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

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

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

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

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

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

1. Introduction
Information flow control (IFC) [20, 23] 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, 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 our model of 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?

- **Proving Security** — A standard way to prove confidentiality is to formulate the property as noninterference, and prove that some state indistinguishability relation is preserved by each step of execution (this is known as an unwinding condition [6, 7]). However, this proof does not propagate down to the implementation: a primitive specification may atomically preserve indistinguishability, but its non-atomic implementation may temporarily break the relation (and hence the unwinding condition may not hold for all steps of the implementation). Thus we must be careful to formulate the security of a primitive’s implementation as a global behavior-based property over the whole execution of the primitive, rather than a local state-based property over individual steps.

- **Cross-Language Linking** — Even if we verify security for all atomic primitive specifications and propagate the proofs to implementations, there may still be incompatibilities between the proofs for C primitives and those for assembly primitives. For example, a security proof for an assembly primitive might express that some data stored in a particular machine register is not leaked; this property cannot be directly chained with one for a C primitive since the C memory model does not contain machine registers. We therefore must support linking the specifications of primitives implemented in different languages.

In this paper, we present a novel methodology for formally verifying end-to-end security of a system like the one shown in Figure 1. First, security is proved for each high-level specification in a standard way, establishing noninterference by showing that a state indistinguishability relation is invariant across the specification. 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.

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

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

We demonstrate the efficacy of our approach by applying it to the mCertiKOS operating system [8]. We modify mCertiKOS to disable all explicit inter-process communication, and then we prove noninterference between user processes with distinct IDs. mCertiKOS guarantees full functional correctness of system calls (with respect to an idealized x86 machine model derived from CompCert’s model) by chaining simulations across many abstraction layers. We implement our general notion of observation function over the existing simulation framework, and then verify security of the high-level system call specifications. The result of this effort is 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 model of assembly execution of the whole system is secure with respect to this policy.

To summarize, the primary contributions of this work are:

- A novel methodology for end-to-end security verification of software systems written in both C and assembly, and extending across various levels of abstraction.

- An end-to-end security proof, completely formalized in the Coq proof assistant [26], of a simple but nontrivial operating system kernel that executes over an extension of the CompCert x86 machine model. The kernel is non-preemptive, and explicit communication between processes is disabled.

The rest of this paper is organized as follows. Sec. 2 introduces the observation function and shows how to use it for policy specification, security proof, and linking. Sec. 3 formalizes our simulation framework and shows how we prove the end-to-end security theorem. Sec. 4 and 5 describe the security property that we prove over mCertiKOS, highlighting the most interesting aspects of our proofs. Sec. 6 discusses
We use observation functions to express high-level policies when the machine is obvious from context. Assume we have some set $L$ of principals or security domains that we wish to fully isolate from one another, and a state transition machine $M$ describing the single-step operational semantics of execution at a particular level of abstraction. For any type of observations, we define the observation function of $M$ to be a function mapping a principal and program state to an observation. For a principal $l$ and state $\sigma$, we express the state observation notationally as $O_{M,l}(\sigma)$, or just $O_l(\sigma)$ 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):

```c
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 $[x \leftarrow 7; y \leftarrow 5]$ 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:

- $O_{A}(s) \triangleq [a \leftarrow s(a); x \leftarrow s(x); y \leftarrow s(y)]$
- $O_{B}(s) \triangleq [b \leftarrow s(b)]$

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:

$$O_{A}(s) \triangleq [a \leftarrow s(a)\%2; x \leftarrow s(x)\%2; y \leftarrow s(y)\%2]$$

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:

$$O_{A}(s) \triangleq (s(x) + s(y))/2$$

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. Intuitively, if a step of execution always preserves indistinguishability, then the final observation of the step can never be influenced by changing unobservable data in the initial state (i.e., high-security inputs cannot influence low-security outputs).

More formally, for any principal $l$ and state transition machine $M$ with single-step transition semantics $T_M$, we say that $M$ is secure for $l$ if the following property holds for all states $\sigma_1, \sigma_2, \sigma'_1$, and $\sigma'_2$:

$$O_l(\sigma_1) = O_l(\sigma_2) \land (\sigma_1, \sigma'_1) \in T_M \land (\sigma_2, \sigma'_2) \in T_M \implies O_l(\sigma'_1) = O_l(\sigma'_2)$$

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:

