# End-to-End Verification of Information-Flow Security for C and Assembly Programs


David Costanzo Zhong Shao Ronghui Gu
Yale University, USA
{david.costanzo,zhong.shao,ronghui.gu}@yale.edu

## Abstract

Protecting the confidentiality of information manipulated by a computing system is one of the most important challenges facing today's cybersecurity community. A promising step toward conquering this challenge is to formally verify that the end-to-end behavior of the computing system really satisfies various information-flow policies. Unfortunately, because today's system software still consists of both C and assembly programs, the end-to-end verification necessarily requires that we not only prove the security properties of individual components, but also carefully preserve these properties through compilation and cross-language linking.

In this paper, we present a novel methodology for formally verifying end-to-end security of a software system that consists of both C and assembly programs. We introduce a general definition of observation function that unifies the concepts of policy specification, state indistinguishability, and whole-execution behaviors. We show how to use different observation functions for different levels of abstraction, and how to link different security proofs across abstraction levels using a special kind of simulation that is guaranteed to preserve state indistinguishability. To demonstrate the effectiveness of our new methodology, we have successfully constructed an end-to-end security proof, fully formalized in the Coq proof assistant, of a nontrivial operating system kernel (running on an extended CompCert x86 assembly machine model). Some parts of the kernel are written in C and some are written in assembly; we verify all of the code, regardless of language.

***Categories and Subject Descriptors*** D.2.4 [*Software Engineering*]: Software/Program Verification—Correctness proofs, formal methods; D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics; D.4.5 [*Operating Systems*]: Reliability—Verification; D.4.6 [*Operating Systems*]: Security and Protection—Information flow controls; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

***General Terms*** Verification, Security, Reliability, Languages, Design

***Keywords*** Information Flow Control; Security Policy Specification; Security-Preserving Simulation; Program Verification; Certified OS Kernels.



*PLDI'16*, June 13–17, 2016, Santa Barbara, CA, USA
ACM. 978-1-4503-4261-2/16/06...$15.00
http://dx.doi.org/10.1145/2908080.2908100

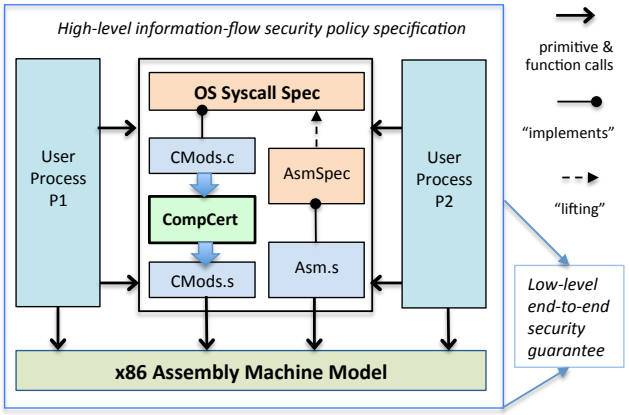


**Figure 1.** An end-to-end software system that consists of both OS modules (in C and assembly) and user processes.

## 1. Introduction

Information flow control (IFC) [21, 24] is a form of analysis that tracks how information propagates through a system. It can be used to state and verify important security-related properties about the system. In this work, we will focus on the read-protection property known as *confidentiality* or *privacy*, using these terms interchangeably with security.

Security is desirable in today's real-world software. Hackers often exploit software bugs to obtain information about protected secrets, such as user passwords or private keys. A formally-verified end-to-end security proof can guarantee such exploits will never be successful. There are many significant roadblocks involved in such a verification, however, and the state-of-the-art is not entirely satisfactory.

Consider the setup of Figure 1, where a large system (e.g., an OS) consists of many separate functions (e.g., system call primitives) written in either C or assembly. Each primitive has a verified atomic specification, and there is a verified compiler, such as CompCert [14], that can correctly compile C programs into assembly. We wish to prove an *end-to-end* security statement about some context program (e.g., P1 or P2) that can call the primitives of the system, which ultimately guarantees that our model of the concrete execution (i.e., the whole-program assembly execution) behaves securely. This goal raises a number of challenges:

- *Policy Specification* — How do we specify a clear and precise security policy, describing how information is allowed to flow between various domains? If we express the policy in terms of the high-level syscall specifications, then what will this imply for the whole-program assembly execution? We need some way of specifying policies at different levels of abstraction, as well as translating between or linking separate policies.
- *Propagating Security* — It is well known [11, 17] that simulations and refinements may not propagate security guarantees. How, then, can we soundly obtain a low-level guarantee from a high-level security verification?
- *Proving Security* — A standard way to prove confidentiality is to formulate the property as noninterference, and prove that some *state indistinguishability* relation is preserved by each step of execution (this is known as an *unwinding condition* [7, 8], and will be discussed in Section 2). However, this proof does not propagate down to the implementation: a syscall specification may atomically preserve indistinguishability, but its non-atomic implementation may temporarily break the relation during intermediate states. Thus we must be careful to formulate the security of a primitive's implementation as a global behavior-based property over the whole execution of the implementation, rather than a local state-based property over individual steps.
- *Cross-Language Linking* — Even if we verify security for all atomic primitive specifications and propagate the proofs to implementations, there still may be incompatibilities between the proofs for C primitives and those for assembly primitives. For example, a security proof for an assembly primitive might express that some data stored in a particular machine register is not leaked; this property cannot be directly chained with one for a C primitive since the C memory model does not contain machine registers. We therefore must support linking the specifications of primitives implemented in different languages.

In this paper, we present a novel methodology for formally verifying end-to-end security of a system like the one shown in Figure 1. First, security is proved for the high-level specification of each syscall in a standard way, establishing noninterference by showing that a state indistinguishability relation is preserved across the specification. Then we apply simulation techniques to automatically obtain a sound security guarantee for the low-level assembly machine execution, which is expressed in terms of whole-execution observations. Simulations are used both for relating specifications with their C or assembly implementations, as well as for relating C implementations with their compiled assembly implementations.

The central idea of our methodology is to introduce a flexible definition of observation that unifies the concepts of policy specification, state indistinguishability, and whole-execution behaviors. For every level of abstraction, we define an *observation function* that describes which portions of a program state are observable to which principals. For example, an observation function might say that "$x$ is observable to Alice" and "$y$ is observable to Bob".

Different abstraction levels can use different observation functions. We might use one observation function mentioning machine registers to verify an assembly primitive, and a second observation function mentioning program variables to verify a C primitive. These observation functions are then linked across abstraction levels via a special kind of simulation that preserves state indistinguishability.

We demonstrate the efficacy of our approach by applying it to the mCertiKOS operating system [9]. We modify mCertiKOS to disable all explicit inter-process communication, and then we prove noninterference between user processes with distinct IDs. mCertiKOS guarantees full functional correctness of system calls (with respect to an x86 machine model derived from CompCert's model) by chaining simulations across many abstraction layers. We implement our general notion of observation function over the existing simulation framework, and then verify security of the high-level system call specifications. The result of this effort is a formally-verified security guarantee for the operating system — we specify exactly which portions of high-level state are observable to which processes, and we prove that the low-level model of assembly execution of the whole system is secure with respect to this policy. The security guarantee can be seen as *end-to-end* in the following two ways: (1) it applies across simulations, propagating from a top-level specification to a concrete implementation; and (2) it applies to whole-execution behaviors, guaranteeing that an entire execution is secure from start to finish.

To summarize, the primary contributions of this work are:

- A novel methodology for end-to-end security verification of software systems written in both C and assembly, and extending across various levels of abstraction.
- An end-to-end security proof, completely formalized in the Coq proof assistant [27], of a simple but nontrivial operating system kernel that executes over an extension of the CompCert x86 machine model. The kernel is non-preemptive, and explicit communication between processes is disabled.

The rest of this paper is organized as follows. Sec. 2 introduces the observation function and shows how to use it for policy specification, security proof, and linking. Sec. 3 formalizes our simulation framework and shows how we prove the end-to-end security theorem. Sec. 4 and 5 describe the security property that we prove over mCertiKOS, highlighting the most interesting aspects of our proofs. Sec. 6 discusses limitations, assumptions, and applicability of our methodology. Finally, Sec. 7 discusses related work and concludes.

## 2. The Observation Function

This section will explore our notion of observation, describing how it cleanly unifies various aspects of security verification. Assume we have some set $\mathcal{L}$ of principals or security domains that we wish to fully isolate from one another, and a state transition machine $M$ describing the single-step operational semantics of execution at a particular level of abstraction. For *any* type of observations, we define the *observation function* of $M$ to be a function mapping a principal and program state to an observation. For a principal $l$ and state $\sigma$, we express the state observation notationally as $\mathcal{O}_{M;l}(\sigma)$, or just $\mathcal{O}_l(\sigma)$ when the machine is obvious from context.

### 2.1 High-Level Security Policies

We use observation functions to express high-level policies. Consider the following C function (assume variables are global for the purpose of presentation):

```
void add() {
    a = x + y;
    b = b + 2; }
```

Clearly, there are flows of information from $x$ and $y$ to $a$, but no such flows to $b$. We express these flows in a policy induced by the observation function. Assume that program state is represented as a partial variable store, mapping variable names to either `None` if the variable is undefined, or `Some` $v$ if the variable is defined and contains integer value $v$. We will use the notation $[x \hookrightarrow 7; y \hookrightarrow 5]$ to indicate the variable store where $x$ maps to `Some` 7, $y$ maps to `Some` 5, and all other variables map to `None`.

We consider the value of $a$ to be observable to Alice (principal $A$), and the value of $b$ to be observable to Bob (principal $B$). Since there is information flow from $x$ and $y$ to $a$ in this example, we will also consider the values of $x$ and $y$ to be observable to Alice. Hence we define the observation type to be partial variable stores (same as program state), and the observation function is:

$$\mathcal{O}_A(\sigma) \triangleq [a \hookrightarrow \sigma(a);\ x \hookrightarrow \sigma(x);\ y \hookrightarrow \sigma(y)]$$
$$\mathcal{O}_B(\sigma) \triangleq [b \hookrightarrow \sigma(b)]$$

This observation function induces a policy over an execution, stating that for each principal, the final observation is dependent only upon the contents of the initial observation. This means that Alice can potentially learn anything about the initial values of $a$, $x$, and $y$, but she can learn nothing about the initial value of $b$. Similarly, Bob cannot learn anything about the initial values of $a$, $x$, or $y$. It should be fairly obvious that the `add` function is secure with respect to this policy; we will discuss how to prove this fact shortly.

