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

Deepak Garg
dg@cs.cmu.edu

Jason Franklin
jfrankli@cs.cmu.edu

Dilsun Kaynar
dilsun@cs.cmu.edu

Anupam Datta
danupam@cmu.edu

***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 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. A key insight that enables us to develop such a theory 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. 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 is 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, 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 an example mashup system and present a modular proof of 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 different adversary models. Furthermore, the proof exercises all the compositional reasoning principles developed in this paper. We also demonstrate the generality of our methods by presenting 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 assume-guarantee reasoning. Another orthogonal approach to secure composition is taken in the *universal composability* or *reactive simulatibility* [21, 22] projects. These simulation-based definitions when satisfied can provide strong composition guarantees. However, they are not based on assume-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 certain interfaces. In contrast, prior work lacked the interface abstraction and had a fixed adversary. Also, the language of actions was fixed in prior work to communication actions, cryptographic operations, and certain operations on shared memory, whereas the new programming model and logic is parametric in the actions in the language. 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. 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. 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.

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 a logic of programs that can be used to express security properties of systems modeled in the programming language.

A. Programming Model

A system is modeled in our framework 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, 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 reasoning principles and soundness theorem presented in this paper are general, and apply irrespective of the actions chosen to model a system.

In order to reason about properties of untrusted threads, the programming model limits their interaction with other threads to stipulated sets of interfaces (see Section IV for details). An interface is a function $f(x) \triangleq e$, 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 must not contain expressions. (We intend to develop a higher-order version of our framework in the near future.) To simplify reasoning principles, the language is interpreted functionally. All expressions and actions a 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	::=	$t \mid \texttt{act}\ a \mid \texttt{let}(e_1, x.e_2) \mid \texttt{if}(b, e_1, e_2) \mid \texttt{call}(f, t)$
Function defns		::=	$f(x) \triangleq e$

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

Operational Semantics: The operational semantics of our programming language define how a configuration, which is the collection of all simultaneously executing threads of the system and any possibly shared state, reduces one step at a time. Formally, a configuration $\mathcal{C}$ contains a set of threads $T_1, \ldots, T_n$ and a shared state σ. 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 in the thread. The execution stack records the calling context of the currently executing program as a sequence of frames.

Thread id	I		
Frame	F	::=	$x.e$
Stack	K	::=	$[] \mid F :: K$
Thread	T	::=	$I\,;K\,;e$
Configuration	$\mathcal{C}$	::=	$\sigma \rhd T_1, \ldots, T_n$

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 $\texttt{eval}\ t\ t'$ 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 $\texttt{eval}\ (3+5)\ 8$. The judgment $\sigma\,; I \rhd a \mapsto \sigma'\,; I \rhd t$ means that action a when executed in thread I updates the shared state σ to σ' and returns term t. As an example, in a system with shared memory, σ may be a map from locations l to terms t, the action $\texttt{write}\ l, t$ may change contents of location l to t, and the following judgment may hold: $\sigma\,; I \rhd \texttt{write}\ l, e \mapsto \sigma[l \mapsto t]\,; I \rhd ()$.

Our primary reduction relation, that for configurations, has the form $\mathcal{C} \longrightarrow \mathcal{C}'$. 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: $\sigma \rhd T \hookrightarrow \sigma' \rhd T'$. 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*.

$\boxed{\sigma \rhd T \hookrightarrow \sigma' \rhd T'}$

$$\frac{\sigma\,;I \rhd a \mapsto \sigma'\,;I \rhd t \qquad \texttt{eval}\ t\ t'}{\sigma \rhd I\,;(x.e) :: K\,;\texttt{act}\ a \hookrightarrow \sigma' \rhd I\,;K\,;e\{t'/x\}}\text{red-act}$$

$$\frac{}{\sigma \rhd I\,;K\,;\texttt{let}(e_1, x.e_2) \hookrightarrow \sigma \rhd I\,;(x.e_2) :: K\,;e_1}\text{red-let}$$

$\boxed{\mathcal{C} \longrightarrow \mathcal{C}'}$

$$\frac{\sigma \rhd T \hookrightarrow \sigma' \rhd T'}{\sigma \rhd T, T_1, \ldots, T_n \longrightarrow \sigma' \rhd T', T_1, \ldots, T_n}\text{red-config}$$

Figure 1: Operational semantics, selected rules

Our reasoning principles establish properties of traces. A trace $\mathcal{T}$ is a finite sequence of reductions $\mathcal{C} \longrightarrow \mathcal{C}'$ starting from an initial configuration. We associate a time point u with every reduction on a trace. The time point is a real number. The only constraint on time points associated with a trace is that they be monotonically increasing along the trace. We assume that individual reduction steps happen instantaneously at the time associated with them on the trace. Diagrammatically, we represent a trace as follows, where u_i is the time at which $\mathcal{C}_{i-1}$ reduces to $\mathcal{C}_i$.

$$\xrightarrow{u_0} \mathcal{C}_0 \xrightarrow{u_1} \mathcal{C}_1 \ldots \xrightarrow{u_n} \mathcal{C}_n$$

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. Our simplified system is a web mashup consisting 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 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 $\texttt{send}\ I, J, m$ 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, we

$\boxed{\text{Interfaces}}$

$\texttt{send\_auth}(i, m) \triangleq \texttt{send}\ i, self, m$

$\texttt{write\_acc}(c, v) \triangleq$
  $\texttt{if ACM}(self, l, \texttt{write})$
  $\texttt{then write}\ c, v$
  $\texttt{else}\ ()$

$\texttt{read\_acc}(c) \triangleq$
  $\texttt{if ACM}(self, l, \texttt{read})$
  $\texttt{then read}\ c$
  $\texttt{else}\ ()$

$\boxed{\text{Access control matrix}}$

$\texttt{ACM}(User, total\_box, \texttt{read})$
$\texttt{ACM}(Agg, total\_box, \texttt{write})$
$\texttt{ACM}(User, pass\_box, \texttt{write})$
$\texttt{ACM}(Agg, pass\_box, \texttt{read})$

$\boxed{\text{Known Programs}}$

$\texttt{agg\_loop}() \triangleq$
  $\_ = \texttt{recv}\ User;$
  $u\_pass = \texttt{read\_acc}(pass\_box);$
  $nonce = \texttt{new};$
  $\texttt{send\_auth}(Bank1, (u\_pass, nonce));$
  $\texttt{send\_auth}(Bank2, (u\_pass, nonce));$
  $(v1, nonce') = \texttt{recv}\ Bank1;$
  $(v2, nonce'') = \texttt{recv}\ Bank2;$
  $\texttt{if}\ (nonce = nonce' = nonce'')$
  $\texttt{then write\_acc}(total\_box, v1 + v2)$
  $\texttt{else}\ ();$
  $\texttt{call agg\_loop}()$

$\texttt{bank1\_loop}() \triangleq$
  $(u\_pass, nonce) = \texttt{recv}\ Agg;$
  $page = \texttt{get bank1.com/balance}, u\_pass;$
  $v = \texttt{parse\_balance}(page);$
  $\texttt{send\_auth}(Agg, (v, nonce));$
  $\texttt{bank1\_loop}()$

$\texttt{bank2\_loop}() \triangleq$
  $(u\_pass, nonce) = \texttt{recv}\ Agg;$
  $page = \texttt{get bank2.com/balance}, u\_pass;$
  $v = \texttt{parse\_balance}(page);$
  $\texttt{send\_auth}(Agg, (v, nonce));$
  $\texttt{bank1\_loop}()$

$\texttt{agg\_body} = \texttt{agg\_loop}();$
$\texttt{bank1\_body} = \texttt{bank1\_loop}();$
$\texttt{bank2\_body} = \texttt{bank2\_loop}();$

Figure 2: A Web Mashup Example

allow sender-id spoofing. 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 t to it, respectively. For this example, we need two cells: $pass\_box$ through which the user provides its password to the aggregator, and $total\_box$ through 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 anyone to write a 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 $\texttt{let}(e_1, x.e_2)$ 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 $\texttt{ACM}(I, c, p)$ which is true if thread I is allowed to perform operation $p \in \{\texttt{read}, \texttt{write}\}$ 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 only to these interfaces, but not the `send`, `read`, and `write` actions.

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, totals the values returned by the banks, puts them in $total\_box$ for the user to see, and loops back to the waiting state.

The two banks run similar programs. Each bank 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 her 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, following prior work on temporal logics [31], we represent safety properties of traces as formulas φ of a first-order temporal logic.

$$\begin{array}{lll} \text{Formulas} & \varphi, \psi & ::= \quad p \mathbin{@} u \mid b \mid t = t' \mid u_1 \leq u_2 \mid \\ & & \quad \varphi \wedge \psi \mid \varphi \vee \psi \mid \neg\varphi \mid \top \mid \bot \mid \\ & & \quad \forall x.\varphi \mid \exists x.\varphi \mid \varphi \mathbin{@} u \end{array}$$

The logic includes only one modal formula, $p \mathbin{@} u$, which means that the atomic formula p holds at time u (on the trace against which the formula is being interpreted). $p \mathbin{@} u$ is a hybrid modality [32, 33]. It is known that all operators of linear temporal logic (LTL) can be expressed using $p \mathbin{@} u$ 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 $\texttt{Mem}(l, t)$ that checks contents of memory may be defined by saying that $\mathcal{T} \models \texttt{Mem}(l, t) \mathbin{@} u$ if and only if, in trace $\mathcal{T}$, memory location l contains term t at time u. Similarly, in an analysis of security protocols, $\mathcal{T} \models \texttt{Enc}(I, t, k) \mathbin{@} u$ may hold if and only if thread I in trace $\mathcal{T}$ 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 $\texttt{Mem}(c, v) \mathbin{@} u$ means that cell c contain value v at time u, $\texttt{Recv}(I, J, v) \mathbin{@} u$ means that thread I receives at time u a message sent

previously by thread J, $\texttt{Read}(I, c, v)$ @ u means that thread I reads value v from cell c at time u, and $\texttt{Get}(I, url, p, v)$ @ 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:

$$
\begin{array}{l}
(\texttt{Mem}(total\_box, v') \ @ \ u_1 \wedge \\
\ \texttt{Mem}(total\_box, v) \ @ \ u_2 \wedge \\
\ (u1 < u2) \wedge (v' \neq v)) \supset \\
\ \ \exists u_{ru}, u_{up}, u_{p1}, u_{p2}. \\
\ \ \exists u\_pass, page1, page2, s. \\
\ \ \ (u_{ru} < u_{up} < u_{p1}, u_{p2} < u_2) \wedge \\
\ \ \ \texttt{Recv}(Agg, User, s) \ @ \ u_{ru} \wedge \\
\ \ \ \texttt{Read}(Agg, pass\_box, u\_pass) \ @ \ u_{up} \wedge \\
\ \ \ \texttt{Get}(Bank1, \texttt{bank1.com/balance}, u\_pass, page1) \ @ \ u_{p1} \wedge \\
\ \ \ \texttt{Get}(Bank2, \texttt{bank2.com/balance}, u\_pass, page2) \ @ \ u_{p2} \wedge \\
\ \ \ v = \texttt{parse\_balance}(page1) + \texttt{parse\_balance}(page2)
\end{array}
$$

The conditions of the implication in the above property state that $total\_box$ contains a different value v at time u_2 than it had at time u_1. The antecedent means that, under these conditions, the user must have requested a total at time u_{ru} and subsequently, bank balances must have been fetched from `bank1.com` and `bank2.com`, and that v equals their total. 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. In our logic of programs such properties, called *invariants*, may be represented through a novel construct and proved through induction on syntax of programs.

Specifically, we express pre- and post-conditions as well as invariants using six kinds of assertions, generically denoted μ, ν.

$$
\begin{array}{rl}
\text{Assertions } \mu, \nu ::= & [e]\langle u_b, u_e, i, x\rangle\varphi \mid \{e\}\langle u_b, u_e, i\rangle\varphi \mid \\
& [f]\langle y, u_b, u_e, i, x\rangle\varphi \mid \{f\}\langle y, u_b, u_e, i\rangle\varphi \mid \\
& []\langle u_b, u_e, i\rangle\varphi \mid [a]\langle u_b, u_e, i, x\rangle\varphi
\end{array}
$$

In these assertions, u_b, u_e, i, x, y are bound variables whose scope is φ. (Recall that φ 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.

- $[e]\langle u_b, u_e, i, x\rangle\varphi$: If in a trace $\mathcal{T}$, 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.
- $\{e\}\langle u_b, u_e, i\rangle\varphi$: If in a trace $\mathcal{T}$, the active expression of thread I at time u'_b is e, and this expression does not return to the caller until time u'_e, then $\varphi\{u'_b/u_b\}\{u'_e/u_e\}\{I/i\}$ holds.
- $[f]\langle y, u_b, u_e, i, x\rangle\varphi$: If in a trace $\mathcal{T}$, 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.
- $\{f\}\langle y, u_b, u_e, i\rangle\varphi$: If in a trace $\mathcal{T}$, 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.
- $[]\langle u_b, u_e, i\rangle\varphi$: If in a trace $\mathcal{T}$, thread I does not perform any effectual reduction in the interval $(u'_b, u'_e]$, then $\varphi\{u'_b/u_b\}\{u'_e/u_e\}\{I/i\}$ 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.)
- $[a]\langle u_b, u_e, i, x\rangle\varphi$: If in a trace $\mathcal{T}$, the active expression of thread I at time u'_b is $\texttt{act}\ a$, 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 $[e]\langle u_b, u_e, i, x\rangle\varphi$ and $[f]\langle y, u_b, u_e, i, x\rangle\varphi$ specify the behavior of a program that completes execution and 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 φ using the construct p @ u. For example, consider the function $f(x) \triangleq \texttt{let}((\texttt{act read}\ l), z.\ (\texttt{act write}\ l, z{+}x))$ 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 $[f]\langle y, u_b, u_e, i, x\rangle\ \forall z.\ (\texttt{Mem}(l, z)\ @\ u_b \supset \texttt{Mem}(l, y + z)\ @\ u_e) \wedge x = ()$. 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 proposals like Hoare-Type Theory [34] follow similar conventions.

The assertions $\{e\}\langle u_b, u_e, i\rangle\varphi$ and $\{f\}\langle y, u_b, u_e, i\rangle\varphi$ specify *invariants* of programs – φ 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 control flow.

Whereas Section IV presents a proof system for establishing the four assertions $[e]\langle u_b, u_e, i, x\rangle\varphi$, $\{e\}\langle u_b, u_e, i\rangle\varphi$, $[f]\langle y, u_b, u_e, i, x\rangle\varphi$, and $\{f\}\langle y, u_b, u_e, i\rangle\varphi$, we do not stipulate rules for establishing the remaining two assertions, $[]\langle u_b, u_e, i\rangle\varphi$ and $[a]\langle u_b, u_e, i, x\rangle\varphi$ 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 axioms chosen for establishing these two assertions are sound. Domain specific reasoning can 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 or web-based systems.

Thread-Local Reasoning: A salient feature of our logic of programs is that the specifications of a program, if established, hold irrespective of actions of threads other than the one in which the program executes. This is implicit in the intuitive meanings of the assertions above as well as the formal semantics of the logic (Section V). For example, the meaning of $[e]\langle u_b, u_e, i, x\rangle\varphi$ is that if in a trace $\mathcal{T}$, 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. However, $\varphi\{u'_b/u_b\}\{u'_e/u_e\}\{I/i\}\{t/x\}$ would hold *irrespective of the reductions that other threads may have performed in the interim*. As in prior work [26], this property of the proof system simplifies reasoning significantly since we do not have to reason about reductions of other threads when we wish to prove a property that is specific to a thread (e.g, that the thread does not write a certain location).

However, this approach does apply directly when the property we wish to prove relies on actions of several threads. Such properties include secrecy of keys in network protocols and integrity properties of shared memory. In a significant extension to existing related work, we show in Section VI how thread-local *invariants* can be used to encode rely-guarantee reasoning, which allows modular proofs of such non-local properties.

Example III.2. Among many others, an important invariant that we prove in order to establish the integrity property in Example III.1 is the following.

$$
\begin{array}{l}
\{\texttt{bank1\_body}\}\ \ \langle ub, ue, i\rangle \\
\quad \forall u, j, i', x.\ (ub < u < ue) \supset \\
\quad \texttt{Send}(i, j, i', x)\ @\ u \supset \\
\qquad \exists u_{r1}, u_{p1}.\ (u_{r1} < u_{p1} < u) \wedge \\
\qquad \exists v'_1, nonce'_1, u\_pass'_1. \\
\qquad\quad \texttt{Recv}(i, Agg, (u\_pass'_1, nonce'_1))\ @\ u_{r1} \wedge \\
\qquad\quad \texttt{Get}(i, \texttt{bank1.com/balance}, u\_pass'_1, v'_1)\ @\ u_{p1} \wedge \\
\qquad\quad x = (\texttt{parse\_balance}(v'_1), nonce'_1)
\end{array}
$$

This invariant states that while a thread i executes the program `bank1_body`, if i sends a message x, then x must be a pair containing a balance obtained from `bank1.com/balance` and a nonce obtained from Agg. Because we need this property to hold irrespective of how much `bank1_body` executes, this 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, otherwise 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. Although, given the access control checks on writes in the example, this property may seem trivial, designing sound principles for establishing such properties required a significant amount of effort and turned out to be technically challenging.

Formally, proofs establish one of two hypothetical judgments: $\Sigma; \Gamma \vdash \varphi$ and $\Sigma; \Gamma; \Delta \vdash \mu$. In both judgments Σ is a list of variables that may occur free in the rest of the judgment, Γ is a list of assumed formulas of the temporal logic and Δ contains assumed specifications of functions.

$$
\begin{array}{lll}
\Sigma & ::= & \cdot \mid \Sigma, x \\
\Gamma & ::= & \cdot \mid \Gamma, \varphi \\
\Delta & ::= & \cdot \mid \Delta, [f]\langle y, u_b, u_e, i, x\rangle\varphi \mid \Delta, \{f\}\langle y, u_b, u_e, i\rangle\varphi
\end{array}
$$

A proof establishing either $\Sigma; \Gamma \vdash \varphi$ or $\Sigma; \Gamma; \Delta \vdash \mu$ is parametric in all variables in Σ, i.e., it holds for all ground instances of the variables. The judgment $\Sigma; \Gamma \vdash \varphi$ coincides with the standard hypothetical judgment of first-order classical logic with equality (we treat $p\ @\ u$ as an atomic formula $p(u)$), with additional axioms to make time points a total order. We elide the rules for establishing this judgment and list them in Appendix B.

Assertions manifest in the judgment $\Sigma; \Delta \vdash \mu$ are established by an analysis of the program in μ 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 $\varphi \supset \varphi'$, 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 in Appendix B. As mentioned in Section III-C, the rules of our proof system rely on the abstract judgments $[a]\langle u_b, u_e, i, x\rangle\varphi$ 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 $\texttt{let}(e_1, y.e_2)$. 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 reduction 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 $[]\langle u_b, u_m, i\rangle\varphi_1$, $[e_1]\langle u_m, u'_m, i, y\rangle\varphi_2$, and $[e_2]\langle u'_m, u_e, i, x\rangle\varphi_3$ all hold (as in the premises of the rule), then $\exists y.\exists u_m.\exists u'_m.((u_b < u_m < u'_m < u_e) \wedge \varphi_1 \wedge \varphi_2 \wedge \varphi_3)$ must hold. This justifies the conclusion of the rule. Other rules for establishing partial correctness assertions may be justified in a similar manner.

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 $\texttt{let}(e_1, y.e_2)$ executes. If this expression starts executing in thread i at time u_b 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 φ holds in each of these three cases, then φ is in invariant of $\texttt{let}(e_1, y.e_2)$. 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 Δ 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 $[\texttt{send}\ j, i', m]\langle u_b, u_e, i, x\rangle\texttt{Send}(i, j, i', m)\ @\ u_e \wedge \forall u \in (u_b, u_e).\neg\texttt{Send}(I, J, I', m')\ @\ u$. 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 $\texttt{HonestThread}(I, e)$ if thread I executes program expression e only.[1] Let $\texttt{Start}(I)\ @\ u$ 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 $\texttt{HonestThread}(I, e)$.

$$\frac{\Sigma;\Gamma;\cdot \vdash \{e\}\langle u_b, u_e, i\rangle\varphi(u_b, u_e, i) \quad \Sigma;\Gamma \vdash \texttt{HonestThread}(I, e) \quad \Sigma;\Gamma \vdash \texttt{Start}(I)\ @\ u}{\Sigma;\Gamma \vdash \forall u'.\ (u' > u) \supset \varphi(u, u', I)}\text{HONTH}$$

The justification for this rule is the following: since $\texttt{HonestThread}(I, e)$ and $\texttt{Start}(I)\ @\ u$, 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 $u' > u$.