```c
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 [18] 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
Also suppose we have a simulation relation \( R \) while \( z \) is unobservable, while the unshaded part is observable.

specializing the unwinding condition above such that states \( \sigma_1 \) and \( \sigma_2 \) are identical:

\[
(\sigma, \sigma'_1) \in T_M \land (\sigma, \sigma'_2) \in T_M \implies \Omega_I(\sigma'_1) = \Omega_I(\sigma'_2)
\]

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 indistinguishability, while the low-level one leaks the secret value of \( x \) into the observable variable \( z \).

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

\[
\Omega_{M,l}(\sigma_1) = \Omega_{M,l}(\sigma_2) \land (\sigma_1, s_1) \in R \land (\sigma_2, s_2) \in R \\
\implies \Omega_{m,l}(s_1) = \Omega_{m,l}(s_2)
\]

3. End-to-End Security Formalization

In this section, we describe the formal proof that some notion of security is preserved across simulation. Most of the technical details are omitted here for clarity of presentation, but can be found in Appendix A.

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:

- a type \( \Sigma_M \) of program state
- a set of initial states \( I_M \) and final states \( F_M \)
- a transition (step) relation \( T_M \) of type \( \mathcal{P}(\Sigma_M \times \Sigma_M) \)
- a type \( \Omega_M \) of observations
- an observation function \( \Omega_{M,l}(\sigma) \) of type \( \mathcal{L} \times \Sigma_M \rightarrow \Omega_M \)

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 \( \sigma \mapsto^n \sigma' \).

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

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

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

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

1. ) \forall \sigma \in I - F_M \exists \sigma'. \sigma \mapsto \sigma' \\
2. ) \forall \sigma, \sigma'. \sigma \in I \land \sigma \mapsto \sigma' \Rightarrow \sigma' \in I \\

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

1. ) safe(\( M, I \)) \\
2. ) \forall \sigma_1, \sigma_2 \in I, \sigma_1' \land \sigma_1 \mapsto \sigma_2 \Rightarrow \sigma_1' \land \sigma_2 \mapsto \sigma_2' \\
3. ) \forall \sigma_1, \sigma_2 \in I . \\
\( O_l(\sigma_1) = O_l(\sigma_2) \Rightarrow (\exists \sigma_1 \in F_M \Rightarrow \sigma_2 \in F_M) \\

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

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

For any machine \( M \) that is behavioral for principal \( l \), we can define the set of whole-execution behaviors that are possible starting from a given state \( \sigma \). We refer to this set as \( B_{M,I}(\sigma) \). The four kinds of behaviors are faulting (getting stuck), termination, silent divergence, and reactive divergence. The definitions can be found in Appendix A; the main point to understand here is that behaviors use the machine’s observation type as a building block. For example, a behavior might say “an execution from \( \sigma \) terminates with final observation \( \sigma' \)”, or “an execution from \( \sigma \) diverges, producing an infinite stream of observations \( os \)”. 

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

\( \forall \sigma_1, \sigma_2 . \rho(\sigma_1, \sigma_2) \Rightarrow B_{m,I}(\sigma_1) = B_{m,I}(\sigma_2) \)

**Simulation** We next formalize our definition of simulation. It is mostly standard, except that we require the simulation relation to preserve indistinguishability.

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

1. ) \forall \sigma, \sigma'. \Sigma_M, s \in \Sigma_m . \\
\sigma \mapsto \sigma' \land R(\sigma, s) \Rightarrow \exists s' \in \Sigma_m . s \mapsto * s' \land R(\sigma', s') \\
2. ) \forall \sigma \in \Sigma_M, s \in \Sigma_m . \\
\sigma \in F_M \land R(\sigma, s) \Rightarrow s \in F_m \\
3. ) \forall \sigma_1, \sigma_2 \in \Sigma_M, s_1, s_2 \in \Sigma_m . \\
\( O_{M,l}(\sigma_1) = O_{M,l}(\sigma_2) \land R(\sigma_1, s_1) \land R(\sigma_2, s_2) \Rightarrow O_{m,l}(s_1) = O_{m,l}(s_2) \)

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.