***Alternative Policies*** Since the observation function can be anything, we can express various intricate policies. For example, we might say that Alice can only observe parities:

$$\mathcal{O}_A(\sigma) \triangleq [a \hookrightarrow \sigma(a)\%2;\ x \hookrightarrow \sigma(x)\%2;\ y \hookrightarrow \sigma(y)\%2]$$

We also do not require observations to be a portion of program state, so we might express that the average of $x$ and $y$ is observable to Alice:

$$\mathcal{O}_A(\sigma) \triangleq (\sigma(x) + \sigma(y))/2$$

Notice how this kind of observation expresses a form of declassification, saying that the average of the secret values in $x$ and $y$ can be declassified to Alice. This security policy has some real-world applications: for example, a company may wish to make the average of its employees' salaries public, without directly releasing any individual's salary.

One important example of observation is a representation of the standard label lattices and tainting used in many security frameworks. Security domains are arranged in a lattice structure, and information is only allowed to flow up the lattice. Suppose we attach a security label to each piece of data in a program state. We can then define the observation function for a label $l$ to be the portion of state that has a label at or below $l$ in the lattice. As is standard, we define the semantics of a program such as $a = x + y$ to set the resulting label of $a$ to be the least upper bound of the labels of $x$ and $y$. Hence any label that is privileged enough to observe $a$ will also be able to observe both $x$ and $y$. We can then prove that this semantics is secure with respect to our lattice-aware observation function. In this way, our observation function can directly model label tainting.

The generality of our observation function allows for the expression of many different kinds of security policies. While we have not exhaustively studied the extent of policy expressibility, we have anecdotally found it to be similar to other frameworks that express observational equivalence in a purely semantic fashion, e.g., Sabelfeld et al.'s PER model [25] and Nanevski et al.'s Relational Hoare Type Theory [22]. To provide better intuition for how observation functions are used to express security policies, we discuss some more examples in semi-formal detail in Appendix A. The observation function used for the mCertiKOS security proof, to be presented in Section 4, also helps in this regard.

### 2.2 Security Formulation

***High-Level Security*** As mentioned in Section 1, we prove security at a high abstraction level by using an unwinding

condition. Specifically, for a given principal $l$, this unwinding condition says that state indistinguishability is preserved by each step of a transition semantics, where two states are said to be indistinguishable just when their observations are equal. Intuitively, if a semantics always preserves indistinguishability, then the final observation can never be influenced by changing unobservable data in the initial state (i.e., high-security inputs cannot influence low-security outputs).

More formally, for any principal $l$ and state transition machine $M$ with single-step transition semantics $T_M$, we say that $M$ is secure for $l$ if the following property holds for all states $\sigma_1$, $\sigma_2$, $\sigma_1'$, and $\sigma_2'$:

$$\mathcal{O}_l(\sigma_1) = \mathcal{O}_l(\sigma_2) \wedge (\sigma_1, \sigma_1') \in T_M \wedge (\sigma_2, \sigma_2') \in T_M \\ \implies \mathcal{O}_l(\sigma_1') = \mathcal{O}_l(\sigma_2')$$

Consider how this property applies to an atomic specification of the `add` function above, using the observation function where only the parities of $a$, $x$, and $y$ are observable to Alice. Two states are indistinguishable to Alice just when the parities of these three variables are the same in the states. Taking the entire function as an atomic step, we see that indistinguishability is indeed preserved since $a$ gets updated to be the sum of $x$ and $y$, and addition is homomorphic with respect to parity. Hence the policy induced by this observation function is provably secure.

***Low-Level Security*** While the above unwinding condition is used to prove security across atomic specifications of functions like `add`, we ultimately require a security guarantee that applies to the non-atomic implementations of these functions. Notice that the implementation of `add` satisfies the unwinding condition: if we consider a machine where a single step corresponds to a single line of C code, then both of the two steps involved in executing `add` preserve indistinguishability. However, this is not true in general. Consider an alternative implementation of `add` with the same atomic specification:

```
void add() {
    a = b;
    a = x + y;
    b = b + 2;  }
```

The first line of this implementation may not preserve indistinguishability since the unobservable value of $b$ is directly written into $a$. Nevertheless, the second line immediately overwrites $a$, reestablishing indistinguishability. This illustrates that we cannot simply prove the unwinding condition for high-level atomic specifications, and expect it to automatically propagate to a non-atomic implementation. We therefore must use a different security definition for low-level implementations, one which considers observations of entire executions rather than just single steps.

Intuitively, we will express low-level security as equality between the "whole-execution observations" produced by two executions starting from indistinguishable states. To formalize this intuition, we must address: (a) the meaning of state indistinguishability at the implementation level; and (b) the meaning of whole-execution observations.

***Low-Level Indistinguishability*** For high-level security, we defined state indistinguishability to be equality of the state-parameterized observation functions. This definition may not work well at a lower level of abstraction, however, since security-relevant logical state may be hidden by simulation. For example, suppose we attach security labels to data in a high-level state, for the purpose of specifying the policy based on label tainting described above. Further suppose that we treat the labels as logical state, erasing them when simulating the high-level specification with its implementation (i.e., the low-level machine model does not contain any physical representation of the security labels). This means that, at the implementation level, we can no longer define the portion of program state belonging to a particular principal. Hence it becomes unclear what state indistinguishability should mean at this level.

We resolve this difficulty by defining low-level state indistinguishability in terms of high-level indistinguishability and simulation. We say that, given a simulation relation $R$ connecting specification to implementation, two low-level states are indistinguishable if there exist two indistinguishable high-level states that are related to the low-level states by $R$. This definition will be fully formalized in Section 3.

***Whole-Execution Observations*** We define the observations made by an entire execution in terms of external events, which are in turn defined by a machine's observation function. Many traditional automaton formulations define an external event as a label on the step relation. Each individual step of an execution may or may not produce an event, and the whole-execution observation, or *behavior*, is the concatenation of all events produced across the execution.

We use the observation function to model external events. The basic idea is to equate an event being produced by a transition with the state observation changing across the transition. This idea by itself does not work, however. When events are expressed externally on transitions, they definitionally enjoy an important monotonicity property: whenever an event is produced, that event cannot be "undone" or "forgotten" at any future point in the execution. When events are expressed as changes in state observation, this property is no longer guaranteed.

We therefore explicitly enforce a monotonicity condition on the observation function of an implementation. We require a partial order to be defined over the observation type of the low-level semantics, as well as a proof that every step of the semantics respects this order. For example, our mCertiKOS proof represents the low-level observation as an output buffer (a Coq list). The partial order is defined based on list prefix, and we prove that execution steps will always respect the order by either leaving the output buffer unchanged or appending to the end of the buffer.

Note that we *only* enforce observation monotonicity on the implementation. It is crucial that we do not enforce it on the high-level specification; doing so would greatly restrict the high-level policies we could specify, and would potentially make the unwinding condition of the high-level security proof unprovable. Intuitively, a non-monotonic observation function expresses which portions of state could potentially influence the observations produced by an execution, while a monotonic observation function expresses which observations the execution has actually produced. We are interested in the former at the specification level, and the latter at the implementation level.

### 2.3 Security-Preserving Simulation

The previous discussion described how to use the observation function to express both high-level and low-level security properties. With some care, we can automatically derive the low-level security property from a simulation and a proof of the high-level security property.

It is known that, in general, security is not automatically preserved across simulation. One potential issue, known as the refinement paradox [11, 17, 18], is that a nondeterministic secure program can be refined into a more deterministic but insecure program. For example, suppose we have a secret boolean value stored in $x$, and a program $P$ that randomly prints either `true` or `false`. $P$ is obviously secure since its output has no dependency on the secret value, but $P$ can be refined by an insecure program $Q$ that directly prints the value of $x$. We avoid this issue by ruling out $P$ as a valid secure program: despite being obviously secure, it does not actually satisfy the unwinding condition defined above and hence is not provably secure in our framework. Note that the seL4 security verification [19] avoids this issue in the same way. In that work, the authors frame their solution as a restriction that disallows specifications from exhibiting any *domain-visible* nondeterminism. Indeed, this can be seen clearly by specializing the unwinding condition above such that states $\sigma_1$ and $\sigma_2$ are identical:

$$(\sigma, \sigma_1') \in T_M \wedge (\sigma, \sigma_2') \in T_M \implies \mathcal{O}_l(\sigma_1') = \mathcal{O}_l(\sigma_2')$$

The successful security verifications of both seL4 and mCertiKOS provide reasonable evidence that this restriction on specifications is not a major hindrance for usability.

Unlike the seL4 verification, however, our framework runs into a second issue with regard to preserving security across simulation. The issue arises from the fact that both simulation relations and observation functions are defined in terms of program state, and they are both arbitrarily general. This means that certain simulation relations may, in some sense, behave poorly with respect to the observation function. Figure 2 illustrates an example. Assume program state at both levels consists of three variables $x$, $y$, and $z$. The observation function is the same at both levels: $x$ and $y$ are unobservable while $z$ is observable. Suppose we have a deterministic

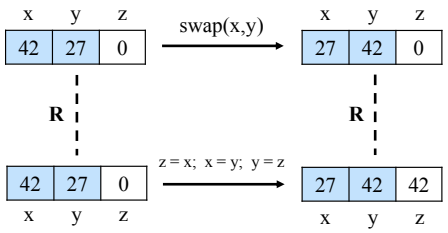


**Figure 2.** Security-Violating Simulation. The shaded part of state is unobservable, while the unshaded part is observable.

specification of the `swap` primitive saying that the values of $x$ and $y$ are swapped, and the value of $z$ is unchanged. Also suppose we have a simulation relation $R$ that relates any two states where $x$ and $y$ have the same values, but $z$ may have different values. Using this simulation relation, it is easy to show that the low-level swap implementation simulates the high-level swap specification.

