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

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 practical, general, and 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. Some parts of the operating system are written in C and some are written in assembly; we verify all of the code, regardless of language.

1. Introduction

Information flow control (IFC) [20, 22] 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 operating system) consists of many separate functions (e.g., system call primitives) written in either C or assembly. Each primitive has a verified atomic specification,

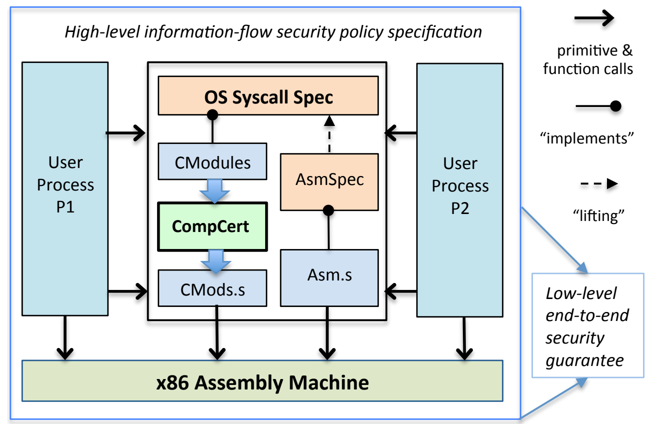

Figure 1. An end-to-end software system that consists of both OS modules (in C and assembly) and user processes.

and there is a verified compiler, such as CompCert [13], that can correctly compile C programs into assembly. We wish to prove an *end-to-end* security statement about some context program that can call the primitives of the system, which ultimately guarantees that the concrete execution (i.e., the whole-program assembly execution) behaves securely. There are many challenges involved, such as:

- *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 primitive 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 [10, 16] 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?
- *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 practical, general, and novel methodology for formally verifying end-to-end security of a system like the one shown in Figure 1. First, security is proved for each high-level specification in a standard way, establishing noninterference by showing that a state indistinguishability relation is invariant across the specification (this is the standard *unwinding condition* [6, 7]). Then we apply simulation techniques to automatically obtain a sound security guarantee for the low-level machine execution, which is expressed in terms of whole-execution observations. Simulations are used both for connecting specifications with implementations, as well as for connecting C implementations with 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 observations. 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 [8] to prove security between user processes with distinct IDs. mCertiKOS guarantees full functional correctness of system calls 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 an end-to-end security guarantee for the operating system — we specify exactly which portions of high-level state are observable to which processes, and we are guaranteed that the low-level assembly execution of the whole system is secure with respect to this policy.

Note that, while most of the mCertiKOS code is written in C and compiled into assembly by CompCert, there are some primitives that must be written directly in assembly. Context switch is a classic example of such a primitive. The primary functionality of a context switch involves copying values between machine registers and kernel objects; hence both the implementation and specification of the primitive must refer to low-level machine state. A key goal and contribution of our security verification is that assembly primitives like context switching are fully handled within the framework, and thus do not need to be trusted.

To summarize, the primary contributions of this work are:

- A novel methodology for end-to-end security verification of complex systems written in different languages extending across various levels of abstraction.
- An end-to-end security proof, completely formalized in the Coq proof assistant [26], of a nontrivial operating system. Some parts of the operating system are written in C and some are written in assembly; we verify *all* of the code, regardless of language.

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. Finally, we discuss related work and then conclude.

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(e.g., in the context of the discussion in Section 1, M could represent single-step execution of atomic specifications, C implementations, or assembly implementations). 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 σ, 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 primitive (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:

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

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` primitive 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:

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

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:

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

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.

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.

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 an execution, where two states are said to be indistinguishable just when their observations are equal:

$$\sigma_1 \overset{l}{\sim} \sigma_2 \ \triangleq\ \mathcal{O}_l(\sigma_1) = \mathcal{O}_l(\sigma_2)$$

Intuitively, if a step of execution always preserves indistinguishability, then the final observation of the step can be 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 σ_1, σ_2, σ_1', and σ_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` primitive 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 primitive 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 Notice that the non-atomic implementation of `add` also satisfies the above security property. That is, 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 example 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.

We will express low-level security as an equality between the whole-execution observations produced by two executions starting from indistinguishable states. There are two challenges involved in formalizing this definition, relating to indistinguishability and whole-execution observations.

Low-Level Indistinguishability For high-level security, we defined state indistinguishability to be equality of the state-parameterized observations. This definition may not make sense at a lower level of abstraction, however. For example, suppose we attach security labels to data in a high-level state, for the purpose of specifying a policy based on label tainting (described above). Further suppose that we treat the labels as purely logical, erasing them when simulating the high-level specification with an implementation. This means that the observation function of the implementation machine cannot be dependent on security labels in any way, and hence equality of observations is not a sensible notion of indistinguishability at the implementation level.