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, $\texttt{HonestThread}(Bank1, \texttt{bank1\_body})$. Further, we may assume that the thread exists in the trace from the beginning, so $\texttt{Start}(Bank1)\ @\ -\infty$. In Example III.2, we listed an invariant of `bank1_body`. Applying the rule (HONTH) to that invariant we may conclude that:

$$\begin{aligned}&\forall u' > -\infty.\quad \forall u, j, i', x.\ (-\infty < u < u') \supset\\ &\quad \texttt{Send}(Bank1, j, i', x)\ @\ u \supset\\ &\qquad \exists u_{r1}, u_{p1}.\ (u_{r1} < u_{p1} < u) \wedge\\ &\qquad \exists v'_1, nonce'_1, u\_pass'_1.\\ &\qquad\quad \texttt{Recv}(Bank1, Agg, (u\_pass'_1, nonce'_1))\ @\ u_{r1} \wedge\\ &\qquad\quad \texttt{Get}(Bank1, \texttt{bank1.com/balance}, u\_pass'_1, v'_1)\ @\ u_{p1} \wedge\\ &\qquad\quad x = (\texttt{parse\_balance}(v'_1), nonce'_1)\end{aligned}$$

Instantiating with $u' = \infty$ and simplifying, we obtain:

$$\begin{aligned}&\forall u, j, i', x.\texttt{Send}(Bank1, j, i', x)\ @\ u \supset\\ &\quad \exists u_{r1}, u_{p1}.\ (u_{r1} < u_{p1} < u) \wedge\\ &\quad \exists v'_1, nonce'_1, u\_pass'_1.\\ &\qquad \texttt{Recv}(Bank1, Agg, (u\_pass'_1, nonce'_1))\ @\ u_{r1} \wedge\\ &\qquad \texttt{Get}(Bank1, \texttt{bank1.com/balance}, u\_pass'_1, v'_1)\ @\ u_{p1} \wedge\\ &\qquad x = (\texttt{parse\_balance}(v'_1), nonce'_1)\end{aligned}$$

[1] Technically, the syntax of our logic requires us to write the suffix $\ldots\ @\ u$ after each atomic formula. However, $\texttt{HonestThread}(I, e)\ @\ u$ is independent of u, so we elide the suffix $@\ u$ after $\texttt{HonestThread}(I, e)$.

$$\frac{\Sigma;\Gamma;\Delta \vdash [a]\langle u_b,u_e,i,x\rangle\varphi}{\Sigma;\Gamma;\Delta \vdash [\texttt{act}\ a]\langle u_b,u_e,i,x\rangle\varphi}\text{PA} \qquad \frac{\Sigma;\Gamma;\Delta \vdash []\langle u_b,u_e,i\rangle\varphi}{\Sigma;\Gamma;\Delta \vdash \{\texttt{act}\ a\}\langle u_b,u_e,i\rangle\varphi}\text{IA}$$

$$\frac{\Sigma;\Gamma;\Delta \vdash []\langle u_b,u_m,i\rangle\varphi_1 \qquad \Sigma;\Gamma;\Delta \vdash [e_1]\langle u_m,u'_m,i,y\rangle\varphi_2 \qquad \Sigma,y;\Gamma;\Delta \vdash [e_2]\langle u'_m,u_e,i,x\rangle\varphi_3}{\Sigma;\Gamma;\Delta \vdash [\texttt{let}(e_1,y.e_2)]\langle u_b,u_e,i,x\rangle\exists y.\exists u_m.\exists u'_m.((u_b<u_m<u'_m<u_e)\wedge\varphi_1\wedge\varphi_2\wedge\varphi_3)}\text{PL}$$

$$\frac{\begin{array}{c}\Sigma;\Gamma;\Delta \vdash []\langle u_b,u_e,i\rangle\varphi \\ \Sigma;\Gamma;\Delta \vdash []\langle u_b,u_m,i\rangle\psi_1 \qquad \Sigma;\Gamma;\Delta \vdash \{e_1\}\langle u_m,u_e,i\rangle\psi_2 \qquad \Sigma,u_b,u_m,u_e,i;\Gamma,u_b<u_m\le u_e,\psi_1,\psi_2 \vdash \varphi \\ \Sigma;\Gamma;\Delta \vdash []\langle u_b,u_m,i\rangle\psi_3 \qquad \Sigma;\Gamma;\Delta \vdash [e_1]\langle u_m,u'_m,i,y\rangle\psi_4 \\ \Sigma,y;\Gamma;\Delta \vdash \{e_2\}\langle u'_m,u_e,i\rangle\psi_5 \qquad \Sigma,u_b,u_m,u'_m,u_e,i,y;\Gamma,u_b<u_m<u'_m\le u_e,\psi_3,\psi_4,\psi_5 \vdash \varphi\end{array}}{\Sigma;\Gamma;\Delta \vdash \{\texttt{let}(e_1,y.e_2)\}\langle u_b,u_e,i\rangle\varphi}\text{IL}$$

$$\frac{f(z)\triangleq e \qquad \Sigma,y;\Gamma;\Delta,[f]\langle y,u_b,u_e,i,x\rangle\varphi \vdash [e\{y/z\}]\langle u_b,u_e,i,x\rangle\varphi}{\Sigma;\Gamma;\Delta \vdash [f]\langle y,u_b,u_e,i,x\rangle\varphi}\text{PF}$$

$$\frac{f(z)\triangleq e \qquad \Sigma,y;\Gamma;\Delta,\{f\}\langle y,u_b,u_e,i\rangle\varphi \vdash \{e\{y/z\}\}\langle u_b,u_e,i\rangle\varphi}{\Sigma;\Gamma;\Delta \vdash \{f\}\langle y,u_b,u_e,i\rangle\varphi}\text{IF}$$

Figure 3: Selected modular rules for establishing program specifications

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 the 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 $\mathcal{F}$, as a set of functions. In general, an adversary that is confined to $\mathcal{F}$ may construct a new set of functions $\mathcal{G}$ that call functions of $\mathcal{F}$ and themselves and combine calls to functions of $\mathcal{F}$ and $\mathcal{G}$ in any way it chooses. To formally represent such an adversary, we need a few definitions.

Definition IV.3 ($\mathcal{F}$-confined expressions)**.** Given an interface $\mathcal{F}$, we call an expression e $\mathcal{F}$-confined if the following hold: (a) All occurrences of `call` in e have the form $\texttt{call}(f,t)$, where $f \in \mathcal{F}$, and (b) `act` does not occur in e.

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

Definition IV.5 ($\mathcal{F}$-confined thread)**.** A (untrusted) thread I is said to be $\mathcal{F}$-confined if I executes a program e and there is a $\mathcal{F}$-limited interface $\mathcal{G}$ such that e is $(\mathcal{F}\cup\mathcal{G})$-confined. The predicate $\texttt{Confined}(I,\mathcal{F})$ holds iff I is $\mathcal{F}$-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 $\forall u_b,u_m,u_e,i.\ ((u_b<u_m\le u_e)\wedge\varphi(u_b,u_m,i)\wedge\varphi(u_m,u_e,i))\supset\varphi(u_b,u_e,i)$.

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.

$$\frac{\begin{array}{c}(\varphi(u_b,u_e,i)\ \text{compositional}) \qquad \cdot;\Gamma;\cdot \vdash []\langle u_b,u_e,i\rangle\varphi(u_b,u_e,i) \\ \forall f\in\mathcal{F}.\ (\cdot;\Gamma;\cdot \vdash \{f\}\langle y,u_b,u_e,i\rangle\varphi(u_b,u_e,i)) \\ \forall f\in\mathcal{F}.\ (\cdot;\Gamma;\cdot \vdash [f]\langle y,u_b,u_e,i,x\rangle\varphi(u_b,u_e,i)) \\ \Sigma;\Gamma \vdash \texttt{Confined}(I,\mathcal{F})\end{array}}{\Sigma;\Gamma,\Gamma' \vdash \forall u_e.\ \varphi(-\infty,u_e,I)}\text{RES}$$

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.

[2]The restriction that the untrusted thread may not execute actions may seem to limit the adversary's abilities but this is not the case because we may give the thread access to interfaces that execute the desired actions immediately. For instance, to allow an adversary access to the `write` action, we may give it the interface $f(x) \triangleq \texttt{write}\ x$.

Since φ is a partial correctness assertion of all functions in $\mathcal{F}$ (fourth premise) and administrative reductions (second premise), as well an invariant of all functions in $\mathcal{F}$, it must hold over all these splits. Therefore, due to the compositionality of φ (first premise), $\varphi(-\infty, u_e, I)$ must hold. The formal justification of this rule is a part of the soundness theorem (Section V) and is non-trivial because we must consider all possible programs that the thread may execute.

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

$$\forall i, v, u.\ Write(i, total\_box, v) \mathbin{@} u \supset i = Agg$$

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 `send_auth`, `write_acc`, `read_acc` and the following two interfaces that mimic the `recv` and `get` actions, then this property can be established using the rule (RES).

$$\texttt{recv\_i}(x) \triangleq \texttt{recv}\ x \qquad \texttt{get\_i}(x) \triangleq \texttt{get}\ x$$

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

$$\begin{array}{l}\forall f \in \mathcal{F}.\ (\{f\}\langle y, u_b, u_e, i\rangle \\ \quad \forall v, u.((u_b < u \leq u_e) \wedge Write(i, total\_box, v) \mathbin{@} u) \\ \qquad \supset i = Agg)\end{array}$$

$$\begin{array}{l}\forall f \in \mathcal{F}.\ ([f]\langle y, u_b, u_e, i, x\rangle \\ \quad \forall v, u.((u_b < u \leq u_e) \wedge Write(i, total\_box, v) \mathbin{@} u) \\ \qquad \supset i = Agg)\end{array}$$

$$\begin{array}{l}[]\langle u_b, u_e, i\rangle \\ \quad \forall v, u.((u_b < u \leq u_e) \wedge Write(i, total\_box, v) \mathbin{@} u) \\ \qquad \supset i = Agg\end{array}$$

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

$$\begin{array}{l}\forall i. \forall u_e. \forall v, u.((-\infty < u \leq u_e) \wedge Write(i, total\_box, v) \mathbin{@} u) \\ \qquad \supset i = Agg\end{array}$$

Choosing $u_e = \infty$ and simplifying, we obtain the required property $\forall i, v, u.\ Write(i, total\_box, v) \mathbin{@} u \supset i = Agg$. 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 `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 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 `send` action, namely `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 φ and assertions μ 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 $\doteq$ denote equality in the domain. For interpreting atomic formulas, we assume the existence of a Boolean valued function $V(\mathcal{T}, u, p)$ ($\mathcal{T}$ is a trace, u is a ground time point, and p is a ground atomic formula) such that $[\![t]\!] \doteq [\![t']\!]$ implies $V(\mathcal{T}, u, p\{t/x\}) = V(\mathcal{T}, u, p\{t'/x\})$. Given these assumptions, we may define the semantics $\mathcal{T} \models \varphi$ of ground temporal formulas φ 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 $\mathcal{T}$ be the trace

$$\xrightarrow{u_0} \mathcal{C}_0 \xrightarrow{u_1} \mathcal{C}_1 \ldots \xrightarrow{u_n} \mathcal{C}_n$$

For any $k \geq 0$, we define the truncation of $\mathcal{T}$ to k, written $\texttt{trunc}(\mathcal{T}, k)$ as the trace which contains only the last $k+1$ configurations of $\mathcal{T}$. If $k > n$ then $\texttt{trunc}(\mathcal{T}, k) = \mathcal{T}$.

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

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

$$\cdots \xrightarrow{u'} \sigma \rhd I\,;K\,;e$$

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

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

Soundness: Assuming that the axioms chosen to reason about the assertions $[a]\langle u_b, u_e, i, x\rangle\varphi$ and $[]\langle u_b, u_e, i\rangle$ 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 $[a]\langle u_b, u_e, i, x\rangle\varphi$ and $[]\langle u_b, u_e, i\rangle\varphi$) is sound. Then for every $\mathcal{T}$,*

1) $\Sigma; \Gamma \vdash \varphi$ *implies* $\mathcal{T} \models (\Sigma; \Gamma \vdash \varphi)$.
2) $\Sigma; \Gamma; \Delta \vdash \mu$ *implies* $\mathcal{T} \models (\Sigma; \Gamma; \Delta \vdash \mu)$.

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 φ 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 subinduct on the number n in the definition of $\mathcal{T}, n \models \mu$. Appendix D shows how the proof proceeds in 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 [35], 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. 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, 36]. Summarily, suppose φ is a property of state. The rely-guarantee method envisages that in order to show that φ holds in all states of the system's execution, it suffices to prove the following three properties:

(A) φ holds initially.

(B) For each thread i, there is property $\psi(i)$ such that for any action that i may perform, if φ holds in the state preceding the action, then $\psi(i)$ holds immediately after i executes the action.

(C) If φ holds in a state and $\psi(i)$ holds in the next state for all i, then φ 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) $\forall i, u.\ (\iota(i) \wedge \forall u' < u.\ \varphi(u')) \supset \psi(u, i)$
$(\varphi(u_1) \wedge \neg\varphi(u_2) \wedge (u_1 < u_2)) \supset$

(3) $\exists i, u_3.\ (u_1 < u_3 \leq u_2) \wedge \iota(i) \wedge \neg\psi(u_3, i) \wedge$
$\forall u_4 \in (u_1, u_3).\ \varphi(u_4)$

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 $\iota(i) = i \in \mathcal{I}$, where $\mathcal{I}$ 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 $\mathcal{I}$ may execute.

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

Condition (3) means that if there is violation of φ at time u_2 but this was not the case at time u_1 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 ι. 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 φ to be the PCL predicate `SafeNet` and ψ to be the PCL predicate `SendsSafeMsg`. 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

[1] J. Misra and K. M. Chandy, "Proofs of networks of processes," *IEEE Transactions on Software Engineering*, vol. 7, no. 4, pp. 417–426, 1981.

[2] C. B. Jones, "Tentative steps toward a development method for interfering programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 5, no. 4, pp. 596–619, 1983.

[3] S. Bellovin, "Security challenges," in *1st ITI Workshop on Dependability and Security*, 2004, panel: Grand challenges and open questions in trusted systems.

[4] J. C. Mitchell, "Programming language methods in computer security," in *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2001, pp. 1–3.

[5] J. M. Wing, "A call to action: Look beyond the horizon," *IEEE Security & Privacy*, vol. 1, no. 6, pp. 62–67, 2003.

[6] A. Barth, C. Jackson, and J. C. Mitchell, "Robust defenses for cross-site request forgery," in *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS 2008)*, 2008, pp. 75–88.

[7] ——, "Securing frame communication in browsers," in *Proceedings of the 17th USENIX Security Symposium*, 2008, pp. 17–30.

[8] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh, "Protecting browsers from DNS rebinding attacks," 2007, pp. 421–431.

[9] S. Chen, Z. Mao, Y.-M. Wang, and M. Zhang, "Pretty-bad-proxy: An overlooked adversary in browsers' HTTPS deployments," in *Proceedings of the 30th IEEE Symposium on Security and Privacy*, 2009, pp. 347–359.

[10] C. Jackson and A. Barth, "Forcehttps: protecting high-security web sites from network attacks," in *Proceedings of the 17th International Conference on World Wide Web (WWW)*, 2008, pp. 525–534.

[11] C. Meadows and D. Pavlovic, "Deriving, attacking and defending the gdoi protocol," in *Proceedings of the 9th European Symposium on Research in Computer Security (ESORICS)*, 2004, pp. 53–72.

[12] N. Asokan, V. Niemi, and K. Nyberg, "Man-in-the-middle in tunnelled authentication protocols," in *Security Protocols Workshop*, 2003, pp. 28–41.

[13] J. C. Mitchell, V. Shmatikov, and U. Stern, "Finite-state analysis of ssl 3.0," in *SSYM'98: Proceedings of the 7th conference on USENIX Security Symposium*, 1998, pp. 16–16.

[14] D. Kuhlman, R. Moriarty, T. Braskich, S. Emeott, and M. Tripunitara, "A correctness proof of a mesh security architecture," in *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF)*, 2008, pp. 315–330.

[15] D. Tsafrir, T. Hertz, D. Wagner, and D. Da Silva, "Portably solving file tocttou races with hardness amplification," in *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, 2008, pp. 1–18.

[16] X. Cai, Y. Gui, and R. Johnson, "Exploiting Unix file-system races via algorithmic complexity attacks," in *SP '09: Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, 2009, pp. 27–41.

[17] D. McCullough, "A hookup theorem for multilevel security," *IEEE Transactions on Software Engineering*, vol. 16, no. 6, pp. 563–568, 1990.

[18] H. Mantel, "On the composition of secure systems," in *SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2002, pp. 88–101.

[19] J. McLean, "Security models and information flow," in *IEEE Symposium on Security and Privacy*, 1990, pp. 180–189.

[20] J. D. Guttman and F. J. Thayer, "Protocol independence through disjoint encryption," in *CSFW*, 2000, pp. 24–34.

[21] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *FOCS*, 2001, pp. 136–145.

[22] B. Pfitzmann and M. Waidner, "A model for asyn-

chronous reactive systems and its application to secure message transmission," in *IEEE Symposium on Security and Privacy*, 2001, pp. 184–.

[23] A. Datta, A. Derek, J. C. Mitchell, and D. Pavlovic, "A derivation system and compositional logic for security protocols," *Journal of Computer Security*, vol. 13, no. 3, pp. 423–482, 2005.

[24] K. Bhargavan, C. Fournet, and A. D. Gordon, "Modular verification of security protocol code by typing," in *Proceedings of the 37th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2010, to appear.

[25] V. Cortier and S. Delaune, "Safely composing security protocols," *Formal Methods in System Design*, vol. 34, no. 1, pp. 1–36, 2009.

[26] A. Datta, A. Derek, J. C. Mitchell, and A. Roy, "Protocol Composition Logic (PCL)," *Electronic Notes in Theoretical Computer Science*, vol. 172, pp. 311–358, 2007.

[27] A. Roy, A. Datta, A. Derek, J. C. Mitchell, and S. Jean-Pierre, "Secrecy analysis in protocol composition logic," in *Formal Logical Methods for System Security and Correctness*. IOS Press, 2008.

[28] C. He, M. Sundararajan, A. Datta, A. Derek, and J. C. Mitchell, "A modular correctness proof of IEEE 802.11i and TLS," in *CCS '05: Proceedings of the 12th ACM Conference on Computer and Communications Security*, 2005, pp. 2–15.

[29] A. Datta, J. Franklin, D. Garg, and D. Kaynar, "A logic of secure systems and its application to trusted computing," in *Proceedings of the 30th IEEE Symposium on Security and Privacy (Oakland)*, 2009, pp. 221–236.

[30] B. Alpern and F. B. Schneider, "Recognizing safety and liveness," *Distributed Computing*, vol. 2, no. 3, pp. 117–126, 1987.

[31] Z. Manna and A. Pnueli, *Temporal verification of reactive systems: safety*. Springer-Verlag New York, Inc., 1995.

[32] J. Reed, "A hybrid logical framework," Ph.D. dissertation, Carnegie Mellon University, 2009.

[33] P. Blackburn, "Representation, reasoning, and relational structures: A hybrid logic manifesto," *Logic Journal of IGPL*, vol. 8, no. 3, pp. 339–365, 2000.

[34] A. Nanevski, G. Morrisett, and L. Birkedal, "Hoare type theory, polymorphism and separation," *Journal of Functional Programming*, vol. 18, no. 5&6, pp. 865–911, 2008.

[35] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman, "Protection in operating systems," *Communications of the ACM*, vol. 19, no. 8, pp. 461–471, 1976.

[36] X. Feng, R. Ferreira, and Z. Shao, "On the relationship between concurrent separation logic and assume-guarantee reasoning," in *Programming Languages and Systems, Proceedings of the 16th European Symposium on Programming (ESOP)*, 2007, pp. 173–188.

$\boxed{\sigma \rhd T \hookrightarrow \sigma' \rhd T'}$

$$\frac{\texttt{eval}\ t\ t'}{\sigma \rhd I\,;(x.e) :: K\,;t \hookrightarrow \sigma \rhd I\,;K\,;e\{t'/x\}}\text{red-term}$$

$$\frac{\sigma\,;I \rhd a \mapsto \sigma'\,;I \rhd t \qquad \texttt{eval}\ t\ t'}{\sigma \rhd I\,;(x.e) :: K\,;\texttt{act}\ a \hookrightarrow \sigma' \rhd I\,;K\,;e\{t'/x\}}\text{red-act}$$

$$\frac{}{\sigma \rhd I\,;K\,;\texttt{let}(e_1, x.e_2) \hookrightarrow \sigma \rhd I\,;(x.e_2) :: K\,;e_1}\text{red-let}$$

$$\frac{\texttt{eval}\ b\ \texttt{true}}{\sigma \rhd I\,;K\,;\texttt{if}(b, e_1, e_2) \hookrightarrow \sigma \rhd I\,;K\,;e_1}\text{red-if-t}$$

$$\frac{\texttt{eval}\ b\ \texttt{false}}{\sigma \rhd I\,;K\,;\texttt{if}(b, e_1, e_2) \hookrightarrow \sigma \rhd I\,;K\,;e_2}\text{red-if-f}$$

$$\frac{f(x) \triangleq e \qquad \texttt{eval}\ t_2\ t_2'}{\sigma \rhd I\,;K\,;\texttt{call}(f, t_2) \hookrightarrow \sigma \rhd I\,;K\,;e\{t_2'/x\}}\text{red-call}$$

$\boxed{\mathcal{C} \longrightarrow \mathcal{C}'}$

$$\frac{\sigma \rhd T \hookrightarrow \sigma' \rhd T'}{\sigma \rhd T, T_1, \ldots, T_n \longrightarrow \sigma' \rhd T', T_1, \ldots, T_n}\text{red-config}$$

Figure 4: Operational semantics

APPENDIX A. OPERATIONAL SEMANTICS

The operational semantics of the programming language are shown in Figure 4.

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 $\mathcal{T} \models \varphi$ and $\mathcal{T}, n \models \mu$. The former means that the *ground* ordinary formula φ holds on trace $\mathcal{T}$. $\mathcal{T}, n \models \mu$ is explained later. We define these relations by induction on φ and μ. Semantics for formulas with free variables are a special case of semantics for hypothetical judgments, which are explained later in this section.

Definition C.1 (Subtraces)**.** Let $\mathcal{T}$ be the trace

$$\xrightarrow{u_0} \mathcal{C}_0 \xrightarrow{u_1} \mathcal{C}_1 \ldots \xrightarrow{u_n} \mathcal{C}_n$$

For any $k \geq 0$, we define the truncation of $\mathcal{T}$ to k, written $\texttt{trunc}(\mathcal{T}, k)$ as the trace which contains only the last $k+1$ configurations of $\mathcal{T}$. More precisely,

- If $k \geq n$, then $\texttt{trunc}(\mathcal{T}, k)$ equals $\mathcal{T}$.