Since the swap specification is deterministic, this example is unrelated to the issue described above, where domain-visible nondeterminism in the high-level program causes trouble. Nevertheless, this example fails to preserve security across simulation: the high-level program clearly preserves indistinguishability, while the low-level one leaks the secret value of $x$ into the observable variable $z$.

As mentioned above, the root cause of this issue is that there is some sort of incompatibility between the simulation relation and the observation function. In particular, security is formulated in terms of a state indistinguishability relation, but the simulation relation may fail to preserve indistinguishability. Indeed, for the example of Figure 2, it is easy to demonstrate two indistinguishable program states that are related by $R$ to two distinguishable ones. Thus our solution to this issue is to restrict simulations to require that state indistinguishability is preserved. More formally, given a principal $l$, in order to show that machine $m$ simulates $M$ under simulation relation $R$, the following property must be proved for all states $\sigma_1$, $\sigma_2$ of $M$, and states $s_1$, $s_2$ of $m$:

$$\mathcal{O}_{M;l}(\sigma_1) = \mathcal{O}_{M;l}(\sigma_2) \wedge (\sigma_1, s_1) \in R \wedge (\sigma_2, s_2) \in R \implies \mathcal{O}_{m;l}(s_1) = \mathcal{O}_{m;l}(s_2)$$

## 3. End-to-End Security Formalization

In this section, we describe the formal proof that security is preserved across simulation. Most of the technical details are omitted here for clarity of presentation, but can be found in the companion technical report [4].

***Machines with Observations*** In the following, assume we have a set $\mathcal{L}$ of distinct principals or security domains.

**Definition 1** (Machine)**.** A *state transition machine* $M$ consists of the following components (assume all sets may be finite or infinite):

- a type $\Sigma_M$ of program state

- a set of initial states $I_M$ and final states $F_M$
- a transition (step) relation $T_M$ of type $\mathcal{P}(\Sigma_M \times \Sigma_M)$
- a type $\Omega_M$ of observations
- an observation function $\mathcal{O}_{M;l}(\sigma)$ of type $\mathcal{L} \times \Sigma_M \to \Omega_M$

When the machine $M$ is clear from context, we use the notation $\sigma \mapsto \sigma'$ to mean $(\sigma, \sigma') \in T_M$. For multiple steps, we define $\sigma \mapsto^n \sigma'$ in the obvious way, meaning that there exists a chain of states $\sigma_0, ..., \sigma_n$ with $\sigma = \sigma_0$, $\sigma' = \sigma_n$, and $\sigma_i \mapsto \sigma_{i+1}$ for all $i \in [0, n)$. We then define $\sigma \mapsto^* \sigma'$ to mean that there exists some $n$ such that $\sigma \mapsto^n \sigma'$, and $\sigma \mapsto^+ \sigma'$ to mean the same but with a nonzero $n$.

Notice that our definition is a bit different from most traditional definitions of automata, in that we do not define any explicit notion of actions on transitions. In traditional definitions, actions are used to represent some combination of input events, output events, and instructions/commands to be executed. In our approach, we advocate moving all of these concepts into the program state — this simplifies the theory, proofs, and policy specifications.

***Initial States vs Initialized States*** Throughout our formalization, we do not require anything regarding initial states of a machine. The reason is related to how we will actually carry out security and simulation proofs in practice (described with respect to the mCertiKOS security proof in Sections 4 and 5). We never attempt to reason about the true initial state of a machine; instead, we assume that some appropriate setup/configuration process brings us from the true initial state to some properly *initialized* state, and then we perform all reasoning under the assumption of proper initialization.

***High-Level Security*** As described in Section 2, we use different notions of security for the high level and the low level. High-level security says that each individual step preserves indistinguishability. It also requires a safety proof as a precondition, guaranteeing that the machine preserves some initialization invariant $I$.

**Definition 2** (Safety)**.** We say that a machine $M$ is safe under state predicate $I$, written $\texttt{safe}(M, I)$, when the following progress and preservation properties hold:

1.) $\forall \sigma \in I - F_M \;.\; \exists \sigma' \;.\; \sigma \mapsto \sigma'$

2.) $\forall \sigma, \sigma' \;.\; \sigma \in I \wedge \sigma \mapsto \sigma' \implies \sigma' \in I$

**Definition 3** (High-Level Security)**.** Machine $M$ is secure for principal $l$ under invariant $I$, written $\Delta M_l^I$, just when:

1.) $\texttt{safe}(M, I)$

2.) $\forall \sigma_1, \sigma_2 \in I,\ \sigma_1', \sigma_2' \;.$
$\mathcal{O}_l(\sigma_1) = \mathcal{O}_l(\sigma_2) \wedge \sigma_1 \mapsto \sigma_1' \wedge \sigma_2 \mapsto \sigma_2'$
$\implies \mathcal{O}_l(\sigma_1') = \mathcal{O}_l(\sigma_2')$

3.) $\forall \sigma_1, \sigma_2 \in I \;.$
$\mathcal{O}_l(\sigma_1) = \mathcal{O}_l(\sigma_2) \implies (\sigma_1 \in F_M \iff \sigma_2 \in F_M)$

***Low-Level Security*** For low-level security, as discussed in Section 2, we first must define whole-execution behaviors with respect to a monotonic observation function.

**Definition 4** (Behavioral Machine)**.** We say that a machine $M$ is behavioral for principal $l$ when we have a partial order defined over $\Omega_M$, and a proof that every step of $M$ is monotonic with respect to this order.

For any machine $M$ that is behavioral for principal $l$, we can define the set of whole-execution behaviors possibly starting from a given state $\sigma$; we denote this set as $\mathcal{B}_{M;l}(\sigma)$. The three kinds of behaviors are faulting (getting stuck), safe termination, and safe divergence. The definitions can be found in the TR [4]; the main point here is that behaviors use the machine's observation function as a building block. For example, a behavior might say "an execution from $\sigma$ terminates with final observation $o$", or "an execution from $\sigma$ diverges, producing an infinite stream of observations $os$".

We can now define whole-execution security of a behavioral machine as behavioral equality. Note that, in our final end-to-end security theorem, the low-level executions in question will be obtained from relating indistinguishable high-level states across simulation. We hide this detail for now inside of an abstract indistinguishability relation $\rho$, and will revisit the relation later in this section.

**Definition 5** (Low-Level Security)**.** Given a machine $m$ that is behavioral for principal $l$, we say that $m$ is behaviorally secure for $l$ under some indistinguishability relation $\rho$, written $\nabla m_l^\rho$, just when:

$$\forall \sigma_1, \sigma_2 \;.\; \rho(\sigma_1, \sigma_2) \implies \mathcal{B}_{m;l}(\sigma_1) = \mathcal{B}_{m;l}(\sigma_2)$$

***Simulation*** We next formalize our definition of simulation. It is a standard definition, except for the following aspects:

1. As explained above, we do not require any relationships to hold between initial states.
2. As described informally in Section 2, we require simulation relations to preserve state indistinguishability.

**Definition 6** (Simulation)**.** Given two machines $M$ and $m$, a principal $l$, and a relation $R$ of type $\mathcal{P}(\Sigma_M \times \Sigma_m)$, we say that $M$ simulates $m$ using $R$, written $M \sqsubseteq_{R;l} m$, when:

1.) $\forall \sigma, \sigma' \in \Sigma_M,\ s \in \Sigma_m \;.$
$\sigma \mapsto \sigma' \wedge R(\sigma, s)$
$\implies \exists s' \in \Sigma_m \;.\; s \mapsto^* s' \wedge R(\sigma', s')$

2.) $\forall \sigma \in \Sigma_M,\ s \in \Sigma_m \;.$
$\sigma \in F_M \wedge R(\sigma, s) \implies s \in F_m$

3.) $\forall \sigma_1, \sigma_2 \in \Sigma_M,\ s_1, s_2 \in \Sigma_m \;.$
$\mathcal{O}_{M;l}(\sigma_1) = \mathcal{O}_{M;l}(\sigma_2) \wedge R(\sigma_1, s_1) \wedge R(\sigma_2, s_2)$
$\implies \mathcal{O}_{m;l}(s_1) = \mathcal{O}_{m;l}(s_2)$

The first property is the main simulation, the second relates final states, and the third preserves indistinguishability. For presentation purposes, we omit details regarding the well-known "infinite stuttering" problem for simulations (described, for example, in [15]). Our Coq definition of simulation includes a well-founded order that prevents infinite stuttering.

Also notice that, contrary to our discussion earlier, we do not define simulations to be relative to an invariant. It would be completely reasonable to require safety of the higher-level machine under some invariant, but this actually ends up being redundant. Since $R$ is an arbitrary relation, we can simply embed an invariant requirement within $R$. In other words, one should think of $R(\sigma, s)$ as saying not only that $\sigma$ and $s$ are related, but also that $\sigma$ satisfies an appropriate invariant.

***End-to-End Security*** Our end-to-end security theorem is proved with the help of a few lemmas about behaviors and simulations, described in the companion TR [4]. As an example, one lemma says that if we have a simulation between behavioral machines $M$ and $m$, then the possible behaviors of $M$ from some state $\sigma$ are a subset of the behaviors of $m$ from a related state $s$. There is one significant technical detail we should mention here regarding these lemmas: behaviors are defined in terms of observations, and the types of observations of two different machines may be different. Hence we technically cannot compare behavior sets directly using standard subset or set equality.

For the purpose of presentation, we will actually assume here that all *behavioral* machines under consideration use the same type for observations. In fact, the mCertiKOS proof is a special case of our framework that obeys this assumption (all machines use the output buffer observation). Our framework is nevertheless capable of handling changes in observation type by adding a new relation to simulations that relates observations; the companion TR [4] contains the details.