We solve this challenge by defining low-level state indistinguishability in terms of high-level indistinguishability and simulation. We say that, given a simulation relation R relating 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 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 [10, 16, 17], 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 σ_1 and σ_2 are identical:

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

The successful security verifications of both seL4 and mCertiKOS provide solid 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 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

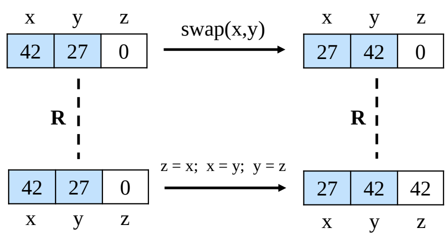

Figure 2. Security-Violating Simulation. The shaded part of state is unobservable, while the unshaded part is observable.

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 σ_1, σ_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 some notion of security is preserved across simulation.

Machines with Observations In the following, assume we have a set $\mathcal{L}$ of isolated 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 Σ_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 Ω_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. The following describes standard ways to support such concepts within program state:

- *input events* — We can represent inputs as an infinite stream (oracle) stored within program state. Whenever a program needs to read input, it dequeues the head of this stream.
- *output events* — Outputs can be represented as a finite list (output buffer). As mentioned in Section 2.2, we will actually use a more general notion of output events, defined in terms of a monotonic observation function.
- *instructions* — Our model of program state will include everything relevant for execution. For assembly execution, this includes machine registers and the memory where code is stored; thus instructions will be implicitly represented in the program semantics by looking up and dereferencing the value of the instruction pointer register. For execution in a higher-level language, the state will include an explicit representation of the code being executed (i.e., we consider the program state to be what many other systems would call the whole-machine *configuration*).

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.

Safety and Determinism We will often need to refer to two additional concepts: safety (relative to an invariant) and determinism. In our context, we define these as follows:

Definition 2 (Safety)**.** We say that a machine M is safe under state predicate I, written $\square M^I$, 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 (Determinism)**.** We say that a machine M is deterministic, written $\downarrow M$, when the following properties hold:

1.) $\forall \sigma, \sigma', \sigma'' \;.\; \sigma \mapsto \sigma' \wedge \sigma \mapsto \sigma'' \implies \sigma' = \sigma''$
2.) $\forall \sigma \in F_M \;.\; \neg \exists \sigma' \;.\; \sigma \mapsto \sigma'$

High-Level Security We can now formally define what it means for a machine to be secure with respect to its observation function. As discussed in Section 2, we must consider two definitions of security: a high-level notion based on the standard unwinding condition, and a low-level notion based on whole-execution behaviors.

We begin with the high-level definition. As mentioned above, we will actually only prove the property for program states that have been properly initialized. Hence safety under the initialization invariant must be proved prior to security.

Definition 4 (High-Level Security)**.** Machine M is secure for principal l under invariant I, written ΔM_l^I, just when:

$$
\begin{array}{ll}
1.) & \Box M^I \\
2.) & \forall \sigma_1, \sigma_2 \in I,\ \sigma_1', \sigma_2' \ . \\
& \quad \mathcal{O}_l(\sigma_1) = \mathcal{O}_l(\sigma_2) \wedge \sigma_1 \mapsto \sigma_1' \wedge \sigma_2 \mapsto \sigma_2' \\
& \quad \Longrightarrow \mathcal{O}_l(\sigma_1') = \mathcal{O}_l(\sigma_2') \\
3.) & \forall \sigma_1, \sigma_2 \in I \ . \\
& \quad \mathcal{O}_l(\sigma_1) = \mathcal{O}_l(\sigma_2) \Longrightarrow (\sigma_1 \in F_M \iff \sigma_2 \in F_M)
\end{array}
$$

The first property of this definition requires that we have already established safety before considering security. The second property is the unwinding condition restricted to invariant I. The third property says that the finality of a state is observable to l (again under invariant I); it is needed to close a potential termination-related security leak.

Low-Level Security For low-level security, the definition is more complex. As discussed in Section 2, we first must define whole-execution behaviors with respect to a monotonic observation function.

Definition 5 (Behavioral State)**.** Given a machine M and a partial order $\preceq$ over the observation type Ω_M, we say that a program state σ is behavioral for principal l, written $\flat M^l(\sigma)$, if all executions starting from σ respect the partial order; i.e., the following monotonicity property holds:

$$\forall \sigma' \ . \ \sigma \mapsto^* \sigma' \Longrightarrow \mathcal{O}_l(\sigma) \preceq \mathcal{O}_l(\sigma')$$

Definition 6 (Behavioral Machine)**.** We say that a machine M is behavioral for principal l, written $\flat M^l$, when the machine has the following components:

- a partial order $\preceq$ over the observation type Ω_M
- a proof that all states of M are behavioral for l

For presentation purposes, we will give a somewhat informal definition of whole-execution behaviors here. The formal Coq definition involves a combination of inductive and coinductive types (to handle behaviors of both terminating and non-terminating executions). Note that our definition is quite similar to the one used in CompCert [14], except that we use state observations as the basic building block, while CompCert uses *traces*, which are input/output events labeled on transitions.