Temporal formulas

$$\frac{}{\Sigma;\Gamma,\varphi\vdash\varphi}\text{hypO} \qquad \frac{}{\Sigma;\Gamma\vdash t=t}\text{eqI} \qquad \frac{\Sigma;\Gamma\vdash t=t' \qquad \Sigma;\Gamma\vdash\varphi\{t/x\}}{\Sigma;\Gamma\vdash\varphi\{t'/x\}}\text{eqE} \qquad \frac{}{\Sigma;\Gamma\vdash u\le u}\text{timeR}$$

$$\frac{\Sigma;\Gamma\vdash u_1\le u_2 \qquad \Sigma;\Gamma\vdash u_2\le u_3}{\Sigma;\Gamma\vdash u_1\le u_3}\text{timeT} \qquad \frac{}{\Sigma;\Gamma\vdash(u_1\le u_2)\vee(u_2\le u_1)}\text{timet}$$

$$\frac{}{\Sigma;\Gamma\vdash((u_1\le u_2)\wedge(u_2\le u_1))\supset u_1=u_2}\text{timeAS} \qquad \frac{\Sigma;\Gamma\vdash\varphi \qquad \Sigma;\Gamma,\varphi\vdash\varphi'}{\Sigma;\Gamma\vdash\varphi'}\text{cutOO} \qquad \frac{}{\Sigma;\Gamma\vdash\varphi\vee\neg\varphi}\text{EM}$$

$$\frac{\Sigma;\Gamma\vdash\varphi \qquad \Sigma;\Gamma\vdash\psi}{\Sigma;\Gamma\vdash\varphi\wedge\psi}\wedge\text{I} \qquad \frac{\Sigma;\Gamma\vdash\varphi\wedge\psi}{\Sigma;\Gamma\vdash\varphi}\wedge\text{E1} \qquad \frac{\Sigma;\Gamma\vdash\varphi\wedge\psi}{\Sigma;\Gamma\vdash\psi}\wedge\text{E2} \qquad \frac{\Sigma;\Gamma\vdash\varphi}{\Sigma;\Gamma\vdash\varphi\vee\psi}\vee\text{I1} \qquad \frac{\Sigma;\Gamma\vdash\psi}{\Sigma;\Gamma\vdash\varphi\vee\psi}\vee\text{I2}$$

$$\frac{\Sigma;\Gamma\vdash\varphi_1\vee\varphi_2 \qquad \Sigma;\Gamma,\varphi_1\vdash\varphi \qquad \Sigma;\Gamma,\varphi_2\vdash\varphi}{\Sigma;\Gamma\vdash\varphi}\vee\text{E} \qquad \frac{\Sigma;\Gamma,\varphi\vdash\psi}{\Sigma;\Gamma\vdash\varphi\supset\psi}\supset\text{I} \qquad \frac{\Sigma;\Gamma\vdash\varphi\supset\psi \qquad \Sigma;\Gamma\vdash\varphi}{\Sigma;\Gamma\vdash\psi}\supset\text{E} \qquad \frac{}{\Sigma;\Gamma\vdash\top}\top\text{I}$$

$$\frac{}{\Sigma;\Gamma,\bot\vdash\varphi}\bot\text{E} \qquad \frac{\Sigma,x;\Gamma\vdash\varphi}{\Sigma;\Gamma\vdash\forall x.\varphi}\forall\text{I} \qquad \frac{\Sigma;\Gamma\vdash\forall x.\varphi}{\Sigma;\Gamma\vdash\varphi\{t/x\}}\forall\text{E} \qquad \frac{\Sigma;\Gamma\vdash\varphi\{t/x\}}{\Sigma;\Gamma\vdash\exists x.\varphi}\exists\text{I} \qquad \frac{\Sigma;\Gamma\vdash\exists x.\varphi \qquad \Sigma,x;\Gamma,\varphi\vdash\varphi}{\Sigma;\Gamma\vdash\varphi}\exists\text{E}$$

Figure 5: Proof System, part 1

- If $k<n$, then $\texttt{trunc}(\mathcal{T},k)$ is the trace:

$$\xrightarrow{u_{n-k}}\mathcal{C}_{n-k}\xrightarrow{u_{n-k+1}}\mathcal{C}_{n-k+1}\ldots\xrightarrow{u_n}\mathcal{C}_n$$

Note that $\texttt{trunc}(\mathcal{T},k)$ is not a trace in the strictest sense since it does not have the auxiliary information that may be associated with $\mathcal{T}$. 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 $t\doteq t'$ 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 $\texttt{eval}\ t\ t'$, then $[\![t]\!]\doteq[\![t']\!]$. Note that given a semantic definition of $[\![\cdot]\!]$, this clause imposes a constraint on `eval`.
  - $[\![[\![t]\!]]\!]\doteq[\![t]\!]$
  - If $[\![t_1]\!]\doteq[\![t_2]\!]$, then $[\![t\{t_1/x\}]\!]\doteq[\![t\{t_2/x\}]\!]$.
- There is a Boolean-valued function $V(\mathcal{T},u,p)$ which given a trace $\mathcal{T}$, a time point u, and a ground atomic formula p, states whether p holds on $\mathcal{T}$ at time u or not. We require the following:
  - If $[\![t]\!]\doteq[\![t']\!]$, then $V(\mathcal{T},u,p\{t/x\})=V(\mathcal{T},u,p\{t'/x\})$.

Semantics of temporal formulas:

- $\mathcal{T}\models t_1=t_2$ iff $[\![t_1]\!]\doteq[\![t_2]\!]$.
- $\mathcal{T}\models u_1\le u_2$ iff u_1 is less than or equal to u_2 in the usual order on real numbers.
- $\mathcal{T}\models p\ @\ u$ iff $V(\mathcal{T},u,p)$.
- $\mathcal{T}\models\varphi\wedge\psi$ iff $\mathcal{T}\models\varphi$ and $\mathcal{T}\models\psi$.
- $\mathcal{T}\models\varphi\vee\psi$ iff $\mathcal{T}\models\varphi$ or $\mathcal{T}\models\psi$.
- $\mathcal{T}\models\varphi\supset\psi$ iff $\mathcal{T}\not\models\varphi$ or $\mathcal{T}\models\psi$.
- $\mathcal{T}\models\top$ always.
- $\mathcal{T}\models\bot$ never.
- $\mathcal{T}\models\forall x.\varphi$ iff for every ground term t, $\mathcal{T}\models\varphi\{t/x\}$.
- $\mathcal{T}\models\exists x.\varphi$ iff there is a ground term t, $\mathcal{T}\models\varphi\{t/x\}$.

Semantics of assertions: The semantic relation for assertions has the form $\mathcal{T},n\models\mu$. 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.

- $\mathcal{T},n\models[]\langle u_b,u_e,i\rangle\varphi$ iff whenever in subtrace $\texttt{trunc}(\mathcal{T},n)$ thread I has no effectual reduction in the interval (u'_b,u'_e), it is the case that $\mathcal{T}\models\varphi\{u'_b/u_b\}\{u'_e/u_e\}\{I/i\}$.
- $\mathcal{T},n\models[a]\langle u_b,u_e,i,x\rangle\varphi$ iff whenever in subtrace $\texttt{trunc}(\mathcal{T},n)$ the active expression in I at time u'_b is $\texttt{act}\ a$, and at time u'_e this action a reduces returning term t to its continuation, it is the case that $\mathcal{T}\models\varphi\{u'_b/u_b\}\{u'_e/u_e\}\{I/i\}\{t/x\}$. More precisely, $\mathcal{T},n\models[a]\langle u_b,u_e,i,x\rangle\varphi$ holds iff $\mathcal{T}\models\varphi\{u'_b/u_b\}\{u'_e/u_e\}\{I/i\}\{t/x\}$ whenever the following pattern matches the subtrace $\texttt{trunc}(\mathcal{T},n)$, and there is no reduction in thread I in the interval (u'_b,u'_e).

$$\xrightarrow{u'_e}\begin{array}{l}\ldots\\ \sigma\rhd I\,;(x.e)::K\,;\texttt{act}\ a\\ \sigma'\rhd I\,;K\,;e\{t/x\}\\ \ldots\end{array}$$

- $\mathcal{T},n\models[e]\langle u_b,u_e,i,x\rangle\varphi$ iff whenever in thread I in subtrace $\texttt{trunc}(\mathcal{T},n)$ the active expression at

Partial correctness assertions

$$\frac{\Sigma;\Gamma;\Delta \vdash []\langle u_b, u_e, i\rangle\varphi}{\Sigma;\Gamma;\Delta \vdash [t]\langle u_b, u_e, i, x\rangle\varphi \wedge (x = t)}\text{PT} \qquad \frac{\Sigma;\Gamma;\Delta \vdash [a]\langle u_b, u_e, i, x\rangle\varphi}{\Sigma;\Gamma;\Delta \vdash [\texttt{act } a]\langle u_b, u_e, i, x\rangle\varphi}\text{PA}$$

$$\frac{\Sigma;\Gamma;\Delta \vdash []\langle u_b, u_m, i\rangle\varphi_1 \qquad \Sigma;\Gamma;\Delta \vdash [e_1]\langle u_m, u'_m, i, y\rangle\varphi_2 \qquad \Sigma, y;\Gamma;\Delta \vdash [e_2]\langle u'_m, u_e, i, x\rangle\varphi_3}{\Sigma;\Gamma;\Delta \vdash [\texttt{let}(e_1, y.e_2)]\langle u_b, u_e, i, x\rangle\exists y.\exists u_m.\exists u'_m.((u_b < u_m < u'_m < u_e) \wedge \varphi_1 \wedge \varphi_2 \wedge \varphi_3)}\text{PL}$$

$$\frac{\Sigma;\Gamma;\Delta \vdash []\langle u_b, u_m, i\rangle\varphi_1 \qquad \Sigma;\Gamma;\Delta \vdash [e_1]\langle u_m, u_e, i, x\rangle\varphi_2 \qquad \Sigma;\Gamma;\Delta \vdash [e_2]\langle u_m, u_e, i, x\rangle\varphi_3}{\Sigma;\Gamma;\Delta \vdash [\texttt{if}(b, e_1, e_2)]\langle u_b, u_e, i, x\rangle\exists u_m.((u_b < u_m < u_e) \wedge \varphi_1 \wedge (b \supset \varphi_2) \wedge ((\neg b) \supset \varphi_3))}\text{PI}$$

$$\frac{\Sigma;\Gamma;\Delta \vdash []\langle u_b, u_m, i\rangle\varphi_1 \qquad \Sigma;\Gamma;\Delta \vdash [f]\langle y, u_m, u_e, i, x\rangle\varphi_2}{\Sigma;\Gamma;\Delta \vdash [\texttt{call}(f, t_2)]\langle u_b, u_e, i, x\rangle\exists u_m.((u_b < u_m < u_e) \wedge \varphi_1 \wedge \varphi_2\{t_2/y\})}\text{PC}$$

$$\frac{f(z) \triangleq e \qquad \Sigma, y;\Gamma;\Delta, [f]\langle y, u_b, u_e, i, x\rangle\varphi \vdash [e\{y/z\}]\langle u_b, u_e, i, x\rangle\varphi}{\Sigma;\Gamma;\Delta \vdash [f]\langle y, u_b, u_e, i, x\rangle\varphi}\text{PF}$$

Invariance assertions

$$\frac{\Sigma;\Gamma;\Delta \vdash []\langle u_b, u_e, i\rangle\varphi}{\Sigma;\Gamma;\Delta \vdash \{t\}\langle u_b, u_e, i\rangle\varphi}\text{IT} \qquad \frac{\Sigma;\Gamma;\Delta \vdash []\langle u_b, u_e, i\rangle\varphi}{\Sigma;\Gamma;\Delta \vdash \{\texttt{act } a\}\langle u_b, u_e, i\rangle\varphi}\text{IA}$$

$$\frac{\begin{array}{c}\Sigma;\Gamma;\Delta \vdash []\langle u_b, u_e, i\rangle\varphi \\ \Sigma;\Gamma;\Delta \vdash []\langle u_b, u_m, i\rangle\psi_1 \qquad \Sigma;\Gamma;\Delta \vdash \{e_1\}\langle u_m, u_e, i\rangle\psi_2 \qquad \Sigma, u_b, u_m, u_e, i;\Gamma, u_b < u_m \le u_e, \psi_1, \psi_2 \vdash \varphi \\ \Sigma;\Gamma;\Delta \vdash []\langle u_b, u_m, i\rangle\psi_3 \qquad \Sigma;\Gamma;\Delta \vdash [e_1]\langle u_m, u'_m, i, y\rangle\psi_4 \\ \Sigma, y;\Gamma;\Delta \vdash \{e_2\}\langle u'_m, u_e, i\rangle\psi_5 \qquad \Sigma, u_b, u_m, u'_m, u_e, i, y;\Gamma, u_b < u_m < u'_m \le u_e, \psi_3, \psi_4, \psi_5 \vdash \varphi\end{array}}{\Sigma;\Gamma;\Delta \vdash \{\texttt{let}(e_1, y.e_2)\}\langle u_b, u_e, i\rangle\varphi}\text{IL}$$

$$\frac{\begin{array}{c}\Sigma;\Gamma;\Delta \vdash []\langle u_b, u_e, i\rangle\varphi \\ \Sigma;\Gamma;\Delta \vdash []\langle u_b, u_m, i\rangle\psi_1 \qquad \Sigma;\Gamma;\Delta \vdash \{e_1\}\langle u_m, u_e, i\rangle\psi_2 \qquad \Sigma, u_b, u_m, u_e, i;\Gamma, u_b < u_m \le u_e, \psi_1, \psi_2, b \vdash \varphi \\ \Sigma;\Gamma;\Delta \vdash []\langle u_b, u_m, i\rangle\psi_3 \qquad \Sigma;\Gamma;\Delta \vdash \{e_2\}\langle u_m, u_e, i\rangle\psi_4 \qquad \Sigma, u_b, u_m, u_e, i;\Gamma, u_b < u_m \le u_e, \psi_3, \psi_4, \neg b \vdash \varphi\end{array}}{\Sigma;\Gamma;\Delta \vdash \{\texttt{if}(b, e_1, e_2)\}\langle u_b, u_e, i\rangle\varphi}\text{II}$$

$$\frac{\begin{array}{c}\Sigma;\Gamma;\Delta \vdash []\langle u_b, u_e, i\rangle\varphi \\ \Sigma;\Gamma;\Delta \vdash []\langle u_b, u_m, i\rangle\psi \qquad \Sigma;\Gamma;\Delta \vdash \{f\}\langle y, u_m, u_e, i\rangle\psi' \\ \Sigma, u_b, u_m, u_e, i;\Gamma, u_b < u_m \le u_e, \psi, \psi'\{t_2/y\} \vdash \varphi\end{array}}{\Sigma;\Gamma;\Delta \vdash \{\texttt{call}(f, t_2)\}\langle u_b, u_e, i\rangle\varphi}\text{IC}$$

$$\frac{f(z) \triangleq e \qquad \Sigma, y;\Gamma;\Delta, \{f\}\langle y, u_b, u_e, i\rangle\varphi \vdash \{e\{y/z\}\}\langle u_b, u_e, i\rangle\varphi}{\Sigma;\Gamma;\Delta \vdash \{f\}\langle y, u_b, u_e, i\rangle\varphi}\text{IF}$$

Trusted and Untrusted Threads

$$\frac{\Sigma;\Gamma;\cdot \vdash \{e\}\langle u_b, u_e, i\rangle\varphi(u_b, u_e, i) \qquad \Sigma;\Gamma \vdash \texttt{HonestThread}(I, e) \qquad \Sigma;\Gamma \vdash \texttt{Start}(I) \mathbin{@} u}{\Sigma;\Gamma \vdash \forall u'.\ (u' > u) \supset \varphi(u, u', I)}\text{HONTH}$$

$$\frac{\begin{array}{c}(\varphi(u_b, u_e, i) \text{ compositional}) \qquad \cdot;\Gamma;\cdot \vdash []\langle u_b, u_e, i\rangle\varphi(u_b, u_e, i) \\ \forall f \in \mathcal{F}.\ (\cdot;\Gamma;\cdot \vdash \{f\}\langle y, u_b, u_e, i\rangle\varphi(u_b, u_e, i)) \qquad \forall f \in \mathcal{F}.\ (\cdot;\Gamma;\cdot \vdash [f]\langle y, u_b, u_e, i, x\rangle\varphi(u_b, u_e, i)) \qquad \Sigma;\Gamma \vdash \texttt{Confined}(I, \mathcal{F})\end{array}}{\Sigma;\Gamma, \Gamma' \vdash \forall u_e.\ \varphi(-\infty, u_e, I)}\text{RES}$$

Figure 6: Proof System, part 2

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

General rules for assertions

$$\frac{(\Sigma;\Gamma;\Delta \vdash \mu) \text{ is an axiom}}{\Sigma,\Sigma';\Gamma,\Gamma';\Delta,\Delta' \vdash \mu}\text{axM} \qquad \frac{}{\Sigma;\Gamma;\Delta,\mu \vdash \mu}\text{hypM} \qquad \frac{\Sigma;\Gamma;\Delta \vdash \mu \qquad \Sigma;\Gamma;\Delta,\mu \vdash \mu'}{\Sigma;\Gamma;\Delta \vdash \mu'}\text{cutMM}$$

$$\frac{\Sigma;\Gamma \vdash \varphi \qquad \Sigma;\Gamma,\varphi;\Delta \vdash \mu'}{\Sigma;\Gamma;\Delta \vdash \mu'}\text{cutOM} \qquad \frac{\Sigma;\Gamma \vdash \varphi}{\Sigma;\Gamma;\Delta \vdash []\langle u_b,u_e,i\rangle\varphi}\text{GOE} \qquad \frac{\Sigma;\Gamma;\Delta \vdash []\langle u_b,u_e,i\rangle\varphi \qquad \Sigma,u_b,u_e,i;\Gamma,\varphi \vdash \varphi'}{\Sigma;\Gamma;\Delta \vdash []\langle u_b,u_e,i\rangle\varphi'}\text{GE}$$

$$\frac{\Sigma;\Gamma \vdash \varphi}{\Sigma;\Gamma;\Delta \vdash [a]\langle u_b,u_e,i,x\rangle\varphi}\text{GOA} \qquad \frac{\Sigma;\Gamma;\Delta \vdash [a]\langle u_b,u_e,i,x\rangle\varphi \qquad \Sigma,u_b,u_e,i,x;\Gamma,\varphi \vdash \varphi'}{\Sigma;\Gamma;\Delta \vdash [a]\langle u_b,u_e,i,x\rangle\varphi'}\text{GA}$$

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

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

$$\frac{\Sigma;\Gamma \vdash \varphi}{\Sigma;\Gamma;\Delta \vdash [f]\langle y,u_b,u_e,i,x\rangle\varphi}\text{GOFP} \qquad \frac{\Sigma;\Gamma;\Delta \vdash [f]\langle y,u_b,u_e,i,x\rangle\varphi \qquad \Sigma,y,u_b,u_e,i,x;\Gamma,\varphi \vdash \varphi'}{\Sigma;\Gamma;\Delta \vdash [f]\langle y,u_b,u_e,i,x\rangle\varphi'}\text{GFP}$$

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

Figure 7: Proof System, part 3

$(x.e') :: K$ in the interval (u', u'_e).

$$\begin{array}{rl} & \ldots \\ & \sigma \rhd I\,;(x.e') :: K\,;e \\ \xrightarrow{u'} & \ldots \\ & \sigma \rhd I\,;(x.e') :: K\,;t \\ \xrightarrow{u'_e} & \sigma \rhd I\,;K\,;e'\{t'/x\} \\ & \ldots \end{array}$$

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

$$\begin{array}{rl} & \ldots \\ \xrightarrow{u'} & \sigma \rhd I\,;K\,;e \\ & \ldots \end{array}$$

- $\mathcal{T}, n \models [f]\langle y, u_b, u_e, i, x\rangle\varphi$ iff whenever in thread I in subtrace $\texttt{trunc}(\mathcal{T}, n)$, f is called at time u'_b with argument t_2, and at time u'_e returns term t to its continuation, it is the case that $\mathcal{T} \models \varphi\{t_2/y\}\{u'_b/u_b\}\{u'_e/u_e\}\{I/i\}\{t/x\}$. More precisely, $\mathcal{T}, n \models [f]\langle y, u_b, u_e, i, x\rangle\varphi$ holds iff $\mathcal{T} \models \varphi\{t_2/y\}\{u'_b/u_b\}\{u'_e/u_e\}\{I/i\}\{t/x\}$ whenever the following pattern matches the subtrace $\texttt{trunc}(\mathcal{T}, n)$ for $\texttt{eval}\ t\ t'$, $\texttt{eval}\ t_2\ t'_2$, and $f(y) \triangleq e$, and in the interval (u'_b, u'_e) the stack of I has suffix $(x.e') :: K$.

$$\begin{array}{rl} & \ldots \\ & \sigma \rhd I\,;(x.e') :: K\,;\texttt{call}(f, t_2) \\ \xrightarrow{u'_b} & \sigma \rhd I\,;(x.e') :: K\,;e\{t'_2/y\} \\ & \ldots \\ & \sigma \rhd I\,;(x.e') :: K\,;t \\ \xrightarrow{u'_e} & \sigma \rhd I\,;K\,;e'\{t'/x\} \\ & \ldots \end{array}$$

- $\mathcal{T}, n \models \{f\}\langle y, u_b, u_e, i\rangle\varphi$ iff whenever in thread I in subtrace $\texttt{trunc}(\mathcal{T}, n)$, f is called at time u'_b with argument t_2, and until time u'_e it has not returned, it is the case that $\mathcal{T} \models \varphi\{t_2/y\}\{u'_b/u_b\}\{u'_e/u_e\}\{I/i\}$. More precisely, $\mathcal{T}, n \models \{f\}\langle y, u_b, u_e, i\rangle\varphi$ holds iff $\mathcal{T} \models \varphi\{t_2/y\}\{u'_b/u_b\}\{u'_e/u_e\}\{I/i\}$ whenever the following pattern matches the subtrace $\texttt{trunc}(\mathcal{T}, n)$ for $\texttt{eval}\ t_2\ t'_2$, and $f(y) \triangleq e$, and in the interval $(u'_b, u'_e]$ the stack of I has suffix K.

$$\begin{array}{rl} & \ldots \\ & \sigma \rhd I\,;K\,;\texttt{call}(f, t_2) \\ \xrightarrow{u'_b} & \sigma \rhd I\,;K\,;e\{t'_2/y\} \\ & \ldots \end{array}$$

Observation C.2. The following hold:

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

Semantics of sequents: We lift semantics from formulas to sequents as follows.