We are now ready to describe how simulations preserve security. As mentioned previously, low-level security uses an indistinguishability relation derived from high-level indistinguishability and a simulation relation:

**Definition 7** (Low-Level Indistinguishability)**.**

$$\begin{aligned}&\phi(M, l, I, R) \triangleq \\ &\quad \lambda s_1, s_2 \,.\, \exists \sigma_1, \sigma_2 \in I \,. \\ &\qquad \mathcal{O}_{M;l}(\sigma_1) = \mathcal{O}_{M;l}(\sigma_2) \wedge R(\sigma_1, s_1) \wedge R(\sigma_2, s_2)\end{aligned}$$

**Theorem 1** (End-to-End Security)**.** *Suppose we have two machines $M$ and $m$, a principal $l$, a high-level initialization invariant $I$, and a simulation $M \sqsubseteq_{R;l} m$. Further suppose that $m$ is deterministic and behavioral for $l$. Let low-level indistinguishability relation $\rho$ be $\phi(M, l, I, R)$ from Definition 7. Then high-level security implies low-level security:*

$$\Delta M_l^I \Longrightarrow \nabla m_l^\rho$$

*Proof Sketch.* We prove this theorem by defining a new machine $N$ between $M$ and $m$, and proving simulations from $M$ to $N$ and from $N$ to $m$. $N$ will mimic $M$ in terms of program states and transitions, while it will mimic $m$ in terms of observations. We prove that $N$ is behavioral, and apply some lemmas to equate the whole-execution behaviors of $m$ with those of $N$. We then formulate the high-level security proof as a bisimulation over $M$, and use this to derive a bisimulation over $N$. Finally, we apply a lemma to connect the bisimulation over $N$ with the whole-execution behaviors of $N$, completing the proof. The details of this proof and the relevant lemmas can be found in the companion TR [4]. □

## 4. Security Definition of mCertiKOS

We will now discuss how we applied our methodology to prove an end-to-end security guarantee between separate processes running on top of the mCertiKOS kernel [9]. During the proof effort, we had to make some changes to the operating system to close potential security holes. We refer to our secure variant of the kernel as mCertiKOS-secure.

### 4.1 mCertiKOS Overview

The starting point for our proof effort was the basic version of the mCertiKOS kernel, described in detail in Section 7 of [9]. We will give an overview of the kernel here. It is composed of 32 abstraction *layers*, which incrementally build up the concepts of physical memory management, virtual memory management, kernel-level processes, and user-level processes. Each layer $L$ consists of the following components:

- a type $\Sigma_L$ of program state, separated into machine registers, concrete memory, and abstract data of type $D_L$
- a set of initial states $I_L$ and final states $F_L$
- a set of primitives $P_L$ implemented by the layer
- for each $p \in P_L$, a specification of type $\mathcal{P}(\Sigma_L \times \Sigma_L)$
- *(if $L$ is not the bottom layer)* for each $p \in P_L$, an implementation written in either LAsm($L'$) or ClightX($L'$) (defined below), where $L'$ is the layer below $L$
- two special primitives called `load` and `store` that model access to global memory; these primitives have no implementation as they are a direct model of how the x86 machine translates virtual addresses using page tables

The top layer is called TSysCall, and the bottom is called MBoot. MBoot describes execution over the model of the actual hardware; the specifications of its primitives are taken as axioms. Implementations of primitives in all layers are written in either a layer-parameterized variant of x86 assembly or a layer-parameterized variant of C.

The assembly language, called LAsm($L$), is a direct extension of CompCert's [14] model of x86 assembly that allows primitives of layer $L$ to be called atomically. When an atomic primitive call occurs, the semantics consults that primitive's specification to take a step. Note that the `load` and `store` primitives are never called explicitly (as they have no implementation), but instead are used to specify the semantics of x86 instructions that read or write memory (e.g., `movl %eax, 0(%ecx)`).

The C variant, called ClightX($L$), is a direct extension of CompCert's Clight language [1] (which is a slightly-simplified version of C). Like LAsm($L$), the semantics is

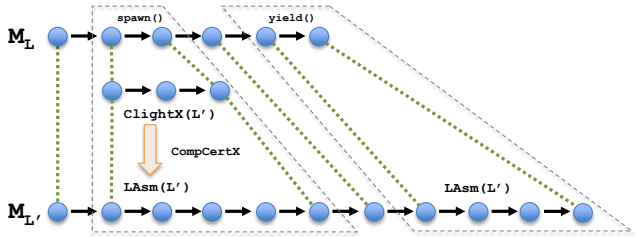


**Figure 3.** Simulation between adjacent layers. Layer $L$ contains primitives `spawn` and `yield`, with the former implemented in ClightX($L'$) and the latter implemented in LAsm($L'$).

extended with the ability to call the primitives of $L$ atomically. ClightX($L$) programs can be compiled to LAsm($L$) in a verified-correct fashion using the CompCertX compiler [9], which is an extension of CompCert.

Each layer $L$ induces a machine $M_L$ of the kind described in Section 3, with transition relation defined by the operational semantics of LAsm($L$).

***Layer Simulation*** Figure 3 illustrates how machines induced by two consecutive layers are connected via simulation. Each step of machine $M_L$ is either a standard assembly command or an atomic primitive call. Steps of the former category are simulated in $M_{L'}$ by exactly the same assembly command. Steps of the latter are simulated using the primitive's implementation, supplied by layer $L$. If the primitive is implemented in LAsm($L'$), then the simulation directly uses the semantics of this implementation. If the primitive is implemented in ClightX($L'$), then CompCertX is used first to compile the implementation into LAsm($L'$). CompCertX is verified to provide a simulation from the ClightX($L'$) execution to the corresponding LAsm($L'$) execution, so this simulation is chained appropriately.

Once every pair of consecutive machines is connected with a simulation, they are combined to obtain a simulation from TSysCall to MBoot. Since the TSysCall layer provides mCertiKOS's system calls as primitives, user process execution is specified at the TSysCall level. To get a better sense of user process execution, we will now give an overview of the TSysCall program state and primitives.

***TSysCall State*** The TSysCall abstract data is a Coq record consisting of 32 separate fields. We will list here those fields that will be relevant to our discussion. In the following, whenever a field name has a subscript of $i$, the field is a finite map from process ID to some data type.

- `out`$_i$ — The output buffer for process $i$. Note that output buffers exist in all layers' abstract data, including MBoot. They are never actually implemented in memory; instead, they are assumed to be a representation of some external method of output (e.g., a monitor or a network channel).
- `HP` — A global, flat view of the user-space memory heap. A *page* is defined as the 4096-byte sequence starting from a physical address that is divisible by 4096.
- `AT` — A global allocation table, represented as a bitmap indicating which pages in the global heap have been allocated. Element $n$ corresponds to page $n$.
- `pgmap`$_i$ — A representation of the two-level page map for process $i$. The page map translates a virtual address between 0 and $2^{32} - 1$ into a physical address.
- `container`$_i$ — A representation of process $i$ that maintains information regarding spawned status, children, parents, and resource quota. A container is itself a Coq record containing the following fields:
  - `used` — A boolean indicating whether process $i$ has been spawned.
  - `parent` — The ID of the parent of process $i$.
  - `nchildren` — The number of children of process $i$.
  - `quota` — The maximum number of pages that process $i$ is allowed to dynamically allocate.
  - `usage` — The current number of pages that process $i$ has dynamically allocated.
- `ctxt`$_i$ — The saved register context of process $i$, containing the register values that will need to be restored the next time process $i$ is scheduled.
- `cid` — The currently-running process ID.

***TSysCall Primitives*** There are 9 primitives in the TSysCall layer, including the load/store primitives. The primitive specifications operate over both the TSysCall abstract data and the machine registers. Note that they do not interact with concrete memory since all relevant portions of memory have already been abstracted into the TSysCall abstract data.

- *Initialization* — `proc_init` sets up the various kernel objects to get everything into a working state. We never attempt to reason about anything that happens prior to initialization; it is assumed that the bootloader will always call `proc_init` appropriately.
- *Load/Store* — Since paging is enabled at the TSysCall level, the `load` and `store` primitives walk the page tables of the currently-running process to translate virtual addresses into physical. If no physical address is found due to no page being mapped, then the faulting virtual address is written into the CR2 control register, the current register context is saved, and the instruction pointer register is updated to point to the entry of the page fault handler primitive.
- *Page Fault* — `pgf_handler` is called immediately after one of the load/store primitives fails to resolve a virtual address. It reads the faulting virtual address from the CR2 register, allocates one or two new pages as appropriate, increases the current process's page usage, and plugs the page(s) into the two-level page table. It then restores the register context that was saved when the load/store primitive faulted. If the current process does not have

enough available quota to allocate the required pages, then the instruction pointer register is updated to point to the entry of the yield primitive (see below).

- *Get Quota* — `get_quota` returns the amount of remaining quota for the currently-executing process. This is useful to provide as a system call since it allows processes to divide their quota among children in any way they wish.
- *Spawn Process* — `proc_create` attempts to spawn a new child process. It takes a quota as a parameter, specifying the maximum number of pages the child process will be allowed to allocate. This quota allowance is taken from the current process's available quota.
- *Yield* — `sys_yield` performs the first step for yielding. It enters kernel mode, disables paging, saves the current registers, and changes the currently-running process ID to the head of the ready queue. It then context switches by restoring the newly-running process's registers. The newly-restored instruction pointer register is guaranteed (proved as an invariant) to point to the function entry of the `start_user` primitive.
- *Start User* — `start_user` performs the simple second step of yielding. It enables paging for the currently-running process and exits kernel mode. The entire functionality of yielding must be split into two primitives (`sys_yield` and `start_user`) because context switching requires writing to the instruction pointer register, and therefore only makes sense when it is the final operation performed by a primitive. Hence yielding is split into one primitive that ends with a context switch, and a second primitive that returns to user mode.
- *Output* — `print` appends its parameter to the currently-running process's output buffer.

### 4.2 Security Overview

We consider each process ID to be a distinct principal. The security property that we aim to prove is exactly the high-level security defined in Section 3 (Definition 3), applied over the TSysCall machine using a carefully-constructed observation function that we define below. Theorem 1 then guarantees security of the corresponding whole-execution behaviors over the MBoot machine (which represents our lowest-level model of the assembly machine).