Definition 7 (Whole-Execution Behaviors)**.** Given a machine M with a partial order defined over Ω_M, and a state σ that is behavioral for principal l, we write $\mathcal{B}_{M;l}(\sigma)$ to represent the (potentially infinite) set of whole-execution behaviors that can arise from some execution of M starting from σ. The behaviors (elements of this set) can be one of four kinds: *fault*, *termination*, *silent divergence*, and *reactive divergence*. In the following, variable o ranges over observations and os ranges over infinite streams of observations:

1. $\texttt{Fault}(o) \in \mathcal{B}_{M;l}(\sigma)$ indicates that there is an execution $\sigma \mapsto^* \sigma'$ where σ' is not a final state, σ' cannot take a step to any state, and $o = \mathcal{O}_l(\sigma')$.
2. $\texttt{Term}(o) \in \mathcal{B}_{M;l}(\sigma)$ indicates that there is an execution $\sigma \mapsto^* \sigma'$ where σ' is a final state and $o = \mathcal{O}_l(\sigma')$.
3. $\texttt{Silent}(o) \in \mathcal{B}_{M;l}(\sigma)$ indicates that there is an execution $\sigma \mapsto^* \sigma'$ where $o = \mathcal{O}_l(\sigma')$ and it is possible for M to take infinitely many steps from σ' without any change in observation (i.e., no new observations are ever produced).
4. $\texttt{React}(os) \in \mathcal{B}_{M;l}(\sigma)$ indicates that there is an infinite execution starting from σ that "produces" each of the infinitely-many observations of os in order. An observation o is "produced" in an execution when there exists some single step in the execution $\sigma' \mapsto \sigma''$ with $o = \mathcal{O}_l(\sigma'')$ and $\mathcal{O}_l(\sigma') \neq \mathcal{O}_l(\sigma'')$.

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 ρ, and will revisit the relation later in this section.

Definition 8 (Low-Level Security)**.** Given a machine M that is behavioral for principal l, we say that M is behaviorally secure for l under state relation ρ, written ∇M_l^ρ, just when:

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

Simulation We next formalize our definition of simulation. It is the same as the 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 9 (Simulation, Attempt 1)**.** 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. Note that, for presentation purposes, we omit details here regarding the well-known "infinite stuttering" problem for simulations (described, for example, in [14]). 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 σ and s are related, but also that σ satisfies an appropriate invariant.

It is easy to show that simulations are reflexive and transitive.

Lemma 1 (Reflexivity)**.**

$$\forall M\ .\ M \sqsubseteq_{Id} M$$

Lemma 2 (Transitivity)**.**

$$\forall M_1, M_2, M_3, R_1, R_2\ .\ M_1 \sqsubseteq_{R_1;l} M_2 \wedge M_2 \sqsubseteq_{R_2;l} M_3 \implies M_1 \sqsubseteq_{R_1 \circ R_2;l} M_3$$

Furthermore, we can easily show that our high-level security definition above is a particularly strong bisimulation for safe machines, where a single step is always simulated by a single step:

Definition 10 (Bisimulation)**.** Given principal l, we say that M bisimulates m using R, written $M \equiv_{R;l} m$ just when $M \sqsubseteq_{R;l} m$ and $m \sqsubseteq_{R^{-1};l} M$.

Definition 11 (Invariant-Aware Indistinguishability)**.**

$$\Theta_l^I(\sigma_1, \sigma_2) \triangleq \sigma_1 \in I \wedge \sigma_2 \in I \wedge \mathcal{O}_l(\sigma_1) = \mathcal{O}_l(\sigma_2)$$

Lemma 3 (High-Level Security Bisimulation)**.**

$$\forall M, I, l\ .\ \Delta M_l^I \implies M \equiv_{\Theta_l^I;l} M$$

End-to-End Security We are almost ready to formalize the end-to-end security guarantee. First, we will need to establish some lemmas regarding behaviors and how they are preserved across simulations. For example, if we have a simulation from M to m, we ideally would like to prove that the possible behaviors of M from some state σ are a subset of the behaviors of m from a related state s. There is one significant technical detail that needs to be addressed: behaviors are defined in terms of observations, and the types of observations of two different machines may be different. Hence we cannot compare behavior sets directly using standard subset or set equality. We will therefore introduce a second relation into our simulations, relating observations of one machine to observations of the other.