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

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)**.** $\texttt{eval}\ t\ t'$ *implies* ($\mathcal{T} \models \varphi\{t/x\}$ *iff* $\mathcal{T} \models \varphi\{t'/x\}$)*.*

Proof: Since $\texttt{eval}\ t\ t'$, $[\![t]\!] \doteq [\![t']\!]$ by assumption. It therefore suffices to show that $[\![t]\!] \doteq [\![t']\!]$ implies ($\mathcal{T} \models \varphi\{t/x\}$ iff $\mathcal{T} \models \varphi\{t'/x\}$). This follows by induction on φ. For the base case where $\varphi = p\ @\ u$, we appeal to the assumption that $[\![t]\!] \doteq [\![t']\!]$ implies $V(\mathcal{T}, u, p\{t/x\}) = V(\mathcal{T}, u, p\{t'/x\})$. ∎

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

Definition D.2 ($\mathcal{F}$-invariant)**.** Given an interface $\mathcal{F}$, a formula $\varphi(u_b, u_e, i)$ containing only the free variables u_b, u_e, and i is called an ($\mathcal{F}$,Γ,Δ)-invariant if the following hold for every Σ:

- $\varphi(u_b, u_e, i)$ is compositional (see Section IV-C for a definition of compositional)
- For every $f \in \mathcal{F}$, $\Sigma; \Gamma; \Delta \vdash \{f\}\langle y, u_b, u_e, i\rangle\varphi(u_b, u_e, i)$
- For every $f \in \mathcal{F}$, $\Sigma; \Gamma; \Delta \vdash [f]\langle y, u_b, u_e, i, x\rangle\varphi(u_b, u_e, i)$
- $\Sigma; \Gamma; \Delta \vdash [\,]\langle u_b, u_e, i\rangle\varphi(u_b, u_e, i)$

Lemma D.3 (Invariance of $\mathcal{F}$-confined expressions)**.** *Let e be $\mathcal{F}$-confined, and let $\varphi(u_b, u_e, i)$ be an ($\mathcal{F}$,Γ,Δ)-invariant. Let $\Sigma \supseteq \texttt{fv}(e)$. Then,*

1) $\Sigma; \Gamma; \Delta \vdash [e]\langle u_b, u_e, i, x\rangle\varphi(u_b, u_e, i)$
2) $\Sigma; \Gamma; \Delta \vdash \{e\}\langle u_b, u_e, i\rangle\varphi(u_b, u_e, i)$

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

Case. $e = t$. To show: $\Sigma; \Gamma; \Delta \vdash [t]\langle u_b, u_e, i, x\rangle\varphi(u_b, u_e, i)$. We have:

1) $\Sigma; \Gamma; \Delta \vdash [\,]\langle u_b, u_e, i, x\rangle\varphi(u_b, u_e, i)$ (Defn. of ($\mathcal{F}$,Γ,Δ)-invariant)
2) $\Sigma; \Gamma; \Delta \vdash [t]\langle u_b, u_e, i, x\rangle\varphi(u_b, u_e, i) \wedge x = t$ (Rule PT on 1)
3) $\Sigma, u_b, u_e, i, x; \Gamma, \varphi(u_b, u_e, i) \wedge x = t \vdash \varphi(u_b, u_e, i)$ (Thm. in predicate logic)
4) $\Sigma; \Gamma; \Delta \vdash [t]\langle u_b, u_e, i, x\rangle\varphi(u_b, u_e, i)$ (Rule GEP on 2,3)

Case. $e = \texttt{act}\ a$ does not apply due to definition of $\mathcal{F}$-confined.

Case. $e = \texttt{let}(e_1, y.e_2)$. To show: $\Sigma; \Gamma; \Delta \vdash [\texttt{let}(e_1, y.e_2)]\langle u_b, u_e, i, x\rangle\varphi(u_b, u_e, i)$. We have:

1) $\Sigma; \Gamma; \Delta \vdash [\,]\langle u_b, u_m, i\rangle\varphi(u_b, u_m, i)$ (Defn. of $\mathcal{F}$-invariant and weakening)
2) $\Sigma; \Gamma; \Delta \vdash [e_1]\langle u_m, u'_m, i, y\rangle\varphi(u_m, u'_m, i)$ (i.h. on e_1)
3) $\Sigma, y; \Gamma; \Delta \vdash [e_2]\langle u'_m, u_e, i, x\rangle\varphi(u'_m, u_e, i)$ (i.h. on e_2)
4) $\Sigma; \Gamma; \Delta \vdash [\texttt{let}(e_1, y.e_2)]\langle u_b, u_e, i, x\rangle\exists y.\exists u_m.\exists u'_m.((u_b < u_m < u'_m < u_e) \wedge \varphi(u_b, u_m, i) \wedge \varphi(u_m, u'_m, i) \wedge \varphi(u'_m, u_e, i))$ (Rule PL on 1–3)
5) $\Sigma, u_b, u_m, u_e, i; \Gamma, u_b < u_m \leq u_e, \varphi(u_b, u_m, i), \varphi(u_m, u_e, i) \vdash \varphi(u_b, u_e, i)$ (Defn. of ($\mathcal{F}$,Γ,Δ)-invariant)
6) $\Sigma, u_b, u_e, i, x; \Gamma, \exists y.\exists u_m.\exists u'_m.((u_b < u_m < u'_m < u_e) \wedge \varphi(u_b, u_m, i) \wedge \varphi(u_m, u'_m, i) \wedge \varphi(u'_m, u_e, i)) \vdash \varphi(u_b, u_e, i)$ (Thm. in predicate logic using 5)
7) $\Sigma; \Gamma; \Delta \vdash [\texttt{let}(e_1, y.e_2)]\langle u_b, u_e, i, x\rangle\varphi(u_b, u_e, i)$ (Rule GEP on 4,6)

Case. $e = \texttt{call}(f, t)$. To show: $\Sigma; \Gamma; \Delta \vdash [\texttt{call}(f, t)]\langle u_b, u_e, i, x\rangle\varphi(u_b, u_e, i)$. By definition of $\mathcal{F}$-confined, $f \in \mathcal{F}$. We have:

1) $\Sigma; \Gamma; \Delta \vdash [\,]\langle u_b, u_m, i\rangle\varphi(u_b, u_e, i)$ (Defn. of ($\mathcal{F}$,Γ,Δ)-invariant)
2) $\Sigma; \Gamma; \Delta \vdash [f]\langle y, u_m, u_e, i, x\rangle\varphi(u_m, u_e, i)$ (Defn. of ($\mathcal{F}$,Γ,Δ)-invariant)
3) $\Sigma; \Gamma; \Delta \vdash [\texttt{call}(f, t)]\langle u_b, u_e, i, x\rangle\exists u_m.((u_b < u_m < u_e) \wedge \varphi(u_b, u_e, i) \wedge \varphi(u_m, u_e, i))$ (Rule PC on 1,2)
4) $\Sigma, u_b, u_m, u_e, i; \Gamma, u_b < u_m \leq u_e, \varphi(u_b, u_m, i), \varphi(u_m, u_e, i) \vdash \varphi(u_b, u_e, i)$ (Defn. of ($\mathcal{F}$,Γ,Δ)-invariant)
5) $\Sigma, u_b, u_e, i, x; \Gamma, \exists u_m.((u_b < u_m < u_e) \wedge \varphi(u_b, u_e, i) \wedge \varphi(u_m, u_e, i)) \vdash \varphi(u_b, u_e, i)$ (Thm. in predicate logic using 4)
6) $\Sigma; \Gamma; \Delta \vdash [\texttt{call}(f, t)]\langle u_b, u_e, i, x\rangle\varphi(u_b, u_e, i)$ (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 $e = \texttt{call}(f,t)$.

Case. $e = \texttt{call}(f,t)$. To show: $\Sigma;\Gamma;\Delta \vdash \{\texttt{call}(f,t)\}\langle u_b,u_e,i\rangle\varphi(u_b,u_e,i)$. We have:

1) $\Sigma;\Gamma;\Delta \vdash []\langle u_b,u_e,i\rangle\varphi(u_b,u_e,i)$ (Defn. of ($\mathcal{F}$,Γ,Δ)-invariant)
2) $\Sigma;\Gamma;\Delta \vdash []\langle u_b,u_m,i\rangle\varphi(u_b,u_m,i)$ (Defn. of ($\mathcal{F}$,Γ,Δ)-invariant)
3) $\Sigma;\Gamma;\Delta \vdash \{f\}\langle y,u_m,u_e,i\rangle\varphi(u_m,u_e,i)$ (Defn. of ($\mathcal{F}$,Γ,Δ)-invariant)
4) $\Sigma,u_b,u_m,u_e,i;\Gamma,u_b < u_m \leq u_e,\varphi(u_b,u_m,i),\varphi(u_m,u_e,i) \vdash \varphi(u_b,u_e,i)$ (Defn. of ($\mathcal{F}$,Γ,Δ)-invariant)
5) $\Sigma;\Gamma;\Delta \vdash \{\texttt{call}(f,t)\}\langle u_b,u_e,i\rangle\varphi(u_b,u_e,i)$ (Rule IC on 1–4)

■

Lemma D.4 (Invariance of $\mathcal{F}$-limited functions)**.** *Let $\mathcal{G}$ be a set of $\mathcal{F}$-limited functions and let $\varphi(u_b,u_e,i)$ be an ($\mathcal{F}$,Γ,Δ)-invariant. Then $\varphi(u_b,u_e,i)$ is an ($\mathcal{F}\cup\mathcal{G}$,$\Gamma$,$\Delta$)-invariant.*

Proof: Let $\mathcal{G} = \{g_k \mid g_k(y) \triangleq e_k \;;\; k = 1,\ldots,n\}$. Define:

$$\begin{array}{lcl}\mu_k & = & [g_k]\langle y,u_b,u_e,i,x\rangle\varphi(u_b,u_e,i)\\ \nu_k & = & \{g_k\}\langle y,u_b,u_e,i\rangle\varphi(u_b,u_e,i)\\ \Delta' & = & \mu_1,\nu_1,\ldots,\mu_n,\nu_n\end{array}$$

Observe that by definition, and due to weakening and rule hypM, $\varphi(u_b,u_e,i)$ is an ($\mathcal{F}\cup\mathcal{G}$,$\Gamma$,($\Delta,\Delta'$))-invariant. By definition of $\mathcal{F}$-limited, it follows that the body e_k of each function in $\mathcal{G}$ is ($\mathcal{F}\cup\mathcal{G}$)-confined. Hence by Lemma D.3, we have:

- $y;\Gamma;\Delta,\Delta' \vdash [e_k]\langle u_b,u_e,i,x\rangle\varphi(u_b,u_e,i)$ for each k
- $y;\Gamma;\Delta,\Delta' \vdash \{e_k\}\langle u_b,u_e,i\rangle\varphi(u_b,u_e,i)$ for each k

Equivalently, we have

A0. $y;\Gamma;\Delta,\mu_1,\nu_1,\ldots,\mu_n,\nu_n \vdash [e_k]\langle u_b,u_e,i,x\rangle\varphi(u_b,u_e,i)$ for each k

B0. $y;\Gamma;\Delta,\mu_1,\nu_1,\ldots,\mu_n,\nu_n \vdash \{e_k\}\langle u_b,u_e,i\rangle\varphi(u_b,u_e,i)$ for each k

Choosing $k = 1$ in (A0), we get $y;\Gamma;\Delta,\mu_1,\nu_1,\ldots,\mu_n,\nu_n \vdash [e_1]\langle u_b,u_e,i,x\rangle\varphi(u_b,u_e,i)$. By rule PF, we have: $\cdot;\Gamma;\Delta,\nu_1,\ldots,\mu_n,\nu_n \vdash [g_1]\langle y,u_b,u_e,i,x\rangle\varphi(u_b,u_e,i)$ or equivalently that $\cdot;\Gamma;\Delta,\nu_1,\ldots,\mu_n,\nu_n \vdash \mu_1$. Using rule cutMM on this and (A0) and (B0), we get:

A0'. $y;\Gamma;\Delta,\nu_1,\ldots,\mu_n,\nu_n \vdash [e_k]\langle u_b,u_e,i,x\rangle\varphi(u_b,u_e,i)$ for each k

B0'. $y;\Gamma;\Delta,\nu_1,\ldots,\mu_n,\nu_n \vdash \{e_k\}\langle u_b,u_e,i\rangle\varphi(u_b,u_e,i)$ for each k

Choosing $k = 1$ in (B0'), we get $y;\Gamma;\Delta,\nu_1,\ldots,\mu_n,\nu_n \vdash \{e_1\}\langle u_b,u_e,i\rangle\varphi(u_b,u_e,i)$. By rule IF, we have: $\cdot;\Gamma;\Delta,\mu_2,\nu_2,\ldots,\mu_n,\nu_n \vdash \{g_1\}\langle y,u_b,u_e,i\rangle\varphi(u_b,u_e,i)$ or equivalently that $\cdot;\Gamma;\Delta,\mu_2,\nu_2,\ldots,\mu_n,\nu_n \vdash \nu_1$. Using rule cutMM on this and (A0') and (B0'), we get:

A1. $y;\Gamma;\Delta,\mu_2,\nu_2,\ldots,\mu_n,\nu_n \vdash [e_k]\langle u_b,u_e,i,x\rangle\varphi(u_b,u_e,i)$ for each k

B1. $y;\Gamma;\Delta,\mu_2,\nu_2,\ldots,\mu_n,\nu_n \vdash \{e_k\}\langle u_b,u_e,i\rangle\varphi(u_b,u_e,i)$ for each k

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

An. $y;\Gamma;\Delta \vdash [e_k]\langle u_b,u_e,i,x\rangle\varphi(u_b,u_e,i)$ for each k

Bn. $y;\Gamma;\Delta \vdash \{e_k\}\langle u_b,u_e,i\rangle\varphi(u_b,u_e,i)$ for each k

Weakening, we get

C. $y;\Gamma;\Delta,\mu_k \vdash [e_k]\langle u_b,u_e,i,x\rangle\varphi(u_b,u_e,i)$ for each k

D. $y;\Gamma;\Delta,\nu_k \vdash \{e_k\}\langle u_b,u_e,i\rangle\varphi(u_b,u_e,i)$ for each k

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

1) $\cdot;\Gamma;\Delta \vdash [g_k]\langle y,u_b,u_e,i,x\rangle\varphi(u_b,u_e,i)$ for each k
2) $\cdot;\Gamma;\Delta \vdash \{g_k\}\langle y,u_b,u_e,i\rangle\varphi(u_b,u_e,i)$ for each k

Further from the definition of ($\mathcal{F}$,Γ,Δ)-invariant we have:

3. $\Sigma;\Gamma;\Delta \vdash []\langle u_b,u_e,i\rangle\varphi(u_b,u_e,i)$
4. $\Sigma,u_b,u_m,u_e,i;\Gamma,u_b < u_m \leq u_e,\varphi(u_b,u_m,i),\varphi(u_m,u_e,i) \vdash \varphi(u_b,u_e,i)$
5. $\Sigma;\Gamma;\Delta \vdash [f]\langle y,u_b,u_e,i,x\rangle\varphi(u_b,u_e,i)$ for each $f \in \mathcal{F}$
6. $\Sigma;\Gamma;\Delta \vdash \{f\}\langle y,u_b,u_e,i\rangle\varphi(u_b,u_e,i)$ for each $f \in \mathcal{F}$

(1)–(6) imply that $\varphi(u_b,u_e,i)$ is an ($\mathcal{F}\cup\mathcal{G}$,$\Gamma$,$\Delta$)-invariant.

■

This lemma is also important in practice. Suppose an adversary is given an interface $\mathcal{F}$ such that $\varphi(u_b,u_e,i)$ is an ($\mathcal{F}$,Γ,Δ)-invariant. Suppose the adversary uses the functions $\mathcal{F}$ to define a new set of functions $\mathcal{G}$. By construction, $\mathcal{G}$ is $\mathcal{F}$-limited. Lemma D.4 tells us that $\varphi(u_b,u_e,i)$ is an ($\mathcal{F}\cup\mathcal{G}$,$\Gamma$,$\Delta$)-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 $[a]\langle u_b,u_e,i,x\rangle\varphi$ and $[]\langle u_b,u_e,i\rangle\varphi$) is sound. Then for every $\mathcal{T}$,*

1) $\Sigma;\Gamma \vdash \varphi$ *implies* $\mathcal{T} \models (\Sigma;\Gamma \vdash \varphi)$.
2) $\Sigma;\Gamma;\Delta \vdash \mu$ *implies* $\mathcal{T} \models (\Sigma;\Gamma;\Delta \vdash \mu)$.

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.

$$\frac{f(z) \triangleq e \qquad \Sigma, y; \Gamma; \Delta, [f]\langle y, u_b, u_e, i, x\rangle\varphi \vdash [e\{y/z\}]\langle u_b, u_e, i, x\rangle\varphi}{\Sigma; \Gamma; \Delta \vdash [f]\langle y, u_b, u_e, i, x\rangle\varphi}\text{PF}$$

We seek to show that $\mathcal{T} \models \Sigma; \Gamma; \Delta \vdash [f]\langle y, u_b, u_e, i, x\rangle\varphi$. Pick a substitution θ for Σ and an n. Assume that:

- $\mathcal{T} \models \Gamma\theta$
- $\mathcal{T}, n \models \Delta\theta$

It suffices to show that $\mathcal{T}, n \models [f]\langle y, u_b, u_e, i, x\rangle\varphi\theta$. We prove a more general statement: we use a subinduction on k to show that $\mathcal{T}, k \models [f]\langle y, u_b, u_e, i, x\rangle\varphi\theta$ for each $k \leq n$.

Subcase. $k = 0$. To show: $\mathcal{T}, 0 \models [f]\langle y, u_b, u_e, i, x\rangle\varphi\theta$. This follows from Observation C.2(2).

Subcase. $k = k' + 1 \leq n$. To show: $\mathcal{T}, k' + 1 \models [f]\langle y, u_b, u_e, i, x\rangle\varphi\theta$. Following the definition of $\mathcal{T}, k' + 1 \models [f]\langle y, u_b, u_e, i, x\rangle\varphi$, suppose that the following pattern matches thread I in $\texttt{trunc}(\mathcal{T}, k' + 1)$ where $\texttt{eval}\ t\ t'$, $\texttt{eval}\ t_2\ t_2'$, $f(z) \triangleq e$, and in the interval (u_b', u_e') the stack of I has suffix $(x.e') :: K$.

$$\begin{array}{ll} & \cdots \\ & \sigma \rhd I\,; (x.e') :: K\,; \texttt{call}(f, t_2) \\ \xrightarrow{u_b'} & \sigma \rhd I\,; (x.e') :: K\,; e\{t_2'/z\} \\ & \cdots \\ & \sigma \rhd I\,; (x.e') :: K\,; t \\ \xrightarrow{u_e'} & \sigma \rhd I\,; K\,; e'\{t'/x\} \\ & \cdots \end{array}$$

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

- $\mathcal{T}, k'' \models (\Delta, [f]\langle y, u_b, u_e, i, x\rangle\varphi)\theta$

Since we also assumed that $\mathcal{T} \models \Gamma\theta$, induction hypothesis on the premise of the rule PF implies that:

- $\mathcal{T}, k'' \models [e\theta\{y/z\}\{t_0/y\}]\langle u_b, u_e, i, x\rangle\varphi\theta\{t_0/y\}$ for any t_0.

Simplifying and choosing $t_0 = t_2'$, we get $\mathcal{T}, k'' \models [e\theta\{t_2'/z\}]\langle u_b, u_e, i, x\rangle\varphi\theta\{t_2'/y\}$. Since e is the body of f, it may contain only one free variable (z), and therefore $e\theta = e$, which gives us $\mathcal{T}, k'' \models [e\{t_2'/z\}]\langle u_b, u_e, i, x\rangle\varphi\theta\{t_2'/y\}$. Now observe that by construction, $\texttt{trunc}(\mathcal{T}, k'')$ matches the pattern:

$$\begin{array}{ll} & \sigma \rhd I\,; (x.e') :: K\,; e\{t_2'/z\} \\ & \cdots \\ & \sigma \rhd I\,; (x.e') :: K\,; t \\ \xrightarrow{u_e'} & \sigma \rhd I\,; K\,; e'\{t'/x\} \\ & \cdots \end{array}$$

By definition of $\mathcal{T}, k'' \models [e\{t_2'/z\}]\langle u_b, u_e, i, x\rangle\varphi\theta\{t_2'/y\}$, we therefore have $\mathcal{T} \models \varphi\theta\{t_2'/y\}\{u_b'/u_b\}\{u_e'/u_e\}\{I/i\}\{t/x\}$. Since $\texttt{eval}\ t_2\ t_2'$, Lemma D.1 implies $\mathcal{T} \models \varphi\theta\{t_2/y\}\{u_b'/u_b\}\{u_e'/u_e\}\{I/i\}\{t/x\}$, as required.