***High-Level Semantics*** High-level security is proved by showing that every step of execution preserves an indistinguishability relation saying that the observable portions of two states are equal. In the mCertiKOS context, however, this property will not quite hold over the TSysCall machine.

To see this, consider any process ID (i.e., principal) $l$, which we call the "observer process". For any TSysCall state $\sigma$, we say that $\sigma$ is "active" if $\texttt{cid}(\sigma) = l$, and "inactive" otherwise. Now consider whether the values in machine registers should be observable to $l$. Clearly, if $l$ is executing, then it can read and write registers however it wishes, so the registers must be considered observable. On the other hand, if some other process $l'$ is executing, then the registers must be unobservable to $l$ if we hope to prove that $l$ and $l'$ are isolated. We conclude that registers should be observable to $l$ only in active states.

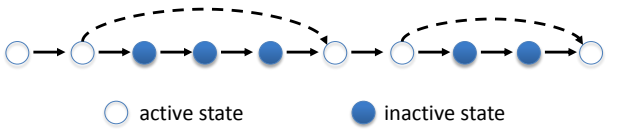


**Figure 4.** The TSysCall-local semantics, defined by taking big steps over the inactive parts of the TSysCall semantics.

What happens, then, if we attempt to prove that indistinguishability is preserved when starting from inactive indistinguishable states? Since the states are inactive, the registers are unobservable, and so the instruction pointer register in particular may have a completely different value in the two states. This means that the indistinguishable states may execute different instructions. If one state executes the yield primitive while the other does not, we may end up in a situation where one resulting state is active but the other is not; clearly, such states cannot be indistinguishable since the registers are observable in one state but not in the other. Thus indistinguishability will not be preserved in this situation.

The fundamental issue here is that, in order to prove that $l$ cannot be influenced by $l'$, we must show that $l$ has no knowledge that $l'$ is even executing. We accomplish this by defining a higher-level machine above the TSysCall machine, where every state is active. We call this the TSysCall-local machine — it is parameterized by principal $l$, and it represents $l$'s local view of the TSysCall machine.

Figure 4 shows how the semantics of TSysCall-local is defined. The solid arrows are transitions of the TSysCall machine, white circles are active TSysCall states, and shaded circles are inactive states. The TSysCall-local semantics is then obtained by combining all of the solid arrows connecting active states with all of the dotted arrows. Note that in the TSysCall layer, the yield primitive is the *only* way that a state can change from active to inactive, or vice-versa. Thus one can think of the TSysCall-local machine as a version of the TSysCall machine where the yield semantics takes a big step, immediately returning to the process that invoked the yield.

Our high-level security property is proved over the TSysCall-local machine, for *any* choice of observer principal $l$. We easily prove simulation from TSysCall-local to TSysCall, so this strategy fits cleanly into our simulation framework.

***Observation Function*** We now define the high-level observation function used in our verification. For a given process ID $l$, the state observation of $\sigma$ is defined as follows:

- *Registers* — All registers are observable if $\sigma$ is active.
- *Output* — The output buffer of $l$ is observable.

- *Virtual Address Space* — We can dereference any virtual address by walking through $l$'s page tables. This will result in a value if the address is actually mapped, or no value otherwise. This function from virtual addresses to option values is observable. Importantly, the physical address at which a value resides is never observable.
- *Spawned* — The spawned status of $l$ is observable.
- *Quota* — The remaining quota of $l$ is observable.
- *Children* — The number of children of $l$ is observable.
- *Active* — It is observable whether $\texttt{cid}(\sigma)$ is equal to $l$.
- *Reg Ctxt* — The saved register context of $l$ is observable.

Notice how the virtual address space component exploits the generality of observation functions. It is not simply a portion of program state, as it refers to both the `pgmap` and `HP` components of state in a nontrivial way.

## 5. Security Verification of mCertiKOS

To prove end-to-end security of mCertiKOS, we must apply Theorem 1 of Section 3, using the simulation from TSysCall-local to MBoot, the high-level observation function described in Section 4, and a low-level observation function that simply projects the output buffer. To apply the theorem, the following facts must be established:

1. MBoot is deterministic.
2. MBoot is behavioral for any principal.
3. The simulation relation from TSysCall-local to MBoot preserves indistinguishability.
4. TSysCall-local satisfies the high-level security property (Definition 3 of Section 3).

Determinism of the MBoot machine is already proved in mCertiKOS. Behaviorality of MBoot is easily established by defining a partial order over output buffers based on list prefix, and showing that every step of MBoot either leaves the buffer untouched or appends to the end of the buffer. To prove that the simulation preserves indistinguishability, we first prove that simulation between consecutive layers in mCertiKOS always preserves the output buffer. Indistinguishability preservation then follows, since the high-level observation function includes the output buffer as a component.

The primary task of the proof effort is, unsurprisingly, establishing the high-level unwinding condition over the TSysCall-local semantics. The proof is done by showing that each primitive of the TSysCall layer preserves indistinguishability. The yield primitive requires some special treatment since the TSysCall-local semantics treats it differently; this will be discussed later in this section.

Figure 5 gives the number of lines of Coq definitions and proof scripts required for the proof effort. The entire effort is broken down into security proofs for primitives, glue code to interface the primitive proofs with the LAsm($L$) semantics, definitions and proofs of the framework described

**Security of Primitives (LOC)**

| | |
|---|---|
| Load | 147 |
| Store | 258 |
| Page Fault | 188 |
| Get Quota | 10 |
| Spawn | 30 |
| Yield | 960 |
| Start User | 11 |
| Print | 17 |
| **Total** | **1621** |

**Security Proof (LOC)**

| | |
|---|---|
| Primitives | 1621 |
| Glue | 853 |
| Framework | 2192 |
| Invariants | 1619 |
| **Total** | **6285** |

**Figure 5.** Approximate Coq LOC of proof effort.

in Section 3, and proofs of new state invariants that were established. We will now discuss the most interesting aspects and difficulties of the TSysCall-local security proof.

***State Invariants*** While mCertiKOS already verifies a number of useful state invariants, some new ones are needed for our security proofs. The new invariants established over TSysCall-local execution are:

1. In all saved register contexts, the instruction pointer register points to the entry of the `start_user` primitive.
2. No page is mapped more than once in the page tables.
3. We are always either in user mode, or we are in kernel mode and the instruction pointer register points to the entry of the `start_user` primitive (meaning that we just yielded and are going to enter user mode in one step).
4. The sum of the available quotas (max quota minus usage) of all spawned processes is less than or equal to the number of unallocated pages in the heap.

***Security of Load/Store Primitives*** The main task for proving security of the 100+ assembly commands of LAsm(TSysCall) is to show that the TSysCall layer's load/store primitives preserve indistinguishability. This requires showing that equality of virtual address spaces is preserved. Reasoning about virtual address spaces can get quite hairy since we always have to consider walking through the page tables, with the possibility of faulting at either of the two levels.

To better understand the intricacies of this proof, consider the following situation. Suppose we have two states $\sigma_1$ and $\sigma_2$ with equal mappings of virtual addresses to option values (where no value indicates a page fault). Suppose we are writing to some virtual address $v$ in two executions on these states. Consider what happens if there exists some other virtual address $v'$ such that $v$ and $v'$ map to the same physical page in the first execution, but map to different physical pages in the second. It is still possible for $\sigma_1$ and $\sigma_2$ to have identical views of their virtual address space, as long as the two different physical pages in the second execution contain the same values everywhere. However, writing to $v$ will change the observable view of $v'$ in the first execution,

but not in the second. Hence, in this situation, it is possible for the store primitive to break indistinguishability.

We encountered this exact counterexample while attempting to prove security, and we resolved the problem by establishing the second state invariant mentioned above. The invariant guarantees that the virtual addresses $v$ and $v'$ will never be able to map to the same physical page.

***Security of Process Spawning*** The `proc_create` primitive was the only one whose security depended on making a major change to the existing mCertiKOS. When the insecure version of mCertiKOS creates a new child process, it chooses the lowest process ID not currently in use. The system call returns this ID to the user. Hence the ID can potentially leak information between different users. For example, suppose Alice spawns a child process and stores its ID into variable $x$. She then calls yield. When execution returns back to her, she again spawns a child and stores the ID into variable $y$. Since mCertiKOS always chooses the lowest available process ID, the value of $y - x - 1$ is exactly the number of times that other processes spawned children while Alice was yielded. In some contexts, this information leak could allow for direct communication between two different processes.

To close this information channel, we had to revamp the way process IDs are chosen in mCertiKOS-secure. The new ID system works as follows. We define a global parameter $m_c$ limiting the number of children any process is allowed to spawn. Suppose a process with ID $i$ and $c$ children ($c < m_c$) spawns a new child. Then the child's ID will always be $i{*}m_c{+}c{+}1$. This formula guarantees that different processes can never interfere with each other via child ID: if $i \neq j$, then the set of possible child IDs for process $i$ is completely disjoint from the set of possible child IDs for process $j$.

***Security of Page Fault*** There are two different interesting aspects of page fault security. The first is that it is a perfect example of a primitive whose implementation does not preserve indistinguishability with each individual step. When a page fault occurs, either one or two new pages must be allocated from the global heap. Because all user processes use the same global heap for allocation, and mCertiKOS always allocates the first available page, the physical address of an allocated page can potentially leak information between processes (this is similar to the information leak via process ID discussed above). The page fault handler, however, must make use of the physical address of a newly-allocated page when inserting a new virtual address mapping into the page table. Therefore, at some point during the actual (non-atomic) execution of the page fault handler, an observable register contains the insecure physical page address. Since we prove that the primitive's atomic specification is secure, however, we know that the insecure value must be overwritten by the time the primitive finishes executing.