Definition 12 (Simulation, Attempt 2)**.** Given two machines M, m, a principal l, a relation R between states of M and states of m, and a relation $\bar{R}$ between observations of M and observations of m, we say that there is a simulation from M to m using R and $\bar{R}$, written $M \sqsubseteq_{R;\bar{R};l} m$, 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.) $\forall \sigma \in \Sigma_M,\ s \in \Sigma_m\ .$
$$R(\sigma, s) \implies \bar{R}(\mathcal{O}_{M;l}(\sigma), \mathcal{O}_{m;l}(s))$$

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

The first two properties are the same as the previous definition, while the previous indistinguishability preservation property has now been split into two new properties. It turns out that we can simplify this definition significantly. The fourth property above actually just states that $\bar{R}$ always relates equal inputs to equal outputs; i.e., $\bar{R}$ is functional. This leads us to our final, simpler definition for simulation:

Definition 13 (Simulation)**.** Given two machines M, m, a principal l, a relation R between states of M and states of m, and a function $\bar{R}$ from observations of M to observations of m, we say that there is a simulation from M to m using

R and $\bar{R}$, written $M \sqsubseteq_{R;\bar{R};l} m$, 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.) $\forall \sigma \in \Sigma_M,\ s \in \Sigma_m$.
$$R(\sigma, s) \implies \bar{R}(\mathcal{O}_{M;l}(\sigma)) = \mathcal{O}_{m;l}(s)$$

Definition 14 (Bisimulation)**.** Given two machines M, m, a principal l, a relation R between states of M and states of m, and an *invertible* function $\bar{R}$ from observations of M to observations of m, we say that there is a bisimulation from M to m using R and $\bar{R}$, written $M \equiv_{R;\bar{R};l} m$, when $M \sqsubseteq_{R;\bar{R};l} m$ and $m \sqsubseteq_{R^{-1};\bar{R}^{-1};l} M$.

In the following, we will overload the $\bar{R}$ function to apply to behaviors in the obvious way (e.g., $\bar{R}(\mathtt{Term}(o)) = \mathtt{Term}(\bar{R}(o))$). Given a simulation $M \sqsubseteq_{R;\bar{R};l} m$, with both M and m behavioral for l, we define subset and equality relations between sets of behaviors by applying $\bar{R}$ to every element of the first set:

Definition 15 (Behavior Subset)**.**
$$\mathcal{B}_{M;l}(\sigma) \sqsubseteq_{\bar{R}} \mathcal{B}_{m;l}(s) \triangleq \forall b \ .\ b \in \mathcal{B}_{M;l}(\sigma) \implies \bar{R}(b) \in \mathcal{B}_{m;l}(s)$$

Definition 16 (Behavior Equality)**.**
$$\mathcal{B}_{M;l}(\sigma) \equiv_{\bar{R}} \mathcal{B}_{m;l}(s) \triangleq \forall b \ .\ b \in \mathcal{B}_{M;l}(\sigma) \iff \bar{R}(b) \in \mathcal{B}_{m;l}(s)$$

Note that this equality relation is asymmetric since $\bar{R}$ can only apply to behaviors in the first set, and $\bar{R}$ is not guaranteed to have an inverse.

We now state the relevant lemmas relating behaviors and simulations, omitting proofs here. For the most part, the proofs are reasonably straightforward, requiring some standard case analyses, as well as applications of induction and coinduction hypotheses.

Lemma 4 (Behavior Exists)**.**
$$\forall M, l, \sigma \ .\ \flat M^l(\sigma) \implies \mathcal{B}_{M;l}(\sigma) \neq \emptyset$$

Lemma 5 (Behavior Determinism)**.**
$$\forall M, l, \sigma \ .\ \flat M^l(\sigma) \wedge \downarrow M \implies |\mathcal{B}_{M;l}(\sigma)| = 1$$

Lemma 6 (Simulation Implies Behavior Subset)**.**
$$\forall M, m, I, R, \bar{R}, l, \sigma, s \ .\ \flat M^l(\sigma) \wedge \flat m^l(s) \wedge \Box M^I \wedge M \sqsubseteq_{R;\bar{R};l} m \wedge \sigma \in I \wedge R(\sigma, s) \implies \mathcal{B}_{M;l}(\sigma) \sqsubseteq_{\bar{R}} \mathcal{B}_{m;l}(s)$$

Lemma 7 (Bisimulation Implies Behavior Equality)**.**
$$\forall M, m, R, \bar{R}, l, \sigma, s \ .\ \flat M^l(\sigma) \wedge \flat m^l(s) \wedge M \equiv_{R;\bar{R};l} m \wedge R(\sigma, s) \implies \mathcal{B}_{M;l}(\sigma) \equiv_{\bar{R}} \mathcal{B}_{m;l}(s)$$

Finally, we state and prove the end-to-end security theorem. We assume there is a specification machine M that is secure for principal l under invariant I, a deterministic implementation machine m that is behavioral for l, and a simulation from M to m. The theorem then says that if we take any two indistinguishable states σ_1, σ_2 of M (which satisfy I), and relate them down to two states s_1, s_2 of m, then the whole-execution behaviors of s_1 and s_2 are equal. Note that we require the implementation to be deterministic — this is a reasonable requirement since the implementation is supposed to represent the real machine execution, and real machines are deterministic. Crucially, the specification may still be nondeterministic.

Since the statement of this theorem is a bit unwieldy, we will reformulate it below to directly express that high-level security implies low-level security.