**End-to-End Security** Our end-to-end security theorem is proved with the help of a few lemmas about behaviors and simulations, described in Appendix A. For example, one lemma basically says that if we have a simulation between behavioral machines \( M \) and \( m \), then the possible behaviors of \( M \) from some state \( \sigma \) are a subset of the behaviors of \( m \) from a related state \( s \). There is one significant technical detail we should mention here regarding these lemmas: behaviors are defined in terms of observations, and the types of observations of two different machines may be different. Hence we technically cannot compare behavior sets directly using standard subset or set equality. For the purpose of presentation, we will actually assume here that all behavioral machines under consideration use the same type for observations. In fact, the mCertIKOS proof is a special case of our framework that obeys this assumption (all behavioral machines use the output buffer observation). Our framework is nevertheless capable of handling changes in observation type by adding a new relation to simulations that relates observations; Appendix A contains the details.

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

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

\( \phi(M, I, I, R) \triangleq \lambda s_1, s_2 . \exists \sigma_1, \sigma_2 \in I . \\
O_{M,l}(\sigma_1) = O_{M,l}(\sigma_2) \land R(\sigma_1, s_1) \land R(\sigma_2, s_2) \)

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

\( \Delta M_l^I \Rightarrow \nabla m_l^I \)

**Proof Sketch.** 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. We prove that \( N \) is behavioral, and apply some lemmas to equate the whole-execution behaviors of \( m \) with those of \( N \). We then formulate the high-level security proof as a bisimulation over \( M \), and use this to derive a bisimulation over \( N \). Finally, we apply a lemma to connect the bisimulation over \( N \) with the whole-execution behaviors of \( N \), completing the proof. The details of this proof and the relevant lemmas can be found in Appendix A.
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 \( \Sigma_L \) of program state, separated into machine registers, concrete memory, and abstract data of type \( D_L \)
- a set of initial states \( I_L \) and final states \( F_L \)
- a set of primitives \( P_L \) implemented by the layer, including two special primitives called \texttt{load} and \texttt{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 \( \text{LAsm}(L') \) or \( \text{ClightX}(L') \) (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 \text{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.

The C variant, called \text{ClightX}(L), is a direct extension of CompCert’s Clight language [2] (which is a slightly-simplified version of C). Like \text{LAsm}(L), the semantics is extended with the ability to call the primitives of \( L \) atomically. \text{ClightX}(L) programs can be compiled to \text{LAsm}(L) in a verified-correct fashion using the CompCertX compiler [8], which is an extension of CompCert.

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

Load/Store Primitives Before continuing, there is one somewhat technical detail regarding the \text{LAsm}(L) semantics that requires explanation. While most layer primitives are called in \text{LAsm}(L) using the \texttt{call} syntax, the special \texttt{load} and \texttt{store} primitives work differently. Whenever an assembly command dereferences an address, the \text{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:

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

The layer below TSysCall is called TDDispatch, and thus this code is written in the language \text{LAsm(TDispatch)}. The first and third lines call primitives of TDDispatch 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 TDDispatch’s \texttt{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.

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 \text{LAsm} with the latter implemented in \text{LAsm(L')}.

![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').](image-url)
**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.