The second interesting aspect of the page fault handler involves some trickiness with running out of heap space. In particular, the global allocation table `AT` must be unobservable since all processes can affect it; this means that the page fault handler may successfully allocate a page in one execution, but fail to allocate a page in an execution from an indistinguishable state due to there being no pages available. Clearly, the observable result of the primitive will be different for these two executions. To resolve this issue, we relate available heap pages to available quota by applying the fourth state invariant mentioned above. Recall that the invariant guarantees that the sum of the available quotas of all spawned processes is always less than or equal to the number of available heap pages. Therefore, if an execution ever fails to allocate a page because no available page exists, the available quota of *all* spawned processes must be zero. Since the available quota is observable, we see that allocation requests will be denied in both executions from indistinguishable states. Therefore, we actually *can* end up in a situation where one execution has pages available for allocation while the other does not; in both executions, however, the available quota will be zero, and so the page allocator will deny the request for allocation.

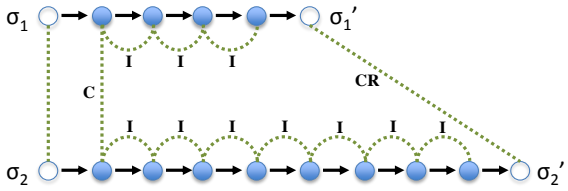


**Figure 6.** Applying the three lemmas to prove the security property of TSysCall-local yielding.

***Security of Yield*** Yielding is by far the most complex primitive to prove secure, as the proof requires reasoning about the relationship between the TSysCall semantics and TSysCall-local semantics. Consider Figure 6, where active states $\sigma_1$ and $\sigma_2$ are indistinguishable, and they both call yield. The TSysCall-local semantics takes a big step over the executions of all non-observer processes; these big steps are unfolded in Figure 6, so the solid arrows are all of the individual steps of the TSysCall semantics. We must establish that a big-step yield of the TSysCall-local machine preserves indistinguishability, meaning that states $\sigma_1'$ and $\sigma_2'$ in Figure 6 must be proved indistinguishable.

We divide this proof into three separate lemmas, proved over the TSysCall semantics:

- *Confidentiality* — If two indistinguishable active states take a step to two inactive states, then those inactive states are indistinguishable.
- *Integrity* — If an inactive state takes a step to another inactive state, then those states are indistinguishable.
- *Confidentiality Restore* — If two indistinguishable inactive states take a step to two active states, then those active states are indistinguishable.

These lemmas are chained together as pictured in Figure 6. The dashed lines indicate indistinguishability. Thus the confidentiality lemma establishes indistinguishability of the initial inactive states after yielding, the integrity lemma establishes indistinguishability of the inactive states immediately preceding a yield back to the observer process, and the confidentiality restore lemma establishes indistinguishability of the active states after yielding back to the observer process.

Note that while the confidentiality and confidentiality restore lemmas apply specifically to the yield primitive (since it is the only primitive that can change active status), the integrity lemma applies to all primitives. Thus, like the security unwinding condition, integrity is proved for each of the TSysCall primitives. The integrity proofs are simpler since the integrity property only requires reasoning about a single execution, whereas security requires comparing two.

The confidentiality restore lemma only applies to the situation where two executions are both yielding back to the observer process. The primary obligation of the proof is to show that if the saved register contexts of two states $\sigma_1$ and $\sigma_2$ are equal, then the actual registers of the resulting states $\sigma_1'$ and $\sigma_2'$ are equal. There is one interesting detail related to this proof: a context switch in mCertiKOS does not save *every* machine register, but instead only saves those registers that are relevant to the local execution of a process (e.g., EAX, ESP, etc.). In particular, the CR2 register, which the page fault handler primitive depends on, is not saved. This means that, immediately after a context switch from some process $i$ to some other process $j$, the CR2 register could contain a virtual address that is private to $i$. How can we then guarantee that $j$ is not influenced by this value? Indeed, if process $j$ immediately calls the page fault handler without first triggering a page fault, then it may very well learn some information about process $i$. We resolve this insecurity by making a very minor change to mCertiKOS: we add a line of assembly code to the implementation of context switch that clears the CR2 register to zero.

## 6. Limitations and Future Work

***Fidelity of the Assembly Machine Model*** Our methodology only yields a security proof for assembly programs that fit within our model of x86 assembly execution. The model is an extended version of CompCert's, primarily designed for supporting all of the features needed by the mCertiKOS kernel implementation (e.g., distinction between kernel and user mode executions). We make no claims about the relationship between our model and the physical hardware that executes x86 assembly code. If one wished to apply our proof to the actual machine execution, the following significant gaps would need to be closed:

- *Completeness Gap* — Our model is certainly not complete for all user-level assembly programs, so it may be possible to violate security on actual hardware by exploiting unmodeled assembly instructions. One example of such an instruction is RDTSC, which reads the time-stamp counter from x86 hardware and stores it into local registers. This time-stamp counter is increased with each clock cycle, so it can be used as a communication channel between processes, leaking information about how much time certain processes have spent executing. We do not model the RDTSC instruction — an assembly program that uses the instruction would not even be considered valid syntax in our model, so there is no way that any verified properties could apply to such a program.
- *Soundness Gap* — In addition to this completeness gap, there is also a potential soundness gap between our machine model and the physical hardware; we must trust that the semantics of all of our modeled assembly instructions are faithful to the actual hardware execution. This is a standard area of trust that arises in any formal verification effort: at some point, we always reach a low-enough level where trust is required, whether this means trusting the operating system that a program is running on, trusting the hardware to meet its published specifications, or trusting the laws of physics that the hardware is presumably obeying. Note that the level of trustworthiness of our machine model is similar to CompCert's, since we use a modest extension over CompCert's model.
- *Safety Gap* — The soundness gap just described requires us to trust that whenever the modeled semantics of an assembly instruction is well-defined, the execution of that instruction on physical hardware will do what the model says. What happens, however, if the modeled semantics gets stuck? The model makes no promises about the actual execution of a stuck semantics; the execution could continue running without issues, but it would no longer be bound by any of our verification. Therefore, even if we closed the completeness and soundness gaps described above to a point of satisfaction, we would still be required to assume that user programs never have undefined semantics in order to apply our verification to the physical execution. This is quite a heavyweight assumption, as user-level code is meant to represent arbitrary and unverified assembly.

***Future Plans for Model Fidelity*** In light of these various unrealistic assumptions required to apply our verification to the physical machine, we are planning to implement a clearer and more streamlined representation of user-mode assembly execution. The mCertiKOS assembly model was designed for verification of the kernel; there is actually no need to use that model for unverified user process execution. Instead, we can design a simple model consisting of registers and a flat memory representing a virtual address space, where an instruction can be one of the following:

- *interrupt* — A trap into the kernel to handle, for example, a privileged instruction or a system call.

- *load/store* — Instructions that use the kernel's load/store primitives to access the virtual address space. These may trigger a page fault, which will be handled by the kernel.
- *other* — Any other user-land instruction, which is assumed to only be able to read/write the values in registers.

This simple model has the benefit of making very clear exactly what assumption needs to hold in order to relate the model to actual execution: the arbitrary user-land instructions must only depend upon and write values in the modeled registers. Notice that the RDTSC instruction described above is an example of an instruction that does *not* satisfy this assumption; hence it would need to be explicitly modeled if we wanted to support it.

We hope that future work can gradually model more and more hardware features and instructions like RDTSC that do not satisfy this assumption. Each new feature could potentially violate security, and thus will require some additional verification effort. For the RDTSC example, we would close the time-stamp counter information channel by setting the time-stamp disable flag (TSD), which causes the hardware to treat RDTSC as a privileged instruction. Then, if a user process attempts to execute the instruction, the hardware will generate an exception and trap into the kernel. The kernel will then handle the exception in a way that is verified to be secure (e.g., it could kill the process, yield to a different process, or return a specialized "local" time-stamp that only represents that specific process's view of time).

***High-Level Policy Specification*** As with any formal verification effort, we must trust that the top-level specification of our system actually expresses our intentions for the system, including the security policy specified as an observation function. Because observation functions can have any type, our notion of security is far more expressive than classical pure noninterference. This does mean, however, that it can potentially be difficult to comprehend the security ramifications of a complex or poorly-constructed observation function. We place the onus on the system verifier to make the observation function as clear and concise as possible. This view is shared by a number of previous security frameworks with highly-expressive policy specification, such as the PER model [25] and Relational Hoare Type Theory [22]. In our mCertiKOS security specification, the virtual address space observation provides a good example of a nontrivial but clear policy specification — hiding physical addresses is, after all, the primary reason to use virtual address spaces.

***Applicability of the Methodology*** In order to utilize our security methodology, the following steps must be taken:

- The high-level security policy must be expressed as isolation between the observation functions of different principals. As mentioned previously, the complete lack of restrictions on the observation function yields a very high level of policy expressiveness. While a systematic exploration of expressiveness remains to be done, we have not encountered any kinds of information flows that are not expressible in terms of an observation function.
- The high-level security property (Definition 3) must be provable over the top-level semantics. In particular, this means that indistinguishability must be preserved on a step-by-step basis. If it is not preserved by each individual step, then the top-level semantics must be abstracted further. For example, in our mCertiKOS security verification, we found that the TSysCall semantics did not preserve indistinguishability on a step-by-step basis; we therefore abstracted it further into the TSysCall-local semantics that hides the executions of non-observer processes. We are unsure if this requirement for single-step indistinguishability preservation could be problematic for other systems. In our experience, however, repeated abstraction to the point of atomicity is highly desirable, as it yields a clear view of what is really going on in the system.
- Indistinguishability-preserving simulations must be established to connect the various levels of abstraction. While the main simulation property can require significant effort, we have not found the indistinguishability preservation property to be difficult to establish in practice. The property generally feels closer to a sanity check than a significant restriction. Consider, for instance, the example of the `swap` primitive from Section 2.3. That example failed to preserve security across simulation because the local variable `z` was being considered observable. A caller of the `swap` primitive should have no knowledge of `z`, however. Thus this is just a poorly-constructed observation function; a reasonable notion of observation would hide the local variable, and indistinguishability preservation would follow naturally.

***Inter-Process Communication*** The mCertiKOS verification presented in this work only applies to a version of the kernel that disables IPC. In the future, we would like to allow some well-specified and disciplined forms of IPC that can still be verified secure. We have actually already started adding IPC — our most recent version of the secure kernel includes an IPC primitive that allows communication between all processes with ID at most $k$ (a parameter that can be modified). The security theorem then holds for any observer process with ID greater than $k$. Ideally, we would like to extend this theorem so that it guarantees some nontrivial properties about those privileged processes with low ID.