Theorem 1 (End-to-End Security)**.**
$$\forall M, m, I, R, \bar{R}, l, \sigma_1, \sigma_2, s_1, s_2 \ .\ \Delta M_l^I \wedge \flat m^l \wedge \downarrow m \wedge M \sqsubseteq_{R;\bar{R};l} m \wedge \sigma_1 \in I \wedge \sigma_2 \in I \wedge \mathcal{O}_{M;l}(\sigma_1) = \mathcal{O}_{M;l}(\sigma_2) \wedge R(\sigma_1, s_1) \wedge R(\sigma_2, s_2) \implies \mathcal{B}_{m;l}(s_1) = \mathcal{B}_{m;l}(s_2)$$

Proof. We prove this theorem by defining a new machine N in 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. More formally, we define N to have the following components:

- program state Σ_M
- initial states I_M
- final states F_M
- transition relation T_M
- observation type Ω_m
- observation function $\mathcal{O}_{N;l}(\sigma) \triangleq \bar{R}(\mathcal{O}_{M;l}(\sigma))$

First, we establish the simulation $M \sqsubseteq_{Id;\bar{R};l} N$. Referring to Definition 13, the first two properties hold trivially since N has the same transition relation and final state set as M. The third property reduces to exactly our definition of $\mathcal{O}_{N;l}(-)$ given above.

Next, we establish the simulation $N \sqsubseteq_{R;Id;l} m$. The first two properties of Definition 13 are exactly the same as the first two properties of the provided simulation

$M \sqsubseteq_{R;\bar{R};l} m$, and thus they hold. For the third property, assuming we know $R(\sigma, s)$, we have $Id(\mathcal{O}_{N;l}(\sigma)) = \mathcal{O}_{N;l}(\sigma) = \bar{R}(\mathcal{O}_{M;l}(\sigma)) = \mathcal{O}_{m;l}(s)$, where the final equality comes from the third property of the provided simulation.

The next step is to relate the behaviors of N with those of m. In order to do this, we first must show that N has well-defined behaviors for executions starting from σ_1 or σ_2. In other words, we must prove $\flat N^l(\sigma_1)$ and $\flat N^l(\sigma_2)$. We will focus on the proof for σ_1; the other proof is analogous. We use the same partial order as provided by the assumption $\flat m^l$. Consider any execution $\sigma_1 \mapsto^* \sigma_1'$ in N. Since $R(\sigma_1, s_1)$, we can use the simulation $N \sqsubseteq_{R;Id;l} m$ established above, yielding an execution $s_1 \mapsto^* s_1'$, for some s_1' (technically, the simulation property only applies to single steps in the higher machine; however, it can easily be extended to multiple steps through induction on the step relation). Additionally, we have $R(\sigma_1, s_1)$ and $R(\sigma_1', s_1')$, implying by the third property of simulation that $\mathcal{O}_{N;l}(\sigma_1) = \mathcal{O}_{m;l}(s_1)$ and $\mathcal{O}_{N;l}(\sigma_1') = \mathcal{O}_{m;l}(s_1')$. Since m is behavioral, we also have $\mathcal{O}_{m;l}(s_1) \preceq \mathcal{O}_{m;l}(s_1')$. Hence we conclude $\mathcal{O}_{N;l}(\sigma_1) \preceq \mathcal{O}_{N;l}(\sigma_1')$, as desired.

We now know that $\flat N^l(\sigma_1)$ and $\flat N^l(\sigma_2)$. Notice that when $\bar{R}$ is Id, our definitions of behavior subset and equality (Definitions 15 and 16) reduce to standard subset and set equality. Therefore, applying Lemma 6 to the established simulation $N \sqsubseteq_{R;Id;l} m$ tells us that $\mathcal{B}_{N;l}(\sigma_1) \subseteq \mathcal{B}_{m;l}(s_1)$ and $\mathcal{B}_{N;l}(\sigma_2) \subseteq \mathcal{B}_{m;l}(s_2)$ (note that the safety precondition of Lemma 6 holds because M and N have the same state type and transition relation). Furthermore, since m is deterministic, Lemma 5 gives us $|\mathcal{B}_{m;l}(s_1)| = |\mathcal{B}_{m;l}(s_2)| = 1$. Since Lemma 4 guarantees that neither $\mathcal{B}_{N;l}(\sigma_1)$ nor $\mathcal{B}_{N;l}(\sigma_2)$ is empty, we conclude that $\mathcal{B}_{N;l}(\sigma_1) = \mathcal{B}_{m;l}(s_1)$ and $\mathcal{B}_{N;l}(\sigma_2) = \mathcal{B}_{m;l}(s_2)$.