- **out\(_i\)** — The output buffer for process \(i\).
- **HP** — A global, flat view of the user-space memory heap. A page is defined as the 4096-byte sequence starting from a physical address that is divisible by 4096.
- **AT** — A global allocation table, represented as a bitmap indicating which pages in the global heap have been allocated. Element \(n\) of this map corresponds to page \(n\).
- **pgmap\(_i\)** — A representation of the two-level page map for process \(i\). The page map translates a virtual address between 0 and \(2^{32} - 1\) into a physical address.
- **container\(_i\)** — A representation of process \(i\) that maintains information regarding spawned status, children, parents, and resource quota. A container is itself a Coq record containing the following fields:
  - **used** — A boolean indicating whether process \(i\) has been spawned.
  - **parent** — The ID of the parent of process \(i\).
  - **nchildren** — The number of children of process \(i\).
  - **quota** — The maximum number of pages that process \(i\) is allowed to dynamically allocate.
  - **usage** — The current number of pages that process \(i\) has dynamically allocated.
  - **ctxt\(_i\)** — The saved register context of process \(i\), containing the register values that will need to be restored the next time process \(i\) is scheduled.
  - **cid** — The currently-running process ID.

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

- **Initialization** — proc_init sets up the various kernel objects to get everything into a working state. We never attempt to reason about anything that happens prior to initialization; it is assumed that the bootloader will always call proc_init appropriately.
- **Load/Store** — Since paging is enabled at the TSysCall level, the load and store primitives walk the page tables of the currently-running process to translate virtual addresses into physical. If no physical address is found due to no page being mapped, then the faulting virtual address is written into the CR2 control register, the current register context is saved, and the instruction pointer register is updated to point to the entry of the page fault handler primitive.
- **Page Fault** — pgf_handler is called immediately after one of the load/store primitives fails to resolve a virtual address. It reads the faulting virtual address from the CR2 register, allocates one or two new pages as appropriate, increases the current process’s page usage, and plugs the page(s) into the page table. It then restores the register context that was saved when the load/store primitive faulted. If the current process does not have enough available quota to allocate the required pages, then the instruction pointer register is updated to point to the entry of the yield primitive (see below).
- **Get Quota** — get_quota returns the amount of remaining quota for the currently-executing process.
- **Spawn Process** — proc_create attempts to spawn a new child process. It takes a quota as a parameter, specifying the maximum number of pages the child process will be allowed to allocate. This quota allowance is taken from the current process’s available quota.
- **Yield** — sys_yield performs the first step for yielding. It enters kernel mode, disables paging, saves the current registers, and changes the currently-running process ID to the head of the ready queue. It then context switches by restoring the newly-running process’s registers. The newly-restored instruction pointer register is guaranteed (proved as an invariant) to point to the function entry of the start_user primitive.
- **Start User** — start_user performs the simple second step of yielding. It enables paging for the currently-running process and exits kernel mode. The entire functionality of yielding must be split into two primitives 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.
- **Output** — print appends its parameter to the currently-running process’s output buffer.

### 4.2 Security Overview

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

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

To see this, consider any process ID (i.e., principal) \( l \), which we call the “observer process”. For any TSysCall state \( \sigma \), we say that \( \sigma \) is “active” if \( \text{cid}(\sigma) = l \), 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 will not be preserved in this situation.

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

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

Our high-level security property is proved over the TSysCall-local machine, for any choice of observer principal \( l \).

We conclude that registers should be observable to

 inactive state

active state

We easily prove simulation from TSysCall-local to TSysCall, so this strategy fits cleanly into our simulation framework.

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

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

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

5. **Security Verification of mCertiKOS**

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

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

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

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

Figure 5 gives the number of lines of Coq definitions and proof scripts required for the proof effort. The entire effort is broken down into security proofs for primitives, glue code to interface the primitive proofs with the LAsm(L) semantics, definitions and proofs of the framework described in Section 3, and proofs of new state invariants that were established. We will now discuss the most interesting aspects and difficulties of the TSysCall-local security proof.

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

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

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

To better understand the intricacies of this proof, consider the following situation. Suppose we have two states $\sigma_1$ and $\sigma_2$ with equal mappings of virtual addresses to option values (where no value indicates a page fault). Suppose we are writing to some virtual address $v$ in two executions on these states. Consider what happens if there exists some other virtual address $v'$ such that $v$ and $v'$ map to the same physical page in the first execution, but map to different physical pages in the second. It is still possible for $\sigma_1$ and $\sigma_2$ to have identical views of their virtual address space, as long as the two different physical pages in the second execution contain the same values everywhere. However, writing to $v$ will change the observable view of $v'$ in the first execution, but not in the second. Hence, in this situation, it is possible for the store primitive to break indistinguishability.