## 7. Related Work and Conclusions

***Observations and Indistinguishability*** Our flexible notion of observation is similarly powerful to purely semantic and relational views of indistinguishability, such as the ones used in Sabelfeld et al.'s PER model [25] and Nanevski et al.'s Relational Hoare Type Theory [22]. In those systems, for example, a variable $x$ is considered observable if its value is equal in two related states. In our system, we directly say that

$x$ is an observation, and then indistinguishability is defined as equality of observations. Our approach may at first glance seem less expressive since it uses a specific definition for indistinguishability. However, we do not put any restrictions on the type of observation: for any given indistinguishability relation $R$, we can represent $R$ by defining the observation function on $\sigma$ to be the set of states related to $\sigma$ by $R$. We have not systematically explored the precise extent of policy expressiveness in our methodology; this could be an interesting direction for future work.

Our observation function is a generalization of the "conditional labels" presented in Costanzo and Shao [3]. There, everything in the state has an associated security label, but there is allowed to be arbitrary dependency between values and labels. For example, a conditional label may say that $x$ has a low label if its value is even, and a high label otherwise. In the methodology presented here, we do not need the labels at all: the state-dependent observation function observes the value of $x$ if it is even, but observes no value if $x$ is odd.

Our approach is also a generalization of Delimited Release [23] and Relaxed Noninterference [16]. Delimited Release allows declassifications only according to certain syntactic expressions (called "escape hatches"). Relaxed Noninterference uses a similar idea, but in a semantic setting: a security label is a function representing a declassification policy, and whenever an unobservable variable $x$ is labeled with function $f$, the value $f(x)$ is considered to be observable. Our observation function can easily express both of these concepts of declassification.

***Security across Simulation/Refinement*** As explained in Sections 1 and 2, refinements and simulations may fail to preserve security. There have been a number of solutions proposed for dealing with this so-called refinement paradox [11, 17, 18]. The one that is most closely related to our setup is Murray et al.'s seL4 security proof [19], where the main security properties are shown to be preserved across refinement. As we mentioned in Section 2, we employ a similar strategy for security preservation in our framework, disallowing high-level specifications from exhibiting domain-visible nondeterminism. Because we use an extremely flexible notion of observation, however, we encounter another difficulty involved in preserving security across simulation; this is resolved with the natural solution of requiring simulation relations to preserve state indistinguishability.

***Comparison with mCertiKOS-base*** Our verified secure kernel builds directly over the "base" version of mCertiKOS presented in Gu et al. [9]. In that version, the many layers of mCertiKOS are connected using CompCert-style simulations, and CompCertX is used to integrate C primitives with assembly primitives. However, that version does not have general notions of observations, events, or behaviors. Technically, CompCert expresses external events using traces that appear on the transition functions of operational semantics, and then defines whole-execution behaviors in terms of events; however, mCertiKOS does not make use of these events (the LAsm semantics completely ignores CompCert traces).

Separately from the security verification effort, a large portion of our work was devoted to developing the framework of generalized observations and indistinguishability-preserving simulations described in Sections 2 and 3 (over 2000 lines of Coq code, as shown in Figure 5), and integrating these ideas into mCertiKOS. The previous mCertiKOS soundness theorem in Gu et al. [9] only claimed a standard simulation between TSysCall and MBoot. We integrated observation functions into the mCertiKOS layers, modified this soundness theorem to establish an indistinguishability-preserving simulation between TSysCall and MBoot, and then defined whole-execution behaviors and proved an extended soundness theorem guaranteeing that the behaviors of executions at the TSysCall level are identical to those of corresponding executions at the MBoot level.

***Security of seL4*** An important work in the area of formal operating system security is the seL4 verified kernel [12, 19, 20, 26]. There are some similarities between the security proof of seL4 and that of mCertiKOS, as both proofs are conducted over a high-level specification and then propagated down to a concrete implementation. Our work, however, has three important novelties over the seL4 work.

First, the seL4's lack of assembly verification is quite significant. Our mCertiKOS kernel consists of 354 lines of assembly code and approximately 3000 lines of C code. Thus the assembly code represents a nontrivial chunk of the codebase that could easily contain security holes. Furthermore, the assembly code has to deal with low-level hardware details like registers, which are not exposed to high level specifications and might have security holes. Indeed, as discussed in Section 5, we needed to patch up a security hole in the context switch primitive related to the CR2 register.

Second, our assembly-level machine is a much more realistic model than the abstract C-level machine used by seL4. For example, virtual memory address translation, page fault handlers, and context switches are not verified in seL4. Section 5 describes the intricacies of security of load/store primitives (with address translation), page fault handler, and yield. None of them would appear in the seL4 proofs because their machine model is too high level. Addressing this issue is not easy because it requires not just assembly verification but also verified linking of C and assembly components.

Third, our generalization of the notion of observation allows for highly expressive security policies. The seL4 verification uses a particular policy model based on intransitive noninterference (the intransitive part helps with specifying what IPC is allowed). Our mCertiKOS verification is a case study using the particular policy expressed by the observation function of Section 4.2, but our methodology allows for all kinds of policy models depending on context. Thus, while the particular security property that we proved over mCertiKOS is not an advance over the seL4 security property,

our new methodology involved in stating and proving the property, and for propagating security proofs through verified compilation and abstraction layers, is a significant advance.

***Security of Other OS Kernels*** Dam et al. [5] aims to prove isolation of separate components that are allowed to communicate across authorized channels. They do not formulate security as standard noninterference, since some communication is allowed. Instead, they prove a property saying that the machine execution is trace-equivalent to execution over an idealized model where the communicating components are running on physically-separated machines. Their setup is fairly different from ours, as we disallow communication between processes and hence prove noninterference. Furthermore, they conduct all verification at the assembly level, whereas our methodology supports verification and linking at both the C and assembly levels.

The Ironclad [10] system aims for full correctness and security verification of a system stack, which shares a similar goal to ours: provide guarantees that apply to the low-level assembly execution of the machine. The overall approaches are quite different, however. Ironclad uses Dafny [13] and Z3 [6] for verification, whereas our approach uses Coq; this means that Ironclad relies on SMT solving, which allows for more automation, but does not produce machine-checkable proofs as Coq does. Another difference is in the treatment of high-level specifications. While Ironclad allows some verification to be done in Dafny using high-level specifications, a trusted translator converts them into low-level specifications expressed in terms of assembly execution. The final security guarantee applies only to the assembly level; one must trust that the guarantee corresponds to the high-level intended specifications. Contrast this to our approach, where we verify that low-level execution conforms to the high-level policy.

***Conclusions*** In this paper, we presented a framework for verifying end-to-end security of C and assembly programs. A flexible observation function is used to specify the security policy, to prove noninterference via unwinding, and to soundly propagate the security guarantee across simulation. We demonstrated the efficacy of our approach by verifying the security of a nontrivial operating system kernel, with IPC disabled. We successfully developed a fully-formalized Coq proof that guarantees security of the low-level model of the kernel's assembly execution.

## Acknowledgments

We thank Benjamin Pierce, members of the CertiKOS team at Yale, our shepherd Stephen McCamant, and anonymous referees for helpful comments and suggestions that improved this paper and the implemented tools. This research is based on work supported in part by NSF grants 1065451, 1319671, and 1521523, and DARPA grants FA8750-10-2-0254, FA8750-12-2-0293, and FA8750-15-C-0082. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

## References

[1] S. Blazy and X. Leroy. Mechanized semantics for the Clight subset of the C language. *J. Automated Reasoning*, 43(3): 263–288, 2009.

[2] S. Chiricescu, A. DeHon, D. Demange, S. Iyer, A. Kliger, G. Morrisett, B. C. Pierce, H. Reubenstein, J. M. Smith, G. T. Sullivan, A. Thomas, J. Tov, C. M. White, and D. Wittenberg. Safe: A clean-slate architecture for secure systems. In *Proceedings of the IEEE International Conference on Technologies for Homeland Security*, Nov. 2013.

[3] D. Costanzo and Z. Shao. A separation logic for enforcing declarative information flow control policies. In *Proc. 3rd International Conference on Principles of Security and Trust (POST)*, pages 179–198, 2014.

[4] D. Costanzo, Z. Shao, and R. Gu. End-to-end verification of information-flow security for C and assembly programs (extended version). Technical Report YALEU/DCS/TR-1522, Dept. of Computer Science, Yale University, April 2016.

[5] M. Dam, R. Guanciale, N. Khakpour, H. Nemati, and O. Schwarz. Formal verification of information flow security for a simple ARM-based separation kernel. In *2013 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 223–234, 2013.

[6] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference (TACAS), Budapest, Hungary. Proceedings*, pages 337–340, 2008.

[7] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[8] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *Proceedings of the 1984 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 29 - May 2, 1984*, pages 75–87, 1984.

[9] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. N. Wu, S. Weng, H. Zhang, and Y. Guo. Deep specifications and certified abstraction layers. In *Proc. 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), Mumbai, India*, pages 595–608, 2015.

[10] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad apps: End-to-end security via automated full-system verification. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Broomfield, CO, USA*, pages 165–181, 2014.

[11] J. Jürjens. Secrecy-preserving refinement. In *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, Berlin, Germany, March 12-16, 2001, Proceedings*, pages 135–152, 2001.

[12] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1), Feb. 2014.

[13] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial*

*Intelligence, and Reasoning (LPAR) - 16th International Conference, Dakar, Senegal*, pages 348–370, 2010.

[14] X. Leroy. The CompCert verified compiler. `http://compcert.inria.fr/`, 2005–2014.

[15] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[16] P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *Proc. 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), Long Beach, California, USA*, pages 158–170, 2005.

[17] C. Morgan. The shadow knows: Refinement and security in sequential programs. *Sci. Comput. Program.*, 74(8):629–653, 2009.

[18] C. Morgan. Compositional noninterference from first principles. *Formal Asp. Comput.*, 24(1):3–26, 2012.

