMsg}(t_p, AKey_0, \mathcal{K}_0) \] (71)

\[ \neg \text{SafeMsg}(n_1, AKey_0, \mathcal{K}_0) \] (72)

\[ \neg \text{SafeMsg}(a, AKey_0, \mathcal{K}_0) \] (73)

However, (70) and (71) cannot hold because \( c_p \) and \( t_p \) are principals and, hence, cannot contain \( AKey_0 \). If (72) holds, then we derive a contradiction as in the second case for client earlier (by replacing \( tgt \) with \( (c_p, t_p, n_1) \)). If (73) holds, then we must have:

\[ a = AKey_0 \] (74)

From (68) and (74) we conclude:

\[ \text{New}(i, AKey_0) @ u_n \] (75)

Using axiom (New3), (75), and (28) we obtain:

\[ u_n = u' \land i = I_0 \] (76)

From (28) and (76), we derive that

\[ \text{SendsSafeMsg}(i, AKey_0, \mathcal{K}_0) @ u \] (77)

This contradicts (66).

Proof of assertion (33): We are trying to show that:

\[ \{\text{tgs}\}(u_b, u_e, i) \forall u \in (u_b, u_e]. \forall u'. \text{SafeNet}(AKey_0, \mathcal{K}_0, u') \supset \text{SendsSafeMsg}(i, AKey_0, \mathcal{K}_0) @ u \]

Using the rules of Figure 6, we show that:

\[ \{\text{tgs}\}(u_b, u_e, i) \forall u \in (u_b, u_e]. \forall x. \text{Send}(i, x) @ u \supset \]

\[ (\exists tgt, encct, s_p, c_p, n_2, skey, a, u_n, u, k. (u_b < u_n < u_n < u_n < u_e) \land \text{Recv}(i, (tgt, encct, c_p, s_p, n_2)) @ u_n \land \text{New}(i, skey) @ u_n \land x = (c_p, ENC(k, (skey, c_p)), ENC(a, (skey, n_2, s_p)))) \] (78)

Now we argue that the temporal formula in the assertion (78) implies that in the assertion (33), which, by rule (GEI), would mean that (33) holds. We use a method of contradiction. Assume that the formula in (78) holds. Further assume that the condition of the implication in (33) holds, but the conclusion does not. So we must have for some \( u \in (u_b, u_e] \) that:

\[ \forall u' < u. \text{SafeNet}(AKey_0, \mathcal{K}_0, u') \] (79)

\[ \neg \text{SendsSafeMsg}(i, AKey_0, \mathcal{K}_0) @ u \] (80)

Expanding the definition of SendsSafeMsg in (80), we further obtain that there is a message \( x \) such that:

\[ \text{Send}(i, x) @ u \land \neg \text{SafeMsg}(x, AKey_0, \mathcal{K}_0) \] (81)

From Send\( (i, x) @ u \) in (81) and (78) it follows that the conclusion of (78), i.e., the following, holds.

\[ (\exists tgt, encct, c_p, s_p, n_2, skey, a, u_n, u_r, k. (u_b < u_r < u_n < u_n < u_e) \land \text{Recv}(i, (tgt, encct, c_p, s_p, n_2)) @ u_r \land \text{New}(i, skey) @ u_r \land x = (c_p, ENC(k, (skey, c_p)), ENC(a, (skey, n_2, s_p)))) \] (82)

The last line of (82) and (81) together yield

\[ \neg \text{SafeMsg}(c_p, ENC(k, (skey, c_p)), ENC(a, (skey, n_2, s_p))), AKey_0, \mathcal{K}_0) \] (83)

By definition of SafeMsg, if follows that at least one of the following must hold:

\[ \neg \text{SafeMsg}(c_p, AKey_0, \mathcal{K}_0) \] (84)

\[ \neg \text{SafeMsg}(s_p, AKey_0, \mathcal{K}_0) \] (85)

\[ \neg \text{SafeMsg}(skey, AKey_0, \mathcal{K}_0) \] (86)

\[ \neg \text{SafeMsg}(n_2, AKey_0, \mathcal{K}_0) \] (87)

However, (84) and (85) cannot hold because \( c_p \) and \( s_p \) are principals and, hence, cannot contain \( AKey_0 \). If (86) holds, then we derive a contradiction as in the first case for client (substituting \( skey \) for \( n_1 \)). If (87) holds, then we derive a contradiction as in the second case for client (replacing \( tgt \) with \( (tgt, encct, c_p, s_p, n_2) \)).

Proof of assertion (34): We are trying to show that:

\[ \{\text{server}\}(u_b, u_e, i) \forall u \in (u_b, u_e]. \forall u'. \text{SafeNet}(AKey_0, \mathcal{K}_0, u') \supset \text{SendsSafeMsg}(i, AKey_0, \mathcal{K}_0) @ u \]

Using the rules of Figure 6, we show that:

\[ \{\text{server}\}(u_b, u_e, i) \forall u \in (u_b, u_e]. \forall x. \text{Send}(i, x) @ u \supset \]

\[ (\exists u, wk, (k, x = ENC(k, i)) \supset (\forall u'. \text{SafeNet}(AKey_0, \mathcal{K}_0, u')) \land \text{SendsSafeMsg}(i, AKey_0, \mathcal{K}_0) @ u \] (88)

Now we argue that the temporal formula in the assertion (88) implies that in the assertion (34), which, by rule (GEI), would mean that (34) holds. We use a method of contradiction. Assume that the formula in (88) holds. Further assume that the condition of the implication in (34) holds, but the conclusion does not. So we must have for some \( u \in (u_b, u_e] \) that:

\[ \forall u' < u. \text{SafeNet}(AKey_0, \mathcal{K}_0, u') \] (89)

\[ \neg \text{SendsSafeMsg}(i, AKey_0, \mathcal{K}_0) @ u \] (90)

Expanding the definition of SendsSafeMsg in (90), we further obtain that there is a message \( x \) such that:
\( \text{Send}(i, x) \oplus u \land \neg \text{SafeMsg}(x, AK_{y_0}, K_0) \) \quad (91)

From \( \text{Send}(i, x) \oplus u \) in (91) and (88) it follows that the conclusion of (88), i.e., the following, holds.

\[ \exists t, k. \; x = \text{ENC}(k, t) \] \quad (92)

(92) and (91) together yield

\[ \neg \text{SafeMsg}(\text{ENC}(k, t), AK_{y_0}, K_0) \] \quad (93)

By definition of \( \text{SafeMsg} \), it follows that:

\[ \neg \text{SafeMsg}(t, AK_{y_0}, K_0) \] \quad (94)

This yields a contradiction because \( t \) is a timestamp and cannot contain \( AK_{y_0} \).