Case.

$$\frac{\begin{array}{c}(\varphi(u_b, u_e, i)\ \text{compositional}) \\ \forall f \in \mathcal{F}.\ (\cdot; \Gamma; \cdot \vdash \{f\}\langle y, u_b, u_e, i\rangle\varphi(u_b, u_e, i)) \\ \forall f \in \mathcal{F}.\ (\cdot; \Gamma; \cdot \vdash [f]\langle y, u_b, u_e, i, x\rangle\varphi(u_b, u_e, i)) \\ \cdot; \Gamma; \cdot \vdash []\langle u_b, u_e, i\rangle\varphi(u_b, u_e, i) \\ \Sigma; \Gamma \vdash \texttt{Confined}(I, \mathcal{F})\end{array}}{\Sigma; \Gamma, \Gamma' \vdash \forall u_e.\ \varphi(-\infty, u_e, I)}\text{RES}$$

Pick a trace $\mathcal{T}$ and a substitution θ for Σ such that $\mathcal{T} \models (\Gamma, \Gamma')\theta$. Since Γ must be closed, it follows that $\mathcal{T} \models \Gamma$. It suffices to show that for any ground u_e', $\mathcal{T} \models \varphi(-\infty, u_e', I)\theta$. From the first four premises, $\varphi(u_b, u_e, i)$ is an ($\mathcal{F}$,Γ,$\cdot$)-invariant, so $\varphi(-\infty, u_e', I)\theta = \varphi(-\infty, u_e', I\theta)$. By i.h. on the fifth premise, $\mathcal{T} \models \texttt{Confined}(I\theta, \mathcal{F})$. 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 $(\mathcal{F} \cup \mathcal{G})$-confined. By Lemma D.4, $\varphi(u_b, u_e, i)$ is an $(\mathcal{F} \cup \mathcal{G}$,$\Gamma$,$\cdot)$-invariant. By Lemma D.3, $\cdot; \Gamma; \cdot \vdash [e]\langle u_b, u_e, i, x\rangle\varphi(u_b, u_e, i)$ and $\cdot; \Gamma; \cdot \vdash \{e\}\langle u_b, u_e, i\rangle\varphi(u_b, u_e, i)$. We distinguish multiple subcases:

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

Subcase. $I\theta$ first appears in $\mathcal{T}$ 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 $\mathcal{T} \models \varphi(-\infty, u'_e, I\theta)$, since clearly $I\theta$ performs no effectual reduction in the interval $(-\infty, u'_e)$.

Subcase. $I\theta$ first appears in $\mathcal{T}$ at time u'_b and $u'_e > u'_b$. As reasoned before the case analysis, $\cdot; \Gamma; \cdot \vdash \{e\}\langle u_b, u_e, i\rangle\varphi(u_b, u_e, i)$, 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 $\mathcal{T}, \infty \models \{e\}\langle u_b, u_e, i\rangle\varphi(u_b, u_e, i)$.[3] By definition of the semantics of modal formulas, we must have $\mathcal{T} \models \varphi(u'_b, u'_e, I\theta)$. (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 $\mathcal{T} \models \varphi(-\infty, u'_b, I\theta)$. Using the soundness of the first premise (choosing $u_b = -\infty$, $u_m = u'_b$, $u_e = u'_e$, $i = I\theta$) we combine $\mathcal{T} \models \varphi(-\infty, u'_b, I\theta)$ and $\mathcal{T} \models \varphi(u'_b, u'_e, I\theta)$ to obtain $\mathcal{T} \models \varphi(-\infty, u'_e, I\theta)$.

■

Appendix E.
Proof of Theorem VI.1

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

(1) $\varphi(-\infty)$

(2) $\forall i, u.\ (\iota(i) \wedge \forall u' < u.\ \varphi(u')) \supset \psi(u, i)$
$(\varphi(u_1) \wedge \neg\varphi(u_2) \wedge (u_1 < u_2)) \supset$

(3) $\exists i, u_3.\ (u_1 < u_3 \leq u_2) \wedge \iota(i) \wedge \neg\psi(u_3, i) \wedge$
$\forall u_4 \in (u_1, u_3).\ \varphi(u_4)$

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 $(u_3 \leq u) \wedge \iota(i) \wedge \neg\psi(u_3, i) \wedge \forall u_4 < u_3.\ \varphi(u_4)$. From $\iota(i)$, $\forall u_4 < u_3.\ \varphi(u_4)$, 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.

[3]Note that the depth (not RES-depth) of the derivation of $\cdot; \Gamma; \cdot \vdash \{e\}\langle u_b, u_e, i\rangle\varphi(u_b, u_e, i)$ may be more than the depth of any of the premises 1–4. Hence this step will not work if we induct merely on the depths given derivations.

Model-specific axioms

(Mem) $(\texttt{Mem}(c, v) \mathbin{@} u \wedge \texttt{Mem}(c, v') \mathbin{@} u' \wedge (u < u') \wedge (v \neq v')) \supset \exists i, u_w \in (u, u').\ \texttt{Write}(i, c, v') \mathbin{@} u_w$

(Receive) $\texttt{Recv}(i, j, v) \mathbin{@} u \supset \exists j', u'.\ (u' < u) \wedge \texttt{Send}(j', i, j, v) \mathbin{@} u'$

(New1) $(\texttt{New}(i, n) \mathbin{@} u \wedge \texttt{Recv}(j, j', v) \mathbin{@} u' \wedge n \in v) \supset u < u'$

(New2) $(\texttt{New}(i, n) \mathbin{@} u \wedge \texttt{Send}(i, j, i', v) \mathbin{@} u' \wedge n \in v) \supset u < u'$

(New3) $(\texttt{New}(i, n) \mathbin{@} u \wedge \texttt{New}(i', n) \mathbin{@} u') \supset ((i = i') \wedge (u = u'))$

(SendU) $(\texttt{Send}(i_1, j, i_2, v) \mathbin{@} u \wedge \texttt{Send}(i'_1, j', i'_2, v') \mathbin{@} u) \supset (v = v')$

Example-specific assumptions

Let $\mathcal{F} = \{\texttt{send\_auth}, \texttt{write\_acc}, \texttt{read\_acc}, \texttt{recv\_i}, \texttt{get\_i}\}$

(Untrust) $\forall i.\ \texttt{Confined}(i, \mathcal{F})$
(Trust1) $\texttt{HonestThread}(Agg, \{\texttt{agg\_body}\})$
(Trust2) $\texttt{HonestThread}(Bank1, \{\texttt{bank1\_body}\})$
(Trust3) $\texttt{HonestThread}(Bank2, \{\texttt{bank2\_body}\})$
(Start1) $\texttt{Start}(Agg) \mathbin{@} -\infty$
(Start2) $\texttt{Start}(Bank1) \mathbin{@} -\infty$
(Start3) $\texttt{Start}(Bank2) \mathbin{@} -\infty$

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

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

$$
\begin{aligned}
&(\texttt{Mem}(total\_box, v') \mathbin{@} u_1 \wedge \\
&\ \texttt{Mem}(total\_box, v) \mathbin{@} u_2 \wedge \\
&\ (u1 < u2) \wedge (v' \neq v)) \supset \\
&\ \exists u_{ru}, u_{up}, u_{p1}, u_{p2}. \\
&\ \exists u\_pass, page1, page2, s. \\
&\quad (u_{ru} < u_{up} < u_{p1}, u_{p2} < u_2) \wedge \\
&\quad \texttt{Recv}(Agg, User, s) \mathbin{@} u_{ru} \wedge \\
&\quad \texttt{Read}(Agg, pass\_box, u\_pass) \mathbin{@} u_{up} \wedge \\
&\quad \texttt{Get}(Bank1, \texttt{bank1.com/balance}, u\_pass, page1) \mathbin{@} u_{p1} \wedge \\
&\quad \texttt{Get}(Bank2, \texttt{bank2.com/balance}, u\_pass, page2) \mathbin{@} u_{p2} \wedge \\
&\quad v = \texttt{parse\_balance}(page1) + \texttt{parse\_balance}(page2)
\end{aligned}
$$

We start from the assumption:

$$\begin{array}{l}(\texttt{Mem}(total\_box, v') \ @ \ u_1 \ \wedge \\ \ \texttt{Mem}(total\_box, v) \ @ \ u_2 \ \wedge \\ \ (u1 < u2) \wedge (v' \neq v))\end{array} \tag{1}$$

and seek to show that:

$$\begin{array}{l}\exists u_{ru}, u_{up}, u_{p1}, u_{p2}. \\ \exists u\_pass, page1, page2, s. \\ \ (u_{ru} < u_{up} < u_{p1}, u_{p2} < u_2) \wedge \\ \ \texttt{Recv}(Agg, User, s) \ @ \ u_{ru} \ \wedge \\ \ \texttt{Read}(Agg, pass\_box, u\_pass) \ @ \ u_{up} \ \wedge \\ \ \texttt{Get}(Bank1, \texttt{bank1.com/balance}, u\_pass, page1) \ @ \ u_{p1} \ \wedge \\ \ \texttt{Get}(Bank2, \texttt{bank2.com/balance}, u\_pass, page2) \ @ \ u_{p2} \ \wedge \\ \ v = \texttt{parse\_balance}(page1) + \texttt{parse\_balance}(page2)\end{array}$$

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

$$\exists i, u_w \in (u_1, u_2]. \ \texttt{Write}(i, total\_box, v) \ @ \ u_w \tag{2}$$

We wish to show that the thread i in (2) is 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 Agg to write to cell $total\_box$. Formally, we prove the following using the rules of Figure 6.

$$\begin{array}{l}\forall f \in \mathcal{F}. \ (\{f\}\langle y, u_b, u_e, i\rangle \\ \qquad \forall v, u. \ ((u_b < u \leq u_e) \wedge \texttt{Write}(i, total\_box, v) \ @ \ u) \\ \qquad\qquad \supset (i = Agg))\end{array} \tag{3}$$

$$\begin{array}{l}\forall f \in \mathcal{F}. \ ([f]\langle y, u_b, u_e, i, x\rangle \\ \qquad \forall v, u. \ ((u_b < u \leq u_e) \wedge \texttt{Write}(i, total\_box, v) \ @ \ u) \\ \qquad\qquad \supset (i = Agg))\end{array} \tag{4}$$

Further, it is easy to see that the invariant $\varphi(u_b, u_e, i) = \forall v, u. \ ((u_b < u \leq u_e) \wedge \texttt{Write}(i, total\_box, v) \ @ \ u) \supset (i = Agg)$ is compositional. Hence applying the rule (RES) to (3), (4), and assumption (Untrust), we obtain (for a thread parameter i):

$$\begin{array}{l}\forall u_e. \ \forall v, u. \ ((-\infty < u \leq u_e) \wedge \texttt{Write}(i, total\_box, v) \ @ \ u) \\ \qquad \supset (i = Agg)\end{array}$$

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

$$\forall i, v, u. \ \texttt{Write}(i, total\_box, v) \ @ \ u \supset (i = Agg) \tag{5}$$

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

$$\exists u_w \in (u_1, u_2]. \ \texttt{Write}(Agg, total\_box, v) \ @ \ u_w \tag{6}$$

We continue our reasoning by examining the program of Agg, which, by assumption (Trust1), we already know to be `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 `agg_loop` and also other assertions about `send_auth`, `write_acc` and `read_acc`, which we omit here.

$$\begin{array}{l}\{\texttt{agg\_body}\}\langle u_b, u_e, i\rangle \\ \ \forall v, u. \ ((u_b < u \leq u_e) \wedge \texttt{Write}(i, total\_box, v) \ @ \ u) \supset \\ \ \ \exists u_{ru}, u_{up}, u_n, u_{s1}, u_{s2}, u_1', u_2'. \ \exists u\_pass, nonce, v_1, v_2, s. \\ \ \ (u_{ru} < u_{up} < u_n < u_{s1} < u_{s2} < u_1' < u_2' < u) \wedge \\ \ \ \texttt{Recv}(i, User, s) \ @ \ u_{ru} \ \wedge \\ \ \ \texttt{Read}(i, pass\_box, u\_pass) \ @ \ u_{up} \ \wedge \\ \ \ \texttt{New}(i, nonce) \ @ \ u_n \ \wedge \\ \ \ \texttt{Send}(i, Bank1, i, (u\_pass, nonce)) \ @ \ u_{s1} \ \wedge \\ \ \ \texttt{Send}(i, Bank2, i, (u\_pass, nonce)) \ @ \ u_{s2} \ \wedge \\ \ \ \texttt{Recv}(i, Bank1, (v_1, nonce)) \ @ \ u_1' \ \wedge \\ \ \ \texttt{Recv}(i, Bank2, (v_2, nonce)) \ @ \ u_2' \ \wedge \\ \ \ v = v1 + v2\end{array} \tag{7}$$

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

$$\begin{array}{l}\forall u_e. \ (u_e > -\infty) \supset \\ \ \forall v, u. \ ((u_b < u \leq u_e) \wedge \texttt{Write}(i, total\_box, v) \ @ \ u) \supset \\ \ \ \exists u_{ru}, u_{up}, u_n, u_{s1}, u_{s2}, u_1', u_2'. \ \exists u\_pass, nonce, v_1, v_2, s. \\ \ \ (u_{ru} < u_{up} < u_n < u_{s1} < u_{s2} < u_1' < u_2' < u) \wedge \\ \ \ \texttt{Recv}(Agg, User, s) \ @ \ u_{ru} \ \wedge \\ \ \ \texttt{Read}(Agg, pass\_box, u\_pass) \ @ \ u_{up} \ \wedge \\ \ \ \texttt{New}(Agg, nonce) \ @ \ u_n \ \wedge \\ \ \ \texttt{Send}(Agg, Bank1, Agg, (u\_pass, nonce)) \ @ \ u_{s1} \ \wedge \\ \ \ \texttt{Send}(Agg, Bank2, Agg, (u\_pass, nonce)) \ @ \ u_{s2} \ \wedge \\ \ \ \texttt{Recv}(Agg, Bank1, (v_1, nonce)) \ @ \ u_1' \ \wedge \\ \ \ \texttt{Recv}(Agg, Bank2, (v_2, nonce)) \ @ \ u_2' \ \wedge \\ \ \ v = v1 + v2\end{array}$$

The instantiation $u_e = \infty$ and some simplification yield:

$$\begin{array}{l}\forall v, u. \ \texttt{Write}(i, total\_box, v) \ @ \ u \supset \\ \ \ \exists u_{ru}, u_{up}, u_n, u_{s1}, u_{s2}, u_1', u_2'. \ \exists u\_pass, nonce, v_1, v_2, s. \\ \ \ (u_{ru} < u_{up} < u_n < u_{s1} < u_{s2} < u_1' < u_2' < u) \wedge \\ \ \ \texttt{Recv}(Agg, User, s) \ @ \ u_{ru} \ \wedge \\ \ \ \texttt{Read}(Agg, pass\_box, u\_pass) \ @ \ u_{up} \ \wedge \\ \ \ \texttt{New}(Agg, nonce) \ @ \ u_n \ \wedge \\ \ \ \texttt{Send}(Agg, Bank1, Agg, (u\_pass, nonce)) \ @ \ u_{s1} \ \wedge \\ \ \ \texttt{Send}(Agg, Bank2, Agg, (u\_pass, nonce)) \ @ \ u_{s2} \ \wedge \\ \ \ \texttt{Recv}(Agg, Bank1, (v_1, nonce)) \ @ \ u_1' \ \wedge \\ \ \ \texttt{Recv}(Agg, Bank2, (v_2, nonce)) \ @ \ u_2' \ \wedge \\ \ \ v = v1 + v2\end{array} \tag{8}$$

Combining (6) and (8) gives:

$$\begin{array}{l}\exists u_w. \ (u_w < u_1 \leq u_2). \\ \ \ \exists u_{ru}, u_{up}, u_n, u_{s1}, u_{s2}, u_1', u_2'. \ \exists u\_pass, nonce, v_1, v_2, s. \\ \ \ (u_{ru} < u_{up} < u_n < u_{s1} < u_{s2} < u_1' < u_2' < u_w) \wedge \\ \ \ \texttt{Recv}(Agg, User, s) \ @ \ u_{ru} \ \wedge \\ \ \ \texttt{Read}(Agg, pass\_box, u\_pass) \ @ \ u_{up} \ \wedge \\ \ \ \texttt{New}(Agg, nonce) \ @ \ u_n \ \wedge \\ \ \ \texttt{Send}(Agg, Bank1, Agg, (u\_pass, nonce)) \ @ \ u_{s1} \ \wedge \\ \ \ \texttt{Send}(Agg, Bank2, Agg, (u\_pass, nonce)) \ @ \ u_{s2} \ \wedge \\ \ \ \texttt{Recv}(Agg, Bank1, (v_1, nonce)) \ @ \ u_1' \ \wedge \\ \ \ \texttt{Recv}(Agg, Bank2, (v_2, nonce)) \ @ \ u_2' \ \wedge \\ \ \ v = v1 + v2\end{array} \tag{9}$$

Next, we wish to reason from the following two lines in (9):

$$\texttt{Recv}(Agg, Bank1, (v_1, nonce)) \mathbin{@} u_1'$$
$$\texttt{Recv}(Agg, Bank2, (v_2, nonce)) \mathbin{@} u_2'$$

We wish to show the following:

$$\exists u_1''.(u_1'' < u_1') \wedge \texttt{Send}(Bank1, Agg, Bank1, (v_1, nonce)) \mathbin{@} u_1''$$
$$\exists u_2''.(u_2'' < u_2') \wedge \texttt{Send}(Bank2, Agg, Bank2, (v_2, nonce)) \mathbin{@} u_2''$$

To prove the above, we start by showing that the following holds in our example:

$$\forall u, i, j, i', v.\ \texttt{Send}(i, j, i', v) \mathbin{@} u \supset (i = i')$$

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.

$$\forall f \in \mathcal{F}.\ (\{f\}\langle y, u_b, u_e, i\rangle \\ \forall u, j, i'.\ ((u_b < u \leq u_e) \wedge \texttt{Send}(i, j, i', v) \mathbin{@} u) \supset (i = i')) \tag{10}$$

$$\forall f \in \mathcal{F}.\ ([f]\langle y, u_b, u_e, i, x\rangle \\ \forall u, j, i'.\ ((u_b < u \leq u_e) \wedge \texttt{Send}(i, j, i', v) \mathbin{@} u) \supset (i = i')) \tag{11}$$

Observe that the invariant $\varphi(u_b, u_e, i) = \forall u, j, i'.\ ((u_b < u \leq u_e) \wedge \texttt{Send}(i, j, i', v) \mathbin{@} u) \supset (i = i')$ is compositional. Hence, applying the rule (RES) to (10), (11), and assumption (Untrust), we obtain the following for any thread parameter i:

$$\forall u_e.\ \forall u, j, i'.\ ((-\infty < u \leq u_e) \wedge \texttt{Send}(i, j, i', v) \mathbin{@} u) \supset (i = i')$$

Instantiating with $u_e = \infty$, simplifying, and quantifying over the parameter i, we get:

$$\forall i, u, j, i'.\ \texttt{Send}(i, j, i', v) \mathbin{@} u \supset (i = i') \tag{12}$$

Combining with axiom (Receive) yields:

$$\forall i, j, v, u.\ \texttt{Recv}(i, j, v) \mathbin{@} u \supset \exists u'.\ (u' < u) \wedge \texttt{Send}(j, i, j, v) \mathbin{@} u' \tag{13}$$

From (9) and (13), we get the following facts in the context of the existential quantifiers in (9):

$$\exists u_1''.(u_1'' < u_1') \wedge \texttt{Send}(Bank1, Agg, Bank1, (v_1, nonce)) \mathbin{@} u_1''$$
$$\exists u_2''.(u_2'' < u_2') \wedge \texttt{Send}(Bank2, Agg, Bank2, (v_2, nonce)) \mathbin{@} u_2'' \tag{14}$$

Next, we would like to reason about the threads $Bank1$ and $Bank2$ from their programs, which, owing to assumptions (Trust2) and (Trust3), we know to be `bank1_body` and `bank2_body`, respectively. We show here the reasoning about $Bank1$; the reasoning about $Bank2$ is similar. Using the rules of Figure 6, we prove the following invariant:

$$\{\texttt{bank1\_body}\}\ \langle u_b, u_e, i\rangle \\ \forall u, j, i', x.\ (u_b < u \leq u_e) \supset \\ \texttt{Send}(i, j, i', x) \mathbin{@} u \supset \\ \exists u_{r1}, u_{p1}.\ (u_{r1} < u_{p1} < u) \wedge \\ \exists v_1', nonce_1', u\_pass_1'. \\ \texttt{Recv}(i, Agg, (u\_pass_1', nonce_1')) \mathbin{@} u_{r1} \wedge \\ \texttt{Get}(i, \texttt{bank1.com/balance}, u\_pass_1', v_1') \mathbin{@} u_{p1} \wedge \\ x = (\texttt{parse\_balance}(v_1'), nonce_1') \tag{15}$$

Applying the rule (HONTH) to assumptions (Trust2) and (Start2) and (15), we obtain:

$$\forall u_e.\ (u_e > -\infty) \supset \\ \forall u, j, i', x.\ (u_b < u \leq u_e) \supset \\ \texttt{Send}(Bank1, j, i', x) \mathbin{@} u \supset \\ \exists u_{r1}, u_{p1}.\ (u_{r1} < u_{p1} < u) \wedge \\ \exists v_1', nonce_1', u\_pass_1'. \\ \texttt{Recv}(Bank1, Agg, (u\_pass_1', nonce_1')) \mathbin{@} u_{r1} \wedge \\ \texttt{Get}(Bank1, \texttt{bank1.com/balance}, u\_pass_1', v_1') \mathbin{@} u_{p1} \wedge \\ x = (\texttt{parse\_balance}(v_1'), nonce_1')$$