[19] T. C. Murray, D. Matichuk, M. Brassil, P. Gammie, and G. Klein. Noninterference for operating system kernels. In *Certified Programs and Proofs (CPP) - Second International Conference, Kyoto, Japan, Proceedings*, pages 126–142, 2012.

[20] T. C. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. sel4: From general purpose to a proof of information flow enforcement. In *IEEE Symposium on Security and Privacy*, pages 415–429, 2013.

[21] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. 1997 ACM Symposium on Operating System Principles (SOSP)*, pages 129–142, 1997.

[22] A. Nanevski, A. Banerjee, and D. Garg. Verification of information flow and access control policies with dependent types. In *IEEE Symposium on Security and Privacy*, pages 165–179, 2011.

[23] A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Software Security - Theories and Systems, Second Mext-NSF-JSPS International Symposium (ISSS), Tokyo, Japan*, pages 174–191, 2003.

[24] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[25] A. Sabelfeld and D. Sands. A Per model of secure information flow in sequential programs. In *Programming Languages and Systems, 8th European Symposium on Programming (ESOP), Amsterdam, The Netherlands, Proceedings*, pages 40–58, 1999.

[26] T. Sewell, S. Winwood, P. Gammie, T. C. Murray, J. Andronick, and G. Klein. seL4 enforces integrity. In *Interactive Theorem Proving (ITP) - Second International Conference, Berg en Dal, The Netherlands, Proceedings*, pages 325–340, 2011.

[27] The Coq development team. The Coq proof assistant. `http://coq.inria.fr`, 1999 – 2015.

# A. Appendix: Example Security Policies

To provide some more intuition on the expressiveness of observation functions, we will briefly present some examples of high-level security policies here.

## A.1 Example 1: Declassify Parity

As a simple starting example, we will go into some more detail on the parity observation function mentioned in Section 2. Suppose we have the following `add` function:

```
void add() {
    a = x + y;
    b = b + 2; }
```

We write the atomic specification of `add` as a relation between input state and output state:

$$(\sigma, \sigma') \in S_{\mathsf{add}} \iff \sigma' = \sigma[a \hookrightarrow \sigma(x) + \sigma(y);\ b \hookrightarrow \sigma(b) + 2]$$

We specify Alice's security policy as an observation function:

$$\mathcal{O}_A(\sigma) \triangleq [a \hookrightarrow \sigma(a)\%2;\ x \hookrightarrow \sigma(x)\%2;\ y \hookrightarrow \sigma(y)\%2]$$

As explained in Sections 2 and 3, we prove security by showing that state indistinguishability is preserved by the high-level semantics. In this example, we assume that the specification of `add` constitutes the entirety of the machine semantics. Hence we must prove:

$$\mathcal{O}_A(\sigma_1) = \mathcal{O}_A(\sigma_2) \wedge (\sigma_1, \sigma_1') \in S_{\mathsf{add}} \wedge (\sigma_2, \sigma_2') \in S_{\mathsf{add}} \\ \implies \mathcal{O}_A(\sigma_1') = \mathcal{O}_A(\sigma_2')$$

This reduces to:

$$[a \hookrightarrow \sigma_1(a)\%2;\ x \hookrightarrow \sigma_1(x)\%2;\ y \hookrightarrow \sigma_1(y)\%2] = \\ [a \hookrightarrow \sigma_2(a)\%2;\ x \hookrightarrow \sigma_2(x)\%2;\ y \hookrightarrow \sigma_2(y)\%2] \\ \implies \\ [a \hookrightarrow (\sigma_1(x) + \sigma_1(y))\%2;\ x \hookrightarrow \sigma_1(x)\%2;\ y \hookrightarrow \sigma_1(y)\%2] = \\ [a \hookrightarrow (\sigma_2(x) + \sigma_2(y))\%2;\ x \hookrightarrow \sigma_2(x)\%2;\ y \hookrightarrow \sigma_2(y)\%2]$$

Since $(a + b)\%2 = (a\%2 + b\%2)\%2$, we see that the atomic specification of `add` is indeed secure with respect to Alice's observation function. Therefore, we are guaranteed that `add` cannot leak any information about program state to Alice beyond the parities of the values in variables $a$, $x$, and $y$.

## A.2 Example 2: Event Calendar Objects

The next example demonstrates modularity of the observation function. Suppose we have a notion of calendar object where various events are scheduled at time slots numbered from 1 to $N$. At each time slot, the calendar contains either `None` representing no event, or `Some` $v$ representing an event whose details are encoded by integer $v$. A program state consists of a calendar object for each principal:

$$\text{calendar } \mathcal{C} \triangleq \mathbb{N} \to \texttt{option}\ \mathbb{Z}$$
$$\text{state } \Sigma \triangleq \mathcal{L} \to \mathcal{C}$$

We define an observation function, parameterized by an observer principal, describing the following policy:

1. Each principal can observe the entire contents of his or her own calendar.
2. Each principal can observe only whether or not time slots are free in other principals' calendars, and hence cannot be influenced by the details of others' scheduled events.

For simplicity, we define the type of observations to be the same as the type for program state ($\Sigma$). For readability, we write $\sigma(l, n)$ to indicate the option event located at slot $n$ of $l$'s calendar in state $\sigma$.

$$\mathcal{O}_l(\sigma) \triangleq \lambda l' \,.\, \lambda n \,.\, \begin{cases} \sigma(l', n), & \text{if } l' = l \\ \texttt{None}, & \text{if } l' \neq l \wedge \sigma(l', n) = \texttt{None} \\ \texttt{Some } 0, & \text{if } l' \neq l \wedge \sigma(l', n) \neq \texttt{None} \end{cases}$$

This observation function only reveals details of scheduled events in a calendar to the calendar's owner, and therefore allows a principal to freely modify his or her own calendar securely. If different principals wish to collaborate in some way, we must verify that such collaboration is secure with respect to this observation function. For example, consider a function `sched` that attempts to schedule some common event among a set of principals. Given a list of principals $L$ and an event $e$, the function will search for the earliest time slot $n$ that is free for all principals in $L$. If such a time slot is found, then all of the involved principals' calendars are updated with event $e$ scheduled at slot $n$. Otherwise, all calendars are unchanged. The following is pseudocode, and operates over a program state that contains an implementation of the per-principal calendars ($\Sigma$) in the array `cals`:

```
void sched(list[int] L, int e) {
    freeSlot = 0;
    for i = 1 to N {
       allFree = true;
       for j = 1 to |L| {
          if (cals[L[j]][i] != None) {
             allFree = false;
             break;
          }
       }
       if (allFree) {
          freeSlot = i;
          break;
       }
    }

    if (freeSlot != 0) {
       for i = 1 to |L|
          cals[L[i]][freeSlot] = Some e;
    }
}
```

With some effort, one can verify that this implementation of `sched` satisfies the high-level specification described above (i.e., the function schedules the new event in the principals' calendars if they all share an available time slot, or does nothing otherwise). Once we have the atomic specification, we can verify that it is secure for all principals, with respect to the observation function defined above. We will not go through details of the security proof here, but the general intuition should be clear: the behavior of `sched` is only dependent on the availability of time slots (i.e., the `None`/`Some` status); the specific details of scheduled events are never used.

### A.3 Example 3: Security Labels and Dynamic Tainting

Our third example concerns dynamic labels and tainting, discussed briefly in Section 2. Even though the observation function is statically defined for an entire execution, we can use dynamic labels to change the observability of data during an execution. Assume we have a lattice of security labels $\mathbb{L}$, with the set of possible labels being a superset of principals $\mathcal{L}$. Let program state be a function mapping variables to a pair $(v, l)$ of integer value $v$ and security label $l$. For a given principal $p$, the observation function will only reveal values that have a security label less than or equal to $p$ in the lattice:

$$\mathcal{O}_p(\sigma) \triangleq \lambda x \,.\, \begin{cases} (v, l), & \text{if } \sigma(x) = (v, l) \wedge l \sqsubseteq p \\ (0, l), & \text{if } \sigma(x) = (v, l) \wedge l \not\sqsubseteq p \end{cases}$$

We can now consider primitives that dynamically change the observability of data by propagating labels. For example, consider a function `add` that takes two parameters $a$ and $b$, and updates variable $x$ to have a value equal to the sum of their values, and a label equal to the least upper bound of their labels. Assuming a direct implementation of labeled integers as objects, the pseudocode will look like:

```
void add(lbl_int a, lbl_int b) {
    x.val = a.val + b.val;
    x.lbl = a.lbl ⊔ b.lbl }
```

The atomic specification of `add` is:

$$(\sigma, \sigma') \in S_{\texttt{add}} \iff \sigma' = \sigma[x \hookrightarrow (\sigma(a).1 + \sigma(b).1,\ \sigma(a).2 \sqcup \sigma(b).2)]$$

The security proof for `add` is straightforward. If two initial states $\sigma_1$ and $\sigma_2$ have equal observations for principal $p$, then there are two possibilities. First, if both of the labels of $a$ and $b$ (in states $\sigma_1$ and $\sigma_2$) are less than or equal to $p$, then indistinguishability tells us that $\sigma_1(a) = \sigma_2(a)$ and $\sigma_1(b) = \sigma_2(b)$. Hence the sum of their values in the two executions will be the same, and so the resulting final states are indeed indistinguishable. Second, if at least one of the labels is not less than or equal to $p$, then the least upper bound of the labels is also not less than or equal to $p$. Hence the observation of $x$ on the final states will be a value of 0, and so the final states are indistinguishable.

We could go further here and build an entire label-aware execution environment. Proving security of the high-level specifications is a similar process to proving soundness in other label-aware systems. We could then either treat the labels as purely logical state (like many statically-typed security systems), erasing them with a simulation relation, or we could verify a refinement to a machine like the one used in the SAFE system [2], where labels are actually implemented in the hardware and the physical machine performs dynamic label checks and tainting. Regardless of this choice of label representation, as long as we make sure our simulation relation preserves indistinguishability (as discussed in Section 2), the security of the high-level specifications will automatically give us the whole-execution noninterference property for the low-level machine.