To complete the proof, we now just need to show that $\mathcal{B}_{N;l}(\sigma_1) = \mathcal{B}_{N;l}(\sigma_2)$. Recall Lemma 3, which expresses a security proof as a bisimulation. Extending that lemma with an observation relation of Id, we have the bisimulation $M \equiv_{\Theta_l^I;Id;l} M$. We would like to apply Lemma 7, but we first need to convert this bisimulation into one on N, since M is not behavioral. Since M and N share program state type, final states, and transition relation, it is not difficult to see that the first two required properties of the simulation $N \sqsubseteq_{\Theta_l^I;Id;l} N$ hold. If we can establish the third property, then we will obtain the desired bisimulation $N \equiv_{\Theta_l^I;Id;l} N$ since Id is obviously invertible and Θ_l^I is symmetric. The third property requires us to prove that $\Theta_l^I(\sigma_1, \sigma_2) \implies \mathcal{O}_{N;l}(\sigma_1) = \mathcal{O}_{N;l}(\sigma_2)$. By definition, $\Theta_l^I(\sigma_1, \sigma_2)$ implies that $\mathcal{O}_{M;l}(\sigma_1) = \mathcal{O}_{M;l}(\sigma_2)$. Notice that $\mathcal{O}_{N;l}(\sigma_1) = \mathcal{O}_{N;l}(\sigma_2)$ following from this fact is exactly the indistinguishability preservation property we discussed earlier. Indeed, utilizing the third property of the established simulation $M \sqsubseteq_{Id;\bar{R};l} N$, we have $\mathcal{O}_{N;l}(\sigma_1) = \bar{R}(\mathcal{O}_{M;l}(\sigma_1)) = \bar{R}(\mathcal{O}_{M;l}(\sigma_2)) = \mathcal{O}_{N;l}(\sigma_2)$.

Finally, we instantiate Lemma 7 with both machines being N. Notice that we technically need to establish the precondition of $\Theta_l^I(\sigma_1, \sigma_2)$ before we can apply the lemma. By definition, this reduces to $\sigma_1 \in I \wedge \sigma_2 \in I \wedge \mathcal{O}_{N;l}(\sigma_1) = \mathcal{O}_{N;l}(\sigma_2)$. The first two facts hold by assumption, and the third one follows from $\mathcal{O}_{M;l}(\sigma_1) = \mathcal{O}_{M;l}(\sigma_2)$ by exactly the same reasoning we just used above. Now, applying Lemma 7 yields the conclusion $\mathcal{B}_{N;l}(\sigma_1) \equiv_{Id} \mathcal{B}_{N;l}(\sigma_2)$. As mentioned earlier, behavior equality reduces to standard set equality when $\bar{R}$ is Id, and so we get the desired $\mathcal{B}_{N;l}(\sigma_1) = \mathcal{B}_{N;l}(\sigma_2)$, completing the proof. $\square$

To reformulate the theorem, we wrap up many of the preconditions into a low-level indistinguishability relation. The relation is parameterized by a high-level machine, a principal, a high-level invariant, and a simulation relation. It says that two low-level states are indistinguishable if we can find two indistinguishable high-level states that are related via the simulation relation.

Definition 17 (Low-Level Indistinguishability)**.**

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

End-to-end security is now reformulated to explicitly say that high-level security implies low-level security.

Corollary 1 (End-to-End Security, Reformulated)**.**

$$\begin{aligned}&\forall M, m, I, R, \bar{R}, l \,. \\ &\quad \flat m^l \wedge \downarrow m \wedge M \sqsubseteq_{R;\bar{R};l} m \\ &\quad \implies \left(\Delta M_l^I \implies \nabla m_l^{\phi(M,l,I,R)}\right)\end{aligned}$$

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 [8]. 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 [8]. 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 Σ_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, including two special primitives called `load` and `store`
- 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

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 [13] 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 primitive specifications are the only part of the language semantics that utilize or alter the abstract data component of program state.

The C variant, called ClightX(L), is a direct extension of CompCert's Clight language [2] (which is a slightly-simplified version of C). Like LAsm(L), the semantics is 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 [8], which is an extension of CompCertthat supports per-function compilation.

Each layer L induces a machine M_L of the kind described in Section 3. The state type and initial/final states of M_L come directly from L. The transition relation of type $\mathcal{P}(\Sigma_L \times \Sigma_L)$ is the operational semantics of LAsm(L). The machine's observation function will be discussed later, as it is an extension that we implemented over the existing mCertiKOS specifically for the security proof.

Load/Store Primitives Before continuing, there is one somewhat technical detail regarding the LAsm(L) semantics that requires explanation. While most layer primitives are called in LAsm(L) using the `call` syntax, the special `load` and `store` primitives work differently. Whenever an assembly command dereferences an address, the LAsm(L) semantics consults the load/store primitives to decide how the dereference is actually resolved. This allows TSysCall to interpret addresses as virtual, while MBoot interprets them as physical. As an example, consider the following snippet of assembly code, taken from the implementation of the page fault handler TSysCall primitive:

```
call trap_get
movl %eax, 0(%esp)
call ptfault_resv
```

The layer below TSysCall is called TDispatch, and thus this code is written in the language LAsm(TDispatch). The first and third lines call primitives of TDispatch atomically. The second line ostensibly writes the value of %eax into the memory location pointed to by %esp. The actual semantics of this line, however, will call TDispatch's `store` primitive with the value of %eax and the address in %esp as parameters. This primitive will translate the destination address from virtual to physical by walking through the page tables of the currently-executing process.