Instantiating with $u_e = \infty$, $u = u_1''$, $j = Agg$, $i' = Bank1$, $x = (v_1, nonce)$, and simplifying, we get:

$$\texttt{Send}(Bank1, Agg, Bank1, (v_1, nonce)) \mathbin{@} u_1'' \supset \\ \exists u_{r1}, u_{p1}.\ (u_{r1} < u_{p1} < u_1'') \wedge \\ \exists v_1', nonce_1', u\_pass_1'. \\ \texttt{Recv}(Bank1, Agg, (u\_pass_1', nonce_1')) \mathbin{@} u_{r1} \wedge \\ \texttt{Get}(Bank1, \texttt{bank1.com/balance}, u\_pass_1', v_1') \mathbin{@} u_{p1} \wedge \\ (v_1, nonce) = (\texttt{parse\_balance}(v_1'), nonce_1')$$

Combining with (14) we get:

$$\exists u_{r1}, u_{p1}, u_1''.\ (u_{r1} < u_{p1} < u_1'' < u_1') \wedge \\ \exists v_1', nonce_1', u\_pass_1'. \\ \texttt{Recv}(Bank1, Agg, (u\_pass_1', nonce_1')) \mathbin{@} u_{r1} \wedge \\ \texttt{Get}(Bank1, \texttt{bank1.com/balance}, u\_pass_1', v_1') \mathbin{@} u_{p1} \wedge \\ (v_1, nonce) = (\texttt{parse\_balance}(v_1'), nonce_1')$$

Simplification using the last line yields:

$$\exists u_{r1}, u_{p1}, u_1''.\ (u_{r1} < u_{p1} < u_1'' < u_1') \wedge \\ \exists v_1', u\_pass_1'. \\ \texttt{Recv}(Bank1, Agg, (u\_pass_1', nonce)) \mathbin{@} u_{r1} \wedge \\ \texttt{Get}(Bank1, \texttt{bank1.com/balance}, u\_pass_1', v_1') \mathbin{@} u_{p1} \wedge \\ v_1 = \texttt{parse\_balance}(v_1')$$

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

$$\begin{aligned}
&\exists u_w.\ (u_w < u_1 \leq u_2).\\
&\quad \exists u_{ru}, u_{up}, u_n, u_{s1}, u_{s2}, u_1', u_2', u_{r1}, u_{p1}, u_{r2}, u_{p2}.\\
&\quad \exists u_pass, nonce, v_1, v_2, s, u_pass_1', u_pass_2', v_1', v_2'.\\
&\quad\ (u_{ru} < u_{up} < u_n < u_{s1} < u_{s2} < u_1' < u_2' < u_w) \wedge\\
&\quad\ (u_{r1} < u_{p1} < u_1') \wedge (u_{r2} < u_{p2} < u_2') \wedge\\
&\quad\ \texttt{Recv}(Agg, User, s) \mathbin{@} u_{ru} \wedge\\
&\quad\ \texttt{Read}(Agg, pass_box, u_pass) \mathbin{@} u_{up} \wedge\\
&\quad\ \texttt{New}(Agg, nonce) \mathbin{@} u_n \wedge\\
&\quad\ \texttt{Send}(Agg, Bank1, Agg, (u_pass, nonce)) \mathbin{@} u_{s1} \wedge\\
&\quad\ \texttt{Send}(Agg, Bank2, Agg, (u_pass, nonce)) \mathbin{@} u_{s2} \wedge\\
&\quad\ \texttt{Recv}(Agg, Bank1, (v_1, nonce)) \mathbin{@} u_1' \wedge\\
&\quad\ \texttt{Recv}(Agg, Bank2, (v_2, nonce)) \mathbin{@} u_2' \wedge\\
&\quad\ v = v1 + v2 \wedge\\
&\quad\ \texttt{Recv}(Bank1, Agg, (u_pass_1', nonce)) \mathbin{@} u_{r1} \wedge\\
&\quad\ \texttt{Get}(Bank1, \texttt{bank1.com/balance}, u_pass_1', v_1') \mathbin{@} u_{p1} \wedge\\
&\quad\ v_1 = \texttt{parse_balance}(v_1') \wedge\\
&\quad\ \texttt{Recv}(Bank2, Agg, (u_pass_2', nonce)) \mathbin{@} u_{r2} \wedge\\
&\quad\ \texttt{Get}(Bank2, \texttt{bank2.com/balance}, u_pass_2', v_2') \mathbin{@} u_{p2} \wedge\\
&\quad\ v_2 = \texttt{parse_balance}(v_2')
\end{aligned} \tag{16}$$

Simplifying using the lines $v_1 = \texttt{parse\_balance}(v_1')$ and $v_2 = \texttt{parse\_balance}(v_2')$, we get:

$$\begin{aligned}
&\exists u_w.\ (u_w < u_1 \leq u_2).\\
&\quad \exists u_{ru}, u_{up}, u_n, u_{s1}, u_{s2}, u_1', u_2', u_{r1}, u_{p1}, u_{r2}, u_{p2}.\\
&\quad \exists u_pass, nonce, s, u_pass_1', u_pass_2', v_1', v_2'.\\
&\quad\ (u_{ru} < u_{up} < u_n < u_{s1} < u_{s2} < u_1' < u_2' < u_w) \wedge\\
&\quad\ (u_{r1} < u_{p1} < u_1') \wedge (u_{r2} < u_{p2} < u_2') \wedge\\
&\quad\ \texttt{Recv}(Agg, User, s) \mathbin{@} u_{ru} \wedge\\
&\quad\ \texttt{Read}(Agg, pass_box, u_pass) \mathbin{@} u_{up} \wedge\\
&\quad\ \texttt{New}(Agg, nonce) \mathbin{@} u_n \wedge\\
&\quad\ \texttt{Send}(Agg, Bank1, Agg, (u_pass, nonce)) \mathbin{@} u_{s1} \wedge\\
&\quad\ \texttt{Send}(Agg, Bank2, Agg, (u_pass, nonce)) \mathbin{@} u_{s2} \wedge\\
&\quad\ \texttt{Recv}(Agg, Bank1, (\texttt{parse_balance}(v_1'), nonce)) \mathbin{@} u_1' \wedge\\
&\quad\ \texttt{Recv}(Agg, Bank2, (\texttt{parse_balance}(v_2'), nonce)) \mathbin{@} u_2' \wedge\\
&\quad\ v = \texttt{parse_balance}(v_1') + \texttt{parse_balance}(v_2') \wedge\\
&\quad\ \texttt{Recv}(Bank1, Agg, (u_pass_1', nonce)) \mathbin{@} u_{r1} \wedge\\
&\quad\ \texttt{Get}(Bank1, \texttt{bank1.com/balance}, u_pass_1', v_1') \mathbin{@} u_{p1} \wedge\\
&\quad\ \texttt{Recv}(Bank2, Agg, (u_pass_2', nonce)) \mathbin{@} u_{r2} \wedge\\
&\quad\ \texttt{Get}(Bank2, \texttt{bank2.com/balance}, u_pass_2', v_2') \mathbin{@} u_{p2}
\end{aligned} \tag{17}$$

Next, observe that we have the facts $\texttt{New}(Agg, nonce) \mathbin{@} u_n$ and $\texttt{Recv}(Bank1, Agg, (u\_pass_1', nonce)) \mathbin{@} u_{r1}$ in the above formula. From axiom (New1), it follows that $u_n < u_{r1}$. Similarly, we may also derive that $u_n < u_{r2}$. Incorporating these into (17) (see the line marked $*$ below), we get:

$$\begin{aligned}
&\exists u_w.\ (u_w < u_1 \leq u_2).\\
&\quad \exists u_{ru}, u_{up}, u_n, u_{s1}, u_{s2}, u_1', u_2', u_{r1}, u_{p1}, u_{r2}, u_{p2}.\\
&\quad \exists u_pass, nonce, s, u_pass_1', u_pass_2', v_1', v_2'.\\
&\quad\ (u_{ru} < u_{up} < u_n < u_{s1} < u_{s2} < u_1' < u_2' < u_w) \wedge\\
*\quad &(u_n < u_{r1} < u_{p1} < u_1') \wedge (u_n < u_{r2} < u_{p2} < u_2') \wedge\\
&\quad\ \texttt{Recv}(Agg, User, s) \mathbin{@} u_{ru} \wedge\\
&\quad\ \texttt{Read}(Agg, pass_box, u_pass) \mathbin{@} u_{up} \wedge\\
&\quad\ \texttt{New}(Agg, nonce) \mathbin{@} u_n \wedge\\
&\quad\ \texttt{Send}(Agg, Bank1, Agg, (u_pass, nonce)) \mathbin{@} u_{s1} \wedge\\
&\quad\ \texttt{Send}(Agg, Bank2, Agg, (u_pass, nonce)) \mathbin{@} u_{s2} \wedge\\
&\quad\ \texttt{Recv}(Agg, Bank1, (\texttt{parse_balance}(v_1'), nonce)) \mathbin{@} u_1' \wedge\\
&\quad\ \texttt{Recv}(Agg, Bank2, (\texttt{parse_balance}(v_2'), nonce)) \mathbin{@} u_2' \wedge\\
&\quad\ v = \texttt{parse_balance}(v_1') + \texttt{parse_balance}(v_2') \wedge\\
&\quad\ \texttt{Recv}(Bank1, Agg, (u_pass_1', nonce)) \mathbin{@} u_{r1} \wedge\\
&\quad\ \texttt{Get}(Bank1, \texttt{bank1.com/balance}, u_pass_1', v_1') \mathbin{@} u_{p1} \wedge\\
&\quad\ \texttt{Recv}(Bank2, Agg, (u_pass_2', nonce)) \mathbin{@} u_{r2} \wedge\\
&\quad\ \texttt{Get}(Bank2, \texttt{bank2.com/balance}, u_pass_2', v_2') \mathbin{@} u_{p2}
\end{aligned} \tag{18}$$

By (13) and the fact $\texttt{Recv}(Bank1, Agg, (u\_pass_1', nonce)) \mathbin{@} u_{r1}$ above, we deduce that:

$$\begin{aligned}
&\exists u_{r1}'.\ (u_{r1}' < u_{r1}) \wedge\\
&\qquad \texttt{Send}(Agg, Bank1, Agg, (u_pass_1', nonce)) \mathbin{@} u_{r1}'
\end{aligned} \tag{19}$$

Due to axiom (New2), it follows that $u_n < u_{r1}'$. Incorporating into (19), we get

$$\begin{aligned}
&\exists u_{r1}'.\ (u_n < u_{r1}' < u_{r1}) \wedge\\
&\qquad \texttt{Send}(Agg, Bank1, Agg, (u_pass_1', nonce)) \mathbin{@} u_{r1}'
\end{aligned} \tag{20}$$

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

$$\begin{aligned}
&\exists u_w.\ (u_w < u_1 \leq u_2).\\
&\quad \exists u_{ru}, u_{up}, u_n, u_{s1}, u_{s2}, u_1', u_2', u_{r1}, u_{p1}, u_{r2}, u_{p2}.\\
&\quad \exists u_pass, nonce, s, u_pass_1', u_pass_2', v_1', v_2'.\\
&\quad\ (u_{ru} < u_{up} < u_n < u_{s1} < u_{s2} < u_1' < u_2' < u_w) \wedge\\
&\quad\ (u_n < u_{r1}' < u_{r1} < u_{p1} < u_1') \wedge\\
&\quad\ (u_n < u_{r2'} < u_{r2} < u_{p2} < u_2') \wedge\\
&\quad\ \texttt{Recv}(Agg, User, s) \mathbin{@} u_{ru} \wedge\\
&\quad\ \texttt{Read}(Agg, pass_box, u_pass) \mathbin{@} u_{up} \wedge\\
&\quad\ \texttt{New}(Agg, nonce) \mathbin{@} u_n \wedge\\
&\quad\ \texttt{Send}(Agg, Bank1, Agg, (u_pass, nonce)) \mathbin{@} u_{s1} \wedge\\
&\quad\ \texttt{Send}(Agg, Bank2, Agg, (u_pass, nonce)) \mathbin{@} u_{s2} \wedge\\
&\quad\ \texttt{Recv}(Agg, Bank1, (\texttt{parse_balance}(v_1'), nonce)) \mathbin{@} u_1' \wedge\\
&\quad\ \texttt{Recv}(Agg, Bank2, (\texttt{parse_balance}(v_2'), nonce)) \mathbin{@} u_2' \wedge\\
&\quad\ v = \texttt{parse_balance}(v_1') + \texttt{parse_balance}(v_2') \wedge\\
&\quad\ \texttt{Send}(Agg, Bank1, Agg, (u_pass_1', nonce)) \mathbin{@} u_{r1}' \wedge\\
&\quad\ \texttt{Recv}(Bank1, Agg, (u_pass_1', nonce)) \mathbin{@} u_{r1} \wedge\\
&\quad\ \texttt{Get}(Bank1, \texttt{bank1.com/balance}, u_pass_1', v_1') \mathbin{@} u_{p1} \wedge\\
&\quad\ \texttt{Send}(Agg, Bank2, Agg, (u_pass_2', nonce)) \mathbin{@} u_{r2}' \wedge\\
&\quad\ \texttt{Recv}(Bank2, Agg, (u_pass_2', nonce)) \mathbin{@} u_{r2} \wedge\\
&\quad\ \texttt{Get}(Bank2, \texttt{bank2.com/balance}, u_pass_2', v_2') \mathbin{@} u_{p2}
\end{aligned} \tag{21}$$

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 $\texttt{Send}(Agg, Bank1, Agg, (u\_pass, nonce)) \mathbin{@} u_{s1}$ and $\texttt{Send}(Agg, Bank1, Agg, (u\_pass_1', nonce)) \mathbin{@} u_{r1}'$,

we easily derive $u\_pass'_1 = u\_pass$. Proving that the aggregator never sends out the same message twice requires some care. Because `agg_body` is a loop, many invariants will not compose with themselves. We show that:

$$\begin{array}{l}\{\texttt{agg\_body}\}\langle u_b, u_e, i\rangle \\ \forall u, v, i', j.\ ((u_b < u \leq u_e) \wedge \texttt{Send}(i, j, i', v)\ @\ u) \supset \\ \quad \exists n, x, u_n.\ (u_b < u_n < u) \wedge \texttt{New}(i, n)\ @\ u_n \wedge (v = (x, n)) \wedge \\ \qquad \forall x', i'', u' \in (u_b, u).\neg(\texttt{Send}(i, j, i'', (x', n))\ @\ u')\end{array} \tag{22}$$

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

$$\begin{array}{l}\forall u_e.\ (u_e > -\infty) \supset \\ \forall u, v, i', j.\ ((-\infty < u \leq u_e) \wedge \texttt{Send}(Agg, j, i', v)\ @\ u) \supset \\ \quad \exists n, x, u_n.\ (-\infty < u_n < u) \wedge \texttt{New}(Agg, n)\ @\ u_n \wedge \\ \qquad (v = (x, n)) \wedge \\ \qquad \forall x', i'', u' \in (-\infty, u). \\ \qquad\quad \neg(\texttt{Send}(Agg, j, i'', (x', n))\ @\ u')\end{array}$$

Instantiating with $u_e = \infty$ and simplifying, we get:

$$\begin{array}{l}\forall u, v, i', j.\ \texttt{Send}(Agg, j, i', v)\ @\ u \supset \\ \quad \exists n, x, u_n.\ (u_n < u) \wedge \texttt{New}(Agg, n)\ @\ u_n \wedge \\ \qquad (v = (x, n)) \wedge \\ \qquad \forall x', i'', u' < u. \\ \qquad\quad \neg(\texttt{Send}(Agg, j, i'', (x', n))\ @\ u')\end{array}$$

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

$$\begin{array}{l}\forall u, u', u_n, x, x', n, j, i', i''. \\ \quad (\texttt{New}(Agg, n)\ @\ u_n \wedge \texttt{Send}(Agg, j, i', (x, n))\ @\ u \wedge \\ \quad \texttt{Send}(Agg, j, i'', (x', n))\ @\ u') \supset (u = u')\end{array}$$

Using axiom (SendU), we derive:

$$\begin{array}{l}\forall u, u', u_n, x, x', n, j, i', i''. \\ \quad (\texttt{New}(Agg, n)\ @\ u_n \wedge \texttt{Send}(Agg, j, i', (x, n))\ @\ u \wedge \\ \quad \texttt{Send}(Agg, j, i'', (x', n))\ @\ u') \supset (x = x')\end{array} \tag{23}$$

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

$$\begin{array}{l}\exists u_w.\ (u_w < u_1 \leq u_2). \\ \quad \exists u_{ru}, u_{up}, u_n, u_{s1}, u_{s2}, u'_1, u'_2, u_{r1}, u_{p1}, u_{r2}, u_{p2}. \\ \quad \exists u\_pass, nonce, s, v'_1, v'_2. \\ \quad (u_{ru} < u_{up} < u_n < u_{s1} < u_{s2} < u'_1 < u'_2 < u_w) \wedge \\ \quad (u_n < u'_{r1} < u_{r1} < u_{p1} < u'_1) \wedge \\ \quad (u_n < u_{r2'} < u_{r2} < u_{p2} < u'_2) \wedge \\ \quad \texttt{Recv}(Agg, User, s)\ @\ u_{ru} \wedge \\ \quad \texttt{Read}(Agg, pass\_box, u\_pass)\ @\ u_{up} \wedge \\ \quad \texttt{New}(Agg, nonce)\ @\ u_n \wedge \\ \quad \texttt{Send}(Agg, Bank1, Agg, (u\_pass, nonce))\ @\ u_{s1} \wedge \\ \quad \texttt{Send}(Agg, Bank2, Agg, (u\_pass, nonce))\ @\ u_{s2} \wedge \\ \quad \texttt{Recv}(Agg, Bank1, (\texttt{parse\_balance}(v'_1), nonce))\ @\ u'_1 \wedge \\ \quad \texttt{Recv}(Agg, Bank2, (\texttt{parse\_balance}(v'_2), nonce))\ @\ u'_2 \wedge \\ \quad v = \texttt{parse\_balance}(v'_1) + \texttt{parse\_balance}(v'_2) \wedge \\ \quad \texttt{Send}(Agg, Bank1, Agg, (u\_pass, nonce))\ @\ u'_{r1} \wedge \\ \quad \texttt{Recv}(Bank1, Agg, (u\_pass'_1, nonce))\ @\ u_{r1} \wedge \\ \quad \texttt{Get}(Bank1, \texttt{bank1.com/balance}, u\_pass, v'_1)\ @\ u_{p1} \wedge \\ \quad \texttt{Send}(Agg, Bank2, Agg, (u\_pass, nonce))\ @\ u'_{r2} \wedge \\ \quad \texttt{Recv}(Bank2, Agg, (u\_pass'_2, nonce))\ @\ u_{r2} \wedge \\ \quad \texttt{Get}(Bank2, \texttt{bank2.com/balance}, u\_pass, v'_2)\ @\ u_{p2}\end{array}$$

Eliminating some unnecessary lines, we obtain:

$$\begin{array}{l}\exists u_{ru}, u_{up}, u_{p1}, u_{p2}. \\ \exists u\_pass, nonce, s, v'_1, v'_2. \\ (u_{ru} < u_{up} < u_{p1}, u_{p2} < u_2) \wedge \\ \texttt{Recv}(Agg, User, s)\ @\ u_{ru} \wedge \\ \texttt{Read}(Agg, pass\_box, u\_pass)\ @\ u_{up} \wedge \\ \texttt{Get}(Bank1, \texttt{bank1.com/balance}, u\_pass, v'_1)\ @\ u_{p1} \wedge \\ \texttt{Get}(Bank2, \texttt{bank2.com/balance}, u\_pass, v'_2)\ @\ u_{p2} \wedge \\ v = \texttt{parse\_balance}(v'_1) + \texttt{parse\_balance}(v'_2)\end{array}$$

Renaming $v'_1 \rightarrow page1$ and $v'_2 \rightarrow page2$, we obtain the required property:

$$\begin{array}{l}\exists u_{ru}, u_{up}, u_{p1}, u_{p2}. \\ \exists u\_pass, nonce, s, page1, page2. \\ (u_{ru} < u_{up} < u_{p1}, u_{p2} < u_2) \wedge \\ \texttt{Recv}(Agg, User, s)\ @\ u_{ru} \wedge \\ \texttt{Read}(Agg, pass\_box, u\_pass)\ @\ u_{up} \wedge \\ \texttt{Get}(Bank1, \texttt{bank1.com/balance}, u\_pass, page1)\ @\ u_{p1} \wedge \\ \texttt{Get}(Bank2, \texttt{bank2.com/balance}, u\_pass, page2)\ @\ u_{p2} \wedge \\ v = \texttt{parse\_balance}(page1) + \texttt{parse\_balance}(page2)\end{array}$$

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 $\texttt{Send}(i, v)$ @ u means that thread i sent message v at time u whereas $\texttt{Recv}(i, v)$ @ 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 $\texttt{Owner}(i, t_p)$ 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 $\texttt{New}(i, n)$ @ 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 $\texttt{Flag}(i, t)$ @ 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 $\texttt{Has}(i, v)$ @ 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: $\texttt{SafeMsg}(v, s, \mathcal{K})$, which means that each occurrence of the atomic term s in the term v is protected through encryption with a key in the set $\mathcal{K}$. For precise, inductive definitions of $\texttt{Has}(i, v)$ and $\texttt{SafeMsg}(v, s, \mathcal{K})$, we refer the reader to the prior work on analysis of Kerberos V5 in PCL [27].

[4] In 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.

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 $\texttt{if}(v = v', e, \texttt{halt})$, where `halt` = `infloop()` for `infloop()` $\triangleq$ `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_p, K_p, T_p, and S_p 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. $\mathcal{K}$ denotes a set of keys and $\mathcal{S}$ denotes a set of principals. Informally, $\texttt{Honest}(P_p)$ means that each thread owned by P_p executes one of the programs from Figure 9, $\texttt{OwnedIn}(i, \mathcal{S})$ means that the owner of thread i lies in the set $\mathcal{S}$, $\texttt{OrigRes}(s, \mathcal{S})$ means that the principal who created the term s lies in the set $\mathcal{S}$, $\texttt{KeyRes}(\mathcal{K}, \mathcal{S})$ means that each key in the set $\mathcal{K}$ is known only to principals in $\mathcal{S}$, $\texttt{KORes}(s, \mathcal{K}, \mathcal{S})$ combines the previous two predicates, $\texttt{SendsSafeMsg}(i, s, \mathcal{K})$ means that if thread i sends s in a message, then all occurrences of s in that message are protected by keys in $\mathcal{K}$, $\texttt{SafeNet}(s, \mathcal{K}, u)$ means that prior to time u, every threads protects s in all messages it sends using keys in $\mathcal{K}$, and $\texttt{SendOut}(i, s)$ 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 `KOHonest`, 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 $\mathcal{S}$.

$$
\begin{array}{l}
\texttt{client} = \\
\quad (K_p, T_p, S_p, t) = \texttt{pick}; \\
\quad n_1 = \texttt{new}; \\
\quad \texttt{send}\ (\texttt{self\_prin}, T_p, n_1); \\
\quad (x, tgt, enc_{kc}) = \texttt{recv}; \\
\quad \texttt{match}\ x, \texttt{self\_prin}; \\
\quad (AKey, n_1', T_p') = \texttt{dec}\ enc_{kc}, k(C_p, K_p); \\
\quad \texttt{match}\ n_1', n_1; \\
\quad \texttt{match}\ T_p', T_p; \\
\\
\quad n_2 = \texttt{new}; \\
\quad enc_{ct} = \texttt{enc}\ AKey, \texttt{self\_prin}; \\
\quad \texttt{send}\ (tgt, enc_{ct}, \texttt{self\_prin}, S_p, n_2); \\
\quad (x, st, enc_{tc}) = \texttt{recv}; \\
\quad \texttt{match}\ x, \texttt{self\_prin}; \\
\quad (SKey, n_2', S_p') = \texttt{dec}\ enc_{tc}, AKey; \\
\quad \texttt{match}\ n_2', n_2; \\
\quad \texttt{match}\ S_p', S_p; \\
\\
\quad enc_{cs} = \texttt{enc}\ SKey, (\texttt{self\_prin}, t); \\
\quad \texttt{send}\ (st, enc_{cs}); \\
\quad enc_{sc} = \texttt{recv}; \\
\quad text_{sc} = \texttt{dec}\ SKey, enc_{sc}; \\
\quad \texttt{match}\ text_{sc}, t; \\
\\
\texttt{kas} = \\
\quad (C_p, T_p, n_1) = \texttt{recv}; \\
\quad AKey = \texttt{new}; \\
\quad \texttt{flag}\ (AKey, C_p, T_p); \\
\quad tgt = \texttt{enc}\ k(T_p, \texttt{self\_prin}), (AKey, C_p); \\
\quad enc_{kc} = \texttt{enc}\ k(C_p, \texttt{self\_prin}), (AKey, n_1, T_p); \\
\quad \texttt{send}\ (C_p, tgt, enc_{kc}); \\
\\
\texttt{tgt} = \\
\quad K_p = \texttt{pick}; \\
\quad (tgt, enc_{ct}, C_p, S_p, n_2) = \texttt{recv}; \\
\quad (AKey, C_p') = \texttt{dec}\ tgt, k(\texttt{self\_prin}, K_p); \\
\quad \texttt{match}\ C_p', C_p; \\
\quad text_{ct} = \texttt{dec}\ enc_{ct}, AKey; \\
\quad \texttt{match}\ text_{ct}, C_p; \\
\quad SKey = \texttt{new}; \\
\quad st = \texttt{enc}\ k(S_p, \texttt{self\_prin}), (SKey, Ch); \\
\quad enc_{tc} = \texttt{enc}\ AKey, (SKey, n_2, S_p); \\
\quad \texttt{send}\ (C_p, st, enc_{tc}); \\
\\
\texttt{server} = \\
\quad T_p = \texttt{pick}; \\
\quad (st, enc_{cs}) = \texttt{recv}; \\
\quad (SKey, C_p) = \texttt{dec}\ st, k(\texttt{self\_prin}, T_p); \\
\quad (C_p', t) = \texttt{dec}\ enc_{cs}, SKey; \\
\quad \texttt{match}\ C_p, C_p'; \\
\quad enc_{sc} = \texttt{enc}\ SKey, t; \\
\quad \texttt{send}\ enc_{sc};
\end{array}
$$

Figure 9: Programs of the four roles in the Kerberos V5 protocol (adapted from [27])

Abbreviations

$$
\begin{array}{rcl}
(\varphi \wedge \psi)\ @\ u & = & (\varphi\ @\ u) \wedge (\psi\ @\ u) \\
(\varphi \vee \psi)\ @\ u & = & (\varphi\ @\ u) \vee (\psi\ @\ u) \\
(\varphi \supset \psi)\ @\ u & = & (\varphi\ @\ u) \supset (\psi\ @\ u) \\
(\neg\varphi)\ @\ u & = & \neg(\varphi\ @\ u) \\
\top\ @\ u & = & \top \\
\bot\ @\ u & = & \bot \\
(\forall x.\varphi)\ @\ u & = & \forall x.\ (\varphi\ @\ u) \\
(\exists x.\varphi)\ @\ u & = & \exists x.\ (\varphi\ @\ u) \\
(\varphi\ @\ u')\ @\ u & = & \varphi\ @\ u' \\
\\
\varphi \circ (u_1, u_2) & = & \forall u.\ (u_1 < u < u_2) \supset (\varphi\ @\ u) \\
\varphi \circ (u_1, u_2] & = & \forall u.\ (u_1 < u \leq u_2) \supset (\varphi\ @\ u) \\
\varphi \circ [u_1, u_2) & = & \forall u.\ (u_1 \leq u < u_2) \supset (\varphi\ @\ u) \\
\varphi \circ [u_1, u_2] & = & \forall u.\ (u_1 \leq u \leq u_2) \supset (\varphi\ @\ u)
\end{array}
$$

Definitions

$$
\begin{array}{l}
\texttt{Honest}(P_p) = \forall i.\ \texttt{Owner}(i, P_p) \supset \\
\qquad \texttt{HonestThread}(i, \{\texttt{client}, \texttt{kas}, \texttt{tgs}, \texttt{server}\}) \\
\texttt{OwnedIn}(i, \mathcal{S}) = \exists i_p.\ \texttt{Owner}(i, i_p) \wedge i_p \in \mathcal{S} \\
\texttt{OrigRes}(s, \mathcal{S}) = \forall i, u.\ \texttt{New}(i, s)\ @\ u \supset \texttt{OwnedIn}(i, \mathcal{S}) \\
\texttt{KeyRes}(\mathcal{K}, \mathcal{S}) = \forall i, k.\ (\texttt{Has}(i, k) \wedge k \in \mathcal{K}) \\
\qquad\qquad\qquad\qquad \supset \texttt{OwnedIn}(i, \mathcal{S}) \\
\texttt{KORes}(s, \mathcal{K}, \mathcal{S}) = \texttt{OrigRes}(s, \mathcal{S}) \wedge \\
\qquad\qquad \forall u.\ \texttt{KeyRes}(\mathcal{K}, \mathcal{S})\ @\ u \\
\texttt{SendsSafeMsg}(i, s, \mathcal{K}) = \texttt{Send}(i, v) \supset \texttt{SafeMsg}(v, s, \mathcal{K}) \\
\texttt{SafeNet}(s, \mathcal{K}, u) = \forall i, u'.\ (u' \leq u) \\
\qquad\qquad \supset \texttt{SendsSafeMsg}(i, s, \mathcal{K})\ @\ u' \\
\texttt{HasOnly}(\mathcal{S}, s) = \forall i.\ \texttt{Has}(i, s) \supset \texttt{OwnedIn}(i, \mathcal{S}) \\
\texttt{SendOut}(i, s) = \exists m.\ \texttt{Send}(i, m) \wedge s \in m
\end{array}
$$

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

Secrecy Property

The property we wish to show is that the key $AKey$ 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:

$$
\begin{array}{rcl}
\varphi_{sec} & = & \forall i, akey, c_p, k_p, t_p, u. \\
& & (\texttt{Flag}(i, (akey, c_p, t_p))\ @\ u \wedge \texttt{Owner}(i, k_p) \wedge \\
& & \texttt{Honest}(k_p) \wedge \texttt{Honest}(c_p) \wedge \texttt{Honest}(t_p)) \\
& & \quad \supset \forall u'.\ \texttt{HasOnly}(\{k_p, c_p, t_p\}, akey)\ @\ u'
\end{array}
$$

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,

Axioms

(Receive) $\texttt{Recv}(i,v) \mathbin{@} u \supset \exists j, u'.\ (u' < u) \wedge \texttt{Send}(j,v) \mathbin{@} u'$

(New3) $(\texttt{New}(i,n) \mathbin{@} u \wedge \texttt{New}(i',n) \mathbin{@} u') \supset ((i = i') \wedge (u = u'))$

(KeyR) $(\texttt{Honest}(I_p) \wedge \texttt{Honest}(J_p)) \supset \texttt{HasOnly}(\{I_p, J_p\}, k(I_p, J_p)) \mathbin{@} u$

(SendN) $\neg\texttt{Send}(i,v) \mathbin{@} -\infty$

(NET) $(\texttt{KORes}(s,\mathcal{K},\mathcal{S}) \wedge \texttt{SafeNet}(s,\mathcal{K},u_1) \wedge \neg\texttt{SafeNet}(s,\mathcal{K},u_2) \wedge (u_1 < u_2)) \supset \exists i, u_3.\ (u_1 < u_3 \leq u_2) \wedge \texttt{OwnedIn}(i,\mathcal{S}) \wedge \neg\texttt{SendsSafeMsg}(i,s,\mathcal{K}) \mathbin{@} u_3 \wedge \forall u_4 \in (u_1, u_3).\ \texttt{SafeNet}(s,\mathcal{K},u_4)$

(POS) $(\texttt{SafeNet}(s,\mathcal{K},u) \wedge \texttt{Has}(i,s) \mathbin{@} u) \supset (\exists u'.\ (u' < u) \wedge \texttt{New}(i,s) \mathbin{@} u') \vee (\exists k.\ (k \in \mathcal{K}) \wedge \texttt{Has}(i,k) \mathbin{@} u)$

Figure 11: Model-specific axioms for secrecy analysis in Kerberos V5

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 φ_{sec}, we choose parameters I_0, $AKey_0$, C_{p0}, K_{p0}, T_{p0}, and U_0 for the variables i, $akey$, c_p, k_p, t_p, and u in the formula φ_{sec} that we want to show and show that:

$$(\texttt{Flag}(I_0, (AKey_0, C_{p0}, T_{p0})) \mathbin{@} U_0 \wedge \texttt{Owner}(I_0, K_{p0}) \wedge \texttt{Honest}(K_{p0}) \wedge \texttt{Honest}(C_{p0}) \wedge \texttt{Honest}(T_{p0})) \supset \forall u'.\ \texttt{HasOnly}(\{K_{p0}, C_{p0}, T_{p0}\}, AKey_0) \mathbin{@} u'$$

Define $\mathcal{S}_0 = \{K_{p0}, C_{p0}, T_{p0}\}$. Then, it suffices to show that

$$\varphi_0 = \forall u'.\ \texttt{HasOnly}(\mathcal{S}_0, AKey_0) \mathbin{@} u'$$

follows from the assumptions

$$\Gamma_0 \quad = \quad \texttt{Flag}(I_0, (AKey_0, C_{p0}, T_{p0})) \mathbin{@} U_0, \texttt{Owner}(I_0, K_{p0}), \texttt{Honest}(K_{p0}), \texttt{Honest}(C_{p0}), \texttt{Honest}(T_{p0}) \tag{24}$$

We now take Γ_0 as our set of assumptions. In the sequel, whenever we derive a modal formula ψ or assertion μ, what we really mean is $\Sigma; \Gamma_0 \vdash \psi$ or $\Sigma; \Gamma_0; \cdot \vdash \mu$ for an appropriate Σ that includes at least I_0, $AKey_0$, C_{p0}, K_{p0}, T_{p0}, and U_0. (What we eventually prove is that $\Sigma; \Gamma_0 \vdash \varphi_0$, from which φ_{sec} follows immediately.)

Preliminary deductions: Let $\mathcal{K}_0 = \{k(T_{p0}, K_{p0}), k(C_{p0}, K_{p0})\}$. Our first step is to prove the following property:

$$\forall i, c_p, t_p, k_p, u, akey.\ (\texttt{Flag}(i, (akey, c_p, t_p)) \mathbin{@} u \wedge \texttt{Owner}(i, k_p) \wedge \texttt{Honest}(k_p)) \supset \exists u' < u.\ \texttt{New}(i, akey) \mathbin{@} u' \wedge \neg\texttt{SendOut}(i, akey) \circ (u', u] \wedge \forall u'.\ \texttt{SendsSafeMsg}(i, akey, \{k(t_p, k_p), k(c_p, k_p)\}) \mathbin{@} u' \tag{25}$$

This property is equivalent to showing that

$$\forall k_p, i.\ (\texttt{Honest}(k_p) \wedge \texttt{Owner}(i, k_p)) \supset \forall c_p, t_p, u, akey.\ \texttt{Flag}(i, (akey, c_p, t_p)) \mathbin{@} u \supset \exists u' < u.\ \texttt{New}(i, akey) \mathbin{@} u' \wedge \neg\texttt{SendOut}(i, akey) \circ (u', u] \wedge \forall u'.\ \texttt{SendsSafeMsg}(i, akey, \{k(t_p, k_p), k(c_p, k_p)\}) \mathbin{@} u'$$

Equivalently, we may assume for parameters i, k_p that $\texttt{Honest}(k_p)$ and $\texttt{Owner}(i, k_p)$ and prove that

$$\forall c_p, t_p, u, akey.\ \texttt{Flag}(i, (akey, c_p, t_p)) \mathbin{@} u \supset \exists u' < u.\ \texttt{New}(i, akey) \mathbin{@} u' \wedge \neg\texttt{SendOut}(i, akey) \circ (u', u] \wedge \forall u'.\ \texttt{SendsSafeMsg}(i, akey, \{k(t_p, k_p), k(c_p, k_p)\}) \mathbin{@} u' \tag{26}$$

We prove the above using the rule (HONTH). We show using the rules of Figure 6 that for $\mathcal{E}_0 = \{\texttt{client}, \texttt{kas}, \texttt{tgs}, \texttt{server}\}$, it is the case that:

$$\forall e \in \mathcal{E}_0.\ (\{e\}\langle u_b, u_e, i\rangle \forall u \in (u_b, u_e].\ \forall c_p, t_p, akey.\ \texttt{Flag}(i, (akey, c_p, t_p)) \mathbin{@} u \supset \exists u' < u.\ \texttt{New}(i, akey) \mathbin{@} u' \wedge \neg\texttt{SendOut}(i, akey) \circ (u', u] \wedge \forall u'.\ \texttt{SendsSafeMsg}(i, akey, \{k(t_p, k_p), k(c_p, k_p)\}) \mathbin{@} u'$$

Since we have already assumed $\texttt{Honest}(k_p)$ and $\texttt{Owner}(i, k_p)$, which imply by definition of $\texttt{Honest}(k_p)$ that $\texttt{HonestThread}(i, \mathcal{E}_0)$, we may apply rule (HONTH) to the above formula to deduce that:

$$\forall u_e.\ (u_e > -\infty) \supset \forall u \in (-\infty, u_e].\ \forall c_p, t_p, akey.\ \texttt{Flag}(i, (akey, c_p, t_p)) \mathbin{@} u \supset \exists u' < u.\ \texttt{New}(i, akey) \mathbin{@} u' \wedge \neg\texttt{SendOut}(i, akey) \circ (u', u] \wedge \forall u'.\ \texttt{SendsSafeMsg}(i, akey, \{k(t_p, k_p), k(c_p, k_p)\}) \mathbin{@} u'$$

Choosing $u_e = \infty$ and simplifying, we obtain (26). Thus (26) holds. Hence (25) also holds. We now instantiate (25) by choosing $i = I_0$, $c_p = C_{p0}$, $t_p = T_{p0}$, $k_p = K_{p0}$, $u = U_0$, and $akey = AKey_0$ to get:

$$(\texttt{Flag}(I_0, (AKey_0, C_{p0}, T_{p0})) \mathbin{@} U_0 \wedge \texttt{Owner}(I_0, K_{p0}) \wedge \texttt{Honest}(K_{p0})) \supset \exists u' < U_0.\ \texttt{New}(I_0, AKey_0) \mathbin{@} u' \wedge \neg\texttt{SendOut}(I_0, AKey_0) \circ (u', U_0] \wedge \forall u'.\ \texttt{SendsSafeMsg}(I_0, AKey_0, \mathcal{K}_0) \mathbin{@} u' \tag{27}$$

The conditions of the implication in (27) are assumed to hold in Γ_0 (see (24)). Therefore, it follows that the consequent of the implication, i.e, the following must also hold:

$$\begin{array}{l}\exists u' < U_0.\ \texttt{New}(I_0, AKey_0) \ @\ u' \wedge \\ \quad \neg\texttt{SendOut}(I_0, AKey_0) \circ (u', U_0] \wedge \\ \quad \forall u'.\ \texttt{SendsSafeMsg}(I_0, AKey_0, \mathcal{K}_0) \ @\ u'\end{array} \tag{28}$$

Next, we wish to prove that $\texttt{OrigRes}(AKey_0, \mathcal{S}_0)$. To do this, we prove the stronger statement: $\forall u, i.\ \texttt{New}(i, AKey_0) \ @\ u \supset i = I_0$. To prove the latter, assume that $\texttt{New}(i, AKey_0) \ @\ u$. By axiom (New3) and $\texttt{New}(I_0, AKey_0) \ @\ u'$ in (28) we obtain $i = I_0$ as required. Hence, $\forall u, i.\ \texttt{New}(i, AKey_0) \ @\ u \supset i = I_0$. This implies $\texttt{OrigRes}(AKey_0, \mathcal{S}_0)$. (Note that $\texttt{OwnedIn}(I_0, \mathcal{S}_0)$ follows from the assumption $\texttt{Owner}(I_0, K_{p0})$ in Γ_0.)

Further the fact $\forall u.\ \texttt{KeyRes}(\mathcal{K}_0, \mathcal{S}_0) \ @\ u$ follows immediately from axiom (KeyR) and the assumptions $\texttt{Honest}(K_{t0})$, $\texttt{Honest}(C_{t0})$, and $\texttt{Honest}(T_{t0})$ in Γ_0. Combining with $\texttt{OrigRes}(AKey_0, \mathcal{S}_0)$, which was derived earlier, we deduce that:

$$\texttt{KORes}(AKey_0, \mathcal{K}_0, \mathcal{S}_0) \tag{29}$$

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 $\forall u.\ \texttt{SafeNet}(AKey_0, \mathcal{K}_0, u)$. In order to apply this technique, we instantiate the framework of Section VI by choosing:

$$\begin{array}{l}\varphi(u) = \texttt{SafeNet}(AKey_0, \mathcal{K}_0, u) \\ \iota(i) = \texttt{OwnedIn}(i, \mathcal{S}_0) \\ \psi(u, i) = \texttt{SendsSafeMsg}(i, AKey_0, \mathcal{K}_0) \ @\ u\end{array}$$

In order to apply the method of Section VI, we must show that the following hold for φ, ι, and ψ as defined above:

(1) $\varphi(-\infty)$

(2) $\forall i, u.\ (\iota(i) \wedge \forall u' < u.\ \varphi(u')) \supset \psi(u, i)$
$(\varphi(u_1) \wedge \neg\varphi(u_2) \wedge (u_1 < u_2)) \supset$

(3) $\exists i, u_3.\ (u_1 < u_3 \leq u_2) \wedge \iota(i) \wedge \neg\psi(u_3, i) \wedge$
$\forall u_4 \in (u_1, u_3).\ \varphi(u_4)$

(1) follows immediately from axiom (SendN) and the definition $\varphi(u) = \texttt{SafeNet}(AKey_0, \mathcal{K}_0, u)$. 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 $s = AKey_0$, $\mathcal{K} = \mathcal{K}_0$, and $\mathcal{S} = \mathcal{S}_0$ to obtain:

$$\begin{array}{l}(\texttt{KORes}(AKey_0, \mathcal{K}_0, \mathcal{S}_0) \wedge \texttt{SafeNet}(AKey_0, \mathcal{K}_0, u_1) \wedge \\ \neg\texttt{SafeNet}(AKey_0, \mathcal{K}_0, u_2) \wedge (u_1 < u_2)) \supset \\ \quad \exists i, u_3.\ (u_1 < u_3 \leq u_2) \wedge \texttt{OwnedIn}(i, \mathcal{S}_0) \wedge \\ \qquad \neg\texttt{SendsSafeMsg}(i, \mathcal{S}_0, \mathcal{K}_0) \ @\ u_3 \wedge \\ \qquad \forall u_4 \in (u_1, u_3).\ \texttt{SafeNet}(AKey_0, \mathcal{K}_0, u_4)\end{array}$$

Since we already know $\texttt{KORes}(AKey_0, \mathcal{K}_0, \mathcal{S}_0)$ from (29), we may eliminate that condition from the above formula to obtain:

$$\begin{array}{l}(\texttt{SafeNet}(AKey_0, \mathcal{K}_0, u_1) \wedge \neg\texttt{SafeNet}(AKey_0, \mathcal{K}_0, u_2) \wedge \\ (u_1 < u_2)) \supset \\ \quad \exists i, u_3.\ (u_1 < u_3 \leq u_2) \wedge \texttt{OwnedIn}(i, \mathcal{S}_0) \wedge \\ \qquad \neg\texttt{SendsSafeMsg}(i, \mathcal{S}_0, \mathcal{K}_0) \ @\ u_3 \wedge \\ \qquad \forall u_4 \in (u_1, u_3).\ \texttt{SafeNet}(AKey_0, \mathcal{K}_0, u_4)\end{array}$$

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

$$\forall u.\ \texttt{SafeNet}(AKey_0, \mathcal{K}_0, u) \tag{30}$$

Next, we fix any time parameter u'_0, and try to show that $\texttt{HasOnly}(\mathcal{S}_0, AKey_0) \ @\ u'_0$. Following the definition of HasOnly assume that for some i, $\texttt{Has}(i, AKey_0) \ @\ u'_0$. It suffices to show that $\texttt{OwnedIn}(i, \mathcal{S}_0)$. From (30), the assumption $\texttt{Has}(i, AKey_0) \ @\ u'_0$, and axiom (POS), we obtain:

$$\begin{array}{l}(\exists u'.\ (u' < u'_0) \wedge \texttt{New}(i, AKey_0) \ @\ u') \vee \\ (\exists k.\ (k \in \mathcal{K}_0) \wedge \texttt{Has}(i, k) \ @\ u'_0)\end{array}$$

We case analyze these two disjuncts. If $\exists u'.\ (u' < u'_0) \wedge \texttt{New}(i, AKey_0) \ @\ u'$, then using axiom (New3) and (28) we obtain $i = I_0$. Since Γ_0 contains the assumption $\texttt{Owner}(I_0, K_{t0})$ and $K_{t0} \in \mathcal{S}_0$, $\texttt{OwnedIn}(i, \mathcal{S}_0)$ follows from definition of OwnedIn.