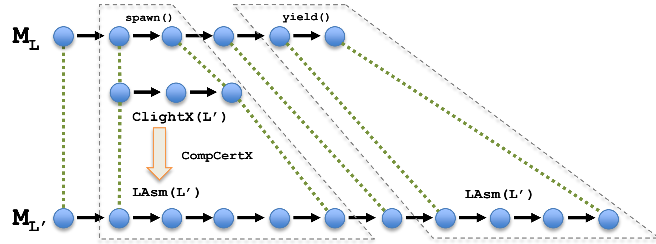

Figure 3. Simulation between adjacent layers. Layer L contains primitives `spawn` and `yield`, with the former implemented in ClightX(L') and the latter implemented in LAsm(L').

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 appropriatelyto get an end-to-end simulation from the M_L execution to the $M_{L'}$ execution.

As a general convention, the simulation relation between consecutive machines only represents an abstraction of some concrete memory into abstract data. In other words, some portion of memory in the lower-level machine is related to some portion of abstract data in the higher-level machine. In this way, as we move up the layers, we have some monotonic notion of abstraction from concrete memory to abstract data. For all intents and purposes, concrete memory has been fully abstracted away at the TSysCall level — user processes have no mechanism for interacting with the memory directly.

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, this indicates that the field is a finite map from process ID to some data type.

- $\texttt{out}_i$ — The output buffer for process i, represented as a list of 32-bit integers.
- `ikern` — A global boolean flag stating whether the machine is currently in kernel mode or user mode.
- `HP` — A global, flat view of the user-space memory heap(physical addresses between 2^{30} and 3×2^{30}). 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 of this map corresponds to the 4096-byte page starting from physical address $4096n$.
- $\texttt{pgmap}_i$ — 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.
- $\texttt{container}_i$ — 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(or 0 for root process 0).
  - `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.
- $\texttt{ctxt}_i$ — 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.
- `rdyQ` — An ordered list of process IDs that are ready to be scheduled.

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 page table (two pages may need to be allocated because the machine uses a two-level page table structure). 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). This means that the process will end up page faulting infinitely, but it will not interfere with the execution of other processes.
- *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 gives processes the power 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 to the next process in the ready queue. 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. Note that output buffers exist in all layers' abstract data, including MBoot. Hence they are never actually implemented in memory; instead, they are assumed to be an accurate representation of some external devices (e.g., monitors).

4.2 Security Overview

We have now provided enough background on mCertiKOS to begin discussing the security verification. We consider each process ID to be a distinct principal or security domain. The high-level security policy expresses which portions of TSysCall state are observable to which principals. The security verification then guarantees complete isolation between all principals: no process's observable state can ever be influenced by the execution of another process.

High-Level Semantics As explained in Sections 2 and 3, 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 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 σ, we say that σ 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.

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 may 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 gives a visual representation of how the semantics of TSysCall-local is defined. The solid arrows are

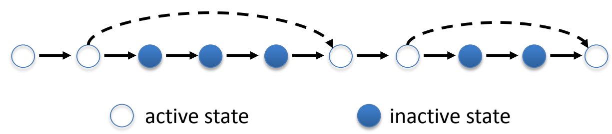

Figure 4. The TSysCall-local semantics, defined by taking big steps over the inactive parts of the TSysCall semantics.

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.

Given all of this discussion, 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, which maps each principal and state to an observation. For a given process ID l, the state observation of σ is defined as follows:

- *Registers* — All registers are observable if σ is active. No registers are observable if σ is inactive.
- *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 (max quota minus usage) 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 of observation exploits the generality of our framework. 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

Now that we have presented all of the relevant setup, we can discuss the details of the mCertiKOS proof effort. The final theorem is the end-to-end security theorem of Section 3, using the simulation from TSysCall-local to MBoot, the

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.

high-level observation function described in Section 4, and a low-level observation function that simply projects the output buffer. Thus 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.

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 as a corollary, since the high-level observation function includes the output buffer as a component.

The largest part 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 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.

Additionally, for a given observer principal l, we assume the invariant that process l has been spawned. Anything occurring before the spawning of l is considered part of the initialization/configuration phase; we are not interested in reasoning about the security of process l before the process even exists in the system.

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 possiblity 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 σ_1 and σ_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 σ_1 and σ_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 that is 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 eventually 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. 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.

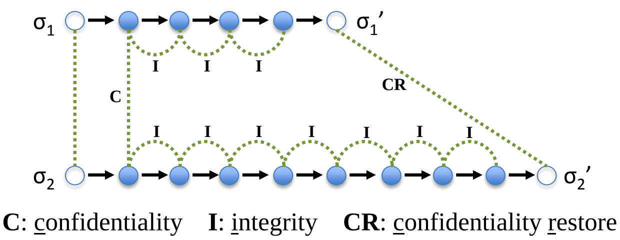

Figure 6. Applying the three lemmas to prove security of TSysCall-local yielding.