If $\exists k.\ (k \in \mathcal{K}_0) \wedge \texttt{Has}(i, k) \ @\ u'_0$, then from fact (29), i.e, $\texttt{KORes}(AKey_0, \mathcal{K}_0, \mathcal{S}_0)$, we immediately obtain $\texttt{OwnedIn}(i, \mathcal{S}_0)$.

Since, in both case analyses we obtain $\texttt{OwnedIn}(i, \mathcal{S}_0)$, it follows that $\texttt{HasOnly}(\mathcal{S}_0, AKey_0) \ @\ u'_0$ for any u'_0. Since u'_0 is a parameter, this implies $\forall u'.\ \texttt{HasOnly}(\mathcal{S}_0, AKey_0) \ @\ u'$, which is the φ_0 that we wanted to prove. Hence, φ_{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 $\varphi(u)$, $\iota(i)$, and $\psi(u, i)$, we must show that:

$$\begin{array}{l}\forall i, u.\ (\texttt{OwnedIn}(i, \mathcal{S}_0) \wedge \forall u' < u.\ \texttt{SafeNet}(AKey_0, \mathcal{K}_0, u')) \\ \qquad \supset \texttt{SendsSafeMsg}(i, AKey_0, \mathcal{K}_0) \ @\ u\end{array}$$

This formula is equivalent to

$$\begin{array}{l}\forall i.\ \texttt{OwnedIn}(i, \mathcal{S}_0) \supset \forall u.\ ((\forall u' < u.\ \texttt{SafeNet}(AKey_0, \mathcal{K}_0, u')) \\ \qquad \supset \texttt{SendsSafeMsg}(i, AKey_0, \mathcal{K}_0) \ @\ u)\end{array}$$

To prove this formula, pick a parameter i_0, and assume $\texttt{OwnedIn}(i_0, \mathcal{S}_0)$. It suffices to show $\forall u.\ ((\forall u' < u.\ \texttt{SafeNet}(AKey_0, \mathcal{K}_0, u')) \supset \texttt{SendsSafeMsg}(i_0, AKey_0, \mathcal{K}_0) \ @\ u)$.

The assumption $\texttt{OwnedIn}(i_0, \mathcal{S}_0)$ implies that there is an i_p such that $\texttt{Owner}(i_0, i_p)$ and $i_p \in \mathcal{S}_0$. From the assumptions $\texttt{Honest}(K_{p0})$, $\texttt{Honest}(T_{p0})$, and $\texttt{Honest}(C_{p0})$ in Γ_0, we obtain $\texttt{HonestThread}(i_0, \mathcal{E}_0)$. Hence, if we can prove the assertion below, then by rule (HONTH), we

would obtain $\forall u.\ ((\forall u' < u.\ \texttt{SafeNet}(AKey_0, \mathcal{K}_0, u')) \supset \texttt{SendsSafeMsg}(i_0, AKey_0, \mathcal{K}_0)\ @\ u)$ as required.

$$\forall e \in \mathcal{E}_0.\ (\{e\}\langle u_b, u_e, i\rangle \\ \forall u \in (u_b, u_e].\ (\forall u' < u.\ \texttt{SafeNet}(AKey_0, \mathcal{K}_0, u')) \\ \supset \texttt{SendsSafeMsg}(i, AKey_0, \mathcal{K}_0)\ @\ u)$$

Therefore, we only need show the above assertion for each $e \in \mathcal{E}_0$. So, we must show:

$$\{\texttt{client}\}\langle u_b, u_e, i\rangle \\ \forall u \in (u_b, u_e].\ (\forall u' < u.\ \texttt{SafeNet}(AKey_0, \mathcal{K}_0, u')) \\ \supset \texttt{SendsSafeMsg}(i, AKey_0, \mathcal{K}_0)\ @\ u \tag{31}$$

$$\{\texttt{kas}\}\langle u_b, u_e, i\rangle \\ \forall u \in (u_b, u_e].\ (\forall u' < u.\ \texttt{SafeNet}(AKey_0, \mathcal{K}_0, u')) \\ \supset \texttt{SendsSafeMsg}(i, AKey_0, \mathcal{K}_0)\ @\ u \tag{32}$$

$$\{\texttt{tgs}\}\langle u_b, u_e, i\rangle \\ \forall u \in (u_b, u_e].\ (\forall u' < u.\ \texttt{SafeNet}(AKey_0, \mathcal{K}_0, u')) \\ \supset \texttt{SendsSafeMsg}(i, AKey_0, \mathcal{K}_0)\ @\ u \tag{33}$$

$$\{\texttt{server}\}\langle u_b, u_e, i\rangle \\ \forall u \in (u_b, u_e].\ (\forall u' < u.\ \texttt{SafeNet}(AKey_0, \mathcal{K}_0, u')) \\ \supset \texttt{SendsSafeMsg}(i, AKey_0, \mathcal{K}_0)\ @\ u \tag{34}$$

We prove these formulas one by one.

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

$$\{\texttt{client}\}\langle u_b, u_e, i\rangle \\ \forall u \in (u_b, u_e].\ (\forall u' < u.\ \texttt{SafeNet}(AKey_0, \mathcal{K}_0, u')) \\ \supset \texttt{SendsSafeMsg}(i, AKey_0, \mathcal{K}_0)\ @\ u$$

Using the rules of Figure 6, we show that

$$\{\texttt{client}\}\langle u_b, u_e, i\rangle\ \forall u \in (u_b, u_e].\ \forall x.\ \texttt{Send}(i, x)\ @\ u \supset \\ (\exists t_p, c_p, n_1, u_1.(u_b < u_1 < u) \wedge \\ \texttt{New}(i, n_1)\ @\ u_1 \wedge \\ \neg\texttt{Flag}(i, (n_1, \_, \_)) \circ (u_b, u_e] \wedge \\ x = (c_p, t_p, n_1)) \vee \\ (\exists tgt, c_p, akey, s_p, n_2, u_2, u_r.(u_b < u_r < u_2 < u) \wedge \\ \texttt{New}(i, n_2)\ @\ u_2 \wedge \\ \neg\texttt{Flag}(i, (n_2, \_, \_)) \circ (u_b, u_e] \wedge \\ \texttt{Recv}(i, tgt)\ @\ u_r \wedge \\ x = (tgt, ENC(akey, c_p), c_p, s_p, n_2)) \vee \\ (\exists st, c_p, t, u_r, skey.(u_b < u_r < u) \wedge \\ \texttt{Recv}(i, st)\ @\ u_r \wedge \\ x = (st, ENC(skey, (c_p, t)))) \tag{35}$$

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 $u \in (u_b, u_e]$ that:

$$\forall u' < u.\ \texttt{SafeNet}(AKey_0, \mathcal{K}_0, u') \tag{36}$$

$$\neg\texttt{SendsSafeMsg}(i, AKey_0, \mathcal{K}_0)\ @\ u \tag{37}$$

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

$$\texttt{Send}(i, x)\ @\ u \wedge \neg\texttt{SafeMsg}(x, AKey_0, \mathcal{K}_0) \tag{38}$$

From $\texttt{Send}(i, x)\ @\ u$ 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:

$$\exists t_p, c_p, n_1, u_1.(u_b < u_1 < u) \wedge \\ \texttt{New}(i, n_1)\ @\ u_1 \wedge \\ \neg\texttt{Flag}(i, (n_1, \_, \_)) \circ (u_b, u_e] \wedge \\ x = (c_p, t_p, n_1) \tag{39}$$

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

$$\neg\texttt{SafeMsg}((c_p, t_p, n_1), AKey_0, \mathcal{K}_0) \tag{40}$$

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

$$\neg\texttt{SafeMsg}(c_p, AKey_0, \mathcal{K}_0) \tag{41}$$

$$\neg\texttt{SafeMsg}(t_p, AKey_0, \mathcal{K}_0) \tag{42}$$

$$\neg\texttt{SafeMsg}(n_1, AKey_0, \mathcal{K}_0) \tag{43}$$

However, (41) and (42) cannot hold because c_p and t_p are principals, and, hence, cannot contain $AKey_0$. Therefore, (43) must be true. This forces $n_1 = AKey_0$. From this and (39) we deduce:

$$\texttt{New}(i, AKey_0)\ @\ u_1 \tag{44}$$

We already know from (28) that there is a $u' < U_0$ such that $\texttt{New}(I_0, AKey_0)\ @\ u'$. Using axiom (New3) we deduce:

$$(i = I_0) \wedge (u' = u_1) \tag{45}$$

From (45) and (39) we deduce:

$$(u_b < u' < u) \wedge \neg\texttt{Flag}(I_0, (AKey_0, \_, \_)) \circ (u_b, u_e] \tag{46}$$

From (28) we also know that

$$\texttt{Flag}(I_0, (AKey_0, C_{p0}, T_{p0}))\ @\ U_0 \tag{47}$$

Using (46), (47), and $u' < U_0$ we obtain:

$$U_0 > u_e \tag{48}$$

Since $u' < u \leq u_e$, we also get:

$$u' < u < U_0 \tag{49}$$

Hence from (28) we obtain $\neg\texttt{SendOut}(I_0, AKey_0) \circ (u', u]$ which implies

$$\neg\texttt{SendOut}(I_0, AKey_0) \mathbin{@} u \tag{50}$$

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

Case. The following disjunct holds:

$$\begin{array}{l}\exists tgt, c_p, akey, s_p, n_2, u_2, u_r.(u_b < u_r < u_2 < u) \wedge \\ \quad \texttt{New}(i, n_2) \mathbin{@} u_2 \wedge \\ \quad \neg\texttt{Flag}(i, (n_2, \_, \_)) \circ (u_b, u_e] \wedge \\ \quad \texttt{Recv}(i, tgt) \mathbin{@} u_r \wedge \\ \quad x = (tgt, ENC(akey, c_p), c_p, s_p, n_2)\end{array} \tag{51}$$

The last line of (51) and (38) together yield

$$\neg\texttt{SafeMsg}((tgt, ENC(akey, c_p), c_p, s_p, n_2), AKey_0, \mathcal{K}_0) \tag{52}$$

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

$$\neg\texttt{SafeMsg}(c_p, AKey_0, \mathcal{K}_0) \tag{53}$$

$$\neg\texttt{SafeMsg}(s_p, AKey_0, \mathcal{K}_0) \tag{54}$$

$$\neg\texttt{SafeMsg}(n_2, AKey_0, \mathcal{K}_0) \tag{55}$$

$$\neg\texttt{SafeMsg}(tgt, AKey_0, \mathcal{K}_0) \tag{56}$$

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 u_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:

$$(u_r < u) \wedge \texttt{Recv}(i, tgt) \mathbin{@} u_r \tag{57}$$

Using axiom (Receive), we obtain a u_s and a j such that:

$$(u_s < u_r < u) \wedge \texttt{Send}(j, tgt) \mathbin{@} u_s \tag{58}$$

From (36) and $u_s < u$ in above formula, we also deduce that $\texttt{SendsSafeMsg}(j, AKey_0, \mathcal{K}_0) \mathbin{@} u_s$. So, from (58) we obtain $\texttt{SafeMsg}(tgt, AKey_0, \mathcal{K}_0)$. This contradicts the assumption (56).

Case. The following disjunct holds:

$$\begin{array}{l}\exists st, c_p, t, u_r, skey.(u_b < u_r < u) \wedge \\ \quad \texttt{Recv}(i, st) \mathbin{@} u_r \wedge \\ \quad x = (st, ENC(skey, (c_p, t)))\end{array} \tag{59}$$

The last line of (59) and (38) together yield

$$\neg\texttt{SafeMsg}((st, ENC(skey, (c_p, t))), AKey_0, \mathcal{K}_0) \tag{60}$$

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

$$\neg\texttt{SafeMsg}(c_p, AKey_0, \mathcal{K}_0) \tag{61}$$

$$\neg\texttt{SafeMsg}(t, AKey_0, \mathcal{K}_0) \tag{62}$$

$$\neg\texttt{SafeMsg}(st, AKey_0, \mathcal{K}_0) \tag{63}$$

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:

$$\begin{array}{l}\{\texttt{kas}\}\langle u_b, u_e, i\rangle \\ \forall u \in (u_b, u_e].\ (\forall u' < u.\ \texttt{SafeNet}(AKey_0, \mathcal{K}_0, u')) \\ \qquad\qquad \supset \texttt{SendsSafeMsg}(i, AKey_0, \mathcal{K}_0) \mathbin{@} u\end{array}$$

Using the rules of Figure 6, we show that

$$\begin{array}{l}\{\texttt{kas}\}\langle u_b, u_e, i\rangle\ \forall u \in (u_b, u_e].\ \forall x.\ \texttt{Send}(i, x) \mathbin{@} u \supset \\ \quad (\exists c_p, t_p, u_r, u_n, u_f, a, k_p, n_1. \\ \qquad (u_b < u_r < u_n < u_f < u \leq u_e) \wedge \texttt{Owner}(i, k_p) \wedge \\ \qquad \texttt{Recv}(i, (c_p, t_p, n_1)) \mathbin{@} u_r \wedge \\ \qquad \texttt{New}(i, a) \mathbin{@} u_n \wedge \\ \qquad \texttt{Flag}(i, (a, c_p, t_p)) \mathbin{@} u_f \wedge \\ \qquad x = (c_p, ENC(k(t_p, k_p), (a, c_p)), \\ \qquad\qquad ENC(k(c_p, k_p), (a, n_1, t_p))))\end{array} \tag{64}$$

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:

$$\forall u' < u.\ \texttt{SafeNet}(AKey_0, \mathcal{K}_0, u') \tag{65}$$

$$\neg\texttt{SendsSafeMsg}(i, AKey_0, \mathcal{K}_0) \mathbin{@} u \tag{66}$$

Expanding the definition of SendsSafeMsg in (66), we further obtain that there is a message x such that:

$$\texttt{Send}(i, x) \mathbin{@} u \wedge \neg\texttt{SafeMsg}(x, AKey_0, \mathcal{K}_0) \tag{67}$$

From $\texttt{Send}(i, x) \mathbin{@} u$ in (67) and (64) it follows that the conclusion of (64), i.e, the following, holds.

$$\begin{array}{l}\exists c_p, t_p, u_r, u_n, u_f, a, k_p, n_1. \\ \quad (u_b < u_r < u_n < u_f < u \leq u_e) \wedge \texttt{Owner}(i, k_p) \wedge \\ \quad \texttt{Recv}(i, (c_p, t_p, n_1)) \mathbin{@} u_r \wedge \\ \quad \texttt{New}(i, a) \mathbin{@} u_n \wedge \\ \quad \texttt{Flag}(i, (a, c_p, t_p)) \mathbin{@} u_f \wedge \\ \quad x = (c_p, ENC(k(t_p, k_p), (a, c_p)), \\ \qquad ENC(k(c_p, k_p), (a, n_1, t_p)))\end{array} \tag{68}$$

The last line of (68) and (67) together yield

$$\neg\texttt{SafeMsg}((c_p, ENC(k(t_p, k_p), (a, c_p)), \\ ENC(k(c_p, k_p), (a, n_1, t_p))), AKey_0, \mathcal{K}_0) \quad (69)$$

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

$$\neg\texttt{SafeMsg}(c_p, AKey_0, \mathcal{K}_0) \quad (70)$$

$$\neg\texttt{SafeMsg}(t_p, AKey_0, \mathcal{K}_0) \quad (71)$$

$$\neg\texttt{SafeMsg}(n_1, AKey_0, \mathcal{K}_0) \quad (72)$$

$$\neg\texttt{SafeMsg}(a, AKey_0, \mathcal{K}_0) \quad (73)$$

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:

$$a = AKey_0 \quad (74)$$

From (68) and (74) we conclude:

$$\texttt{New}(i, AKey_0) \ @ \ u_n \quad (75)$$

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

$$u_n = u' \wedge i = I_0 \quad (76)$$

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

$$\texttt{SendsSafeMsg}(i, AKey_0, \mathcal{K}_0) \ @ \ u \quad (77)$$

This contradicts (66).

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

$$\{\texttt{tgs}\}\langle u_b, u_e, i\rangle \\ \forall u \in (u_b, u_e].\ (\forall u' < u.\ \texttt{SafeNet}(AKey_0, \mathcal{K}_0, u')) \\ \supset \texttt{SendsSafeMsg}(i, AKey_0, \mathcal{K}_0) \ @ \ u$$

Using the rules of Figure 6, we show that

$$\{\texttt{tgs}\}\langle u_b, u_e, i\rangle\ \forall u \in (u_b, u_e].\ \forall x.\ \texttt{Send}(i, x) \ @ \ u \supset \\ (\exists tgt, enc_{ct}, s_p, c_p, n_2, skey, a, u_n, u_r, k. \\ (u_b < u_r < u_n < u \leq u_e) \wedge \\ \texttt{Recv}(i, (tgt, enc_{ct}, c_p, s_p, n_2)) \ @ \ u_r \wedge \\ \texttt{New}(i, skey) \ @ \ u_n \wedge \\ x = (c_p, ENC(k, (skey, c_p)), ENC(a, (skey, n_2, s_p)))) \quad (78)$$

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:

$$\forall u' < u.\ \texttt{SafeNet}(AKey_0, \mathcal{K}_0, u') \quad (79)$$

$$\neg\texttt{SendsSafeMsg}(i, AKey_0, \mathcal{K}_0) \ @ \ u \quad (80)$$

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

$$\texttt{Send}(i, x) \ @ \ u \wedge \neg\texttt{SafeMsg}(x, AKey_0, \mathcal{K}_0) \quad (81)$$

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

$$(\exists tgt, enc_{ct}, c_p, s_p, n_2, skey, a, u_n, u_r, k. \\ (u_b < u_r < u_n < u \leq u_e) \wedge \\ \texttt{Recv}(i, (tgt, enc_{ct}, c_p, s_p, n_2)) \ @ \ u_r \wedge \\ \texttt{New}(i, skey) \ @ \ u_n \wedge \\ x = (c_p, ENC(k, (skey, c_p)), ENC(a, (skey, n_2, s_p)))) \quad (82)$$

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

$$\neg\texttt{SafeMsg}((c_p, ENC(k, (skey, c_p)), ENC(a, (skey, n_2, s_p))), \\ AKey_0, \mathcal{K}_0) \quad (83)$$

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

$$\neg\texttt{SafeMsg}(c_p, AKey_0, \mathcal{K}_0) \quad (84)$$

$$\neg\texttt{SafeMsg}(s_p, AKey_0, \mathcal{K}_0) \quad (85)$$

$$\neg\texttt{SafeMsg}(skey, AKey_0, \mathcal{K}_0) \quad (86)$$

$$\neg\texttt{SafeMsg}(n_2, AKey_0, \mathcal{K}_0) \quad (87)$$

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 $(tgt, enc_{ct}, c_p, s_p, n_2)$).

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

$$\{\texttt{server}\}\langle u_b, u_e, i\rangle \\ \forall u \in (u_b, u_e].\ (\forall u' < u.\ \texttt{SafeNet}(AKey_0, \mathcal{K}_0, u')) \\ \supset \texttt{SendsSafeMsg}(i, AKey_0, \mathcal{K}_0) \ @ \ u$$

Using the rules of Figure 6, we show that

$$\{\texttt{server}\}\langle u_b, u_e, i\rangle\ \forall u \in (u_b, u_e].\ \forall x.\ \texttt{Send}(i, x) \ @ \ u \supset \\ (\exists t, k.\ x = ENC(k, t)) \quad (88)$$

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:

$$\forall u' < u.\ \texttt{SafeNet}(AKey_0, \mathcal{K}_0, u') \quad (89)$$

$$\neg\texttt{SendsSafeMsg}(i, AKey_0, \mathcal{K}_0) \ @ \ u \quad (90)$$

Expanding the definition of SendsSafeMsg in (90), we further obtain that there is a message x such that:

$$\texttt{Send}(i, x) \mathbin{@} u \wedge \neg\texttt{SafeMsg}(x, AKey_0, \mathcal{K}_0) \tag{91}$$

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

$$\exists t, k.\ x = ENC(k, t) \tag{92}$$

(92) and (91) together yield

$$\neg\texttt{SafeMsg}(ENC(k, t), AKey_0, \mathcal{K}_0) \tag{93}$$

By definition of SafeMsg, if follows that:

$$\neg\texttt{SafeMsg}(t, AKey_0, \mathcal{K}_0) \tag{94}$$

This yields a contradiction because t is a timestamp and cannot contain $AKey_0$.