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 σ_1 and σ_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 σ_1' and σ_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.

In the mCertiKOS proof, we actually generalize the confidentiality lemma to apply to other primitives besides yield.

- *Generalized Confidentiality* — Two indistinguishable active states always take a step to indistinguishable states.

We frame all of the high-level security proofs for the other primitives as instances of this confidentiality lemma. This means that we derive high-level security of the entire TSysCall-local machine by proving this generalized confidentiality lemma along with integrity and confidentiality restore.

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 σ_1 and σ_2 are equal, then the actual registers of the resulting states σ_1' and σ_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.

Security of Other Primitives We do not need to reason about security of `proc_init` since we assume that initialization occurs appropriately, and no process is ever allowed to call the primitive again after initialization finishes. None of the primitives `get_quota`, `start_user`, or `print` brought up any difficulties for security verification.

6. 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's PER model [24] and Nanevski's Relational Hoare Type Theory [21]. 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 σ to be the set of states related to σ by R.

Our observation function is a generalization of the "conditional labels" presented in [3]. In that work, 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 [15]. 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, e.g. [10, 16, 17]. 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.

Security of OS Kernels The most directly-related work in the area of formal operating system security is the seL4 verified kernel [11, 18, 19, 25]. There are a lot of similarities between the security proof of seL4 and the security proof of mCertiKOS, as both proofs are conducted over a high-level specification and then propagated down to a concrete implementation. There are two main aspects of our methodology that are novel in comparison to seL4. First, the seL4 verification uses a classic notion of observation, similar to external events; the type of observations are the same at the abstract and concrete levels, and the refinement property guarantees that high-level specifications and low-level implementations produce identical observations; our work generalizes observations, allowing them to express different notions of security at different abstraction levels. Second, the seL4 verification only goes down to the level of C implementation; for kernel primitives implemented in assembly, such as context switch, security is verified with respect to an atomic specification that is *assumed* to be correct; the security guarantee we prove about mCertiKOS, on the other hand, applies to the actual assembly execution of the operating system.

Another related work is the information-flow security verification of the PROSPER separation kernel [4]. The goal of that verification effort is 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 works at both the C and assembly levels, using verified compilation to link implementations in different languages.

The Ironclad [9] system aims for full correctness and security verification of an entire system stack. That work 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. One difference is that Ironclad uses Dafny [12], Boogie [1], and Z3 [5] for verification, whereas our approach uses Coq. This means that Ironclad relies heavily on SMT solving, which allows for a large amount of automation in the verification, but does not produce machine-checkable proof evidence like 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.

HiStar HiStar [27] is an operating system that attaches security labels onto all objects. When one object communicates with another, the kernel performs relevant label checks or taints labels appropriately. Certain trusted processes can declassify labels. HiStar has a lot of promise as a security-oriented operating system, but it does not currently come with any formal guarantees. One future goal for mCertiKOS-secure is to implement some of HiStar's features like label checking, and then verify their security. The methodology presented in this work demonstrates an important step towards achieving this goal.

Conclusion 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. We successfully developed a fully-formalized Coq proof that guarantees security of the kernel's assembly execution.

One important area for future work regards inter-process communication. Currently, mCertiKOS-secure does not allow any form of communication between processes. It would be nice to allow some well-specified and disciplined forms of IPC. We have actually already started adding IPC — the version of mCertiKOS-secure submitted in the anonymous supplementary material 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 an observer process with ID less than or equal to k.

References

[1] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, pages 364–387, 2005.

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

[3] D. Costanzo and Z. Shao. A separation logic for enforcing declarative information flow control policies. In *POST*, pages 179–198, 2014.

[4] 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'13, Berlin, Germany, November 4-8, 2013*, pages 223–234, 2013.

[5] 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 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 337–340, 2008.

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

[7] 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.

[8] 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 *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 595–608, 2015.

[9] 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 '14, Broomfield, CO, USA, October 6-8, 2014.*, pages 165–181, 2014.

[10] 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.

[11] 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.

[12] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, pages 348–370, 2010.

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

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

[15] P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *POPL*, pages 158–170, 2005.

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

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

[18] 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.

[19] T. C. Murray, D. Matichuk, M. Brassil, P. Gammie, and G. Klein. Noninterference for operating system kernels. In *Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings*, pages 126–142, 2012.

[20] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *SOSP*, pages 129–142, 1997.

[21] 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.

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

[23] A. Sabelfeld and A. C. Myers. A model for delimited information release. In *ISSS*, pages 174–191, 2003.

[24] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. In *ESOP*, pages 40–58, 1999.

[25] T. Sewell, S. Winwood, P. Gammie, T. C. Murray, J. Andronick, and G. Klein. sel4 enforces integrity. In *Interactive Theorem Proving - Second International Conference, ITP 2011, Berg en Dal, The Netherlands, August 22-25, 2011. Proceedings*, pages 325–340, 2011.

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

[27] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. In *OSDI*, pages 263–278, 2006.