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

**Security of Process Spawning** The `proc_create` primitive was the only one whose security depended on making a major change to the existing mCertiKOS. When the insecure version of mCertiKOS creates a new child process, it chooses the lowest process ID 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 yield. 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** One interesting aspect of page fault security involves some trickiness with running out of heap space. When allocating a page, mCertiKOS always chooses the first page available. Therefore, the global allocation table $AT$ must be unobservable to prevent an information leak (similar to the information leak via process ID discussed above). 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.
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 active states immediately preceding a yield back to the observer process, and the confidentiality restore lemma establishes indistinguishability of the active states after yielding back to the observer process.

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

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

### 6. Limitations and Future Work

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

- **Completeness Gap** — Our model is certainly not complete for all user-level assembly programs, so it may be possible to violate security on actual hardware by exploiting unmodeled assembly instructions. One example of such an instruction is RDTSC, which reads the time-stamp counter from x86 hardware and stores it into local registers. This time-stamp counter is increased with each clock cycle, so it can be used as a communication channel between processes, leaking information about how much time certain processes have spent executing. We do not model the
we can design a simple model consisting of registers and a

Soundness Gap — In addition to this completeness gap, there is also a potential soundness gap between our machine model and the physical hardware; we must trust that the semantics of all of our modeled assembly instructions are faithful to the actual hardware execution. This is a standard area of trust that arises in any formal verification effort: at some point, we always reach a low-enough level where trust is required, whether this means trusting the operating system that a program is running on, trusting the hardware to meet its published specifications, or trusting the laws of physics that the hardware is presumably obeying. Note that the level of trustworthiness of our machine model is similar to CompCert’s, since we use a moderate extension over CompCert’s model.

Safety Gap — The soundness gap just described requires us to trust that whenever the modeled semantics of an assembly instruction is well-defined, the execution of that instruction on physical hardware will do what the model says. What happens, however, if the modeled semantics gets stuck? The model makes no promises about the actual execution of a stuck semantics; the execution could continue running without issues, but it would no longer be bound by any of our verification. Therefore, even if we closed the completeness and soundness gaps described above to a point of satisfaction, we would still be required to assume that user programs never have undefined semantics in order to apply our verification to the physical execution. This is quite a heavyweight assumption, as user-level code is meant to represent arbitrary and unverified assembly.

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

• interrupt — A trap into the kernel to handle, for example, a privileged instruction or a system call.
• load/store — Instructions that use the kernel’s load/store primitives to access the virtual address space. These may trigger a page fault, which will be handled by the kernel.
• other — Any other user-land instruction, which is assumed to only be able to read/write the values in registers.

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

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

High-Level Policy Specification — As with any formal verification effort, we must trust that the top-level specification of our system actually expresses our intentions for the system, including the security policy expressed as an observation function. Because observation functions can have any type, our notion of security is far more expressive than classical pure noninterference. This does mean, however, that it can potentially be difficult to comprehend the security ramifications of a complex or poorly-constructed observation function. We place onus on the system verifier to make the observation function as clear and concise as possible. This view is shared by a number of previous security frameworks with highly-expressive policy specification, such as the PER model [24] and Relational Hoare Type Theory [21]. In our mcKCertKOS security specification, the virtual address space portion of observation provides a good example of a nontrivial but clear policy specification — hiding physical addresses is, after all, the primary reason to use virtual address spaces.

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

• The high-level security policy must be expressed as isolation between the observation functions of different principals. As mentioned previously, the complete lack of restrictions on the observation function yields a very high level of policy expressiveness. While a systematic exploration of expressiveness remains to be done, we have not encountered any kinds of information flows that are not expressible in terms of an observation function.
• The high-level security property (Definition 3) must be provable over the top-level semantics. In particular, this means that indistinguishability must be preserved on a
step-by-step basis. If it is not preserved by each individual step, then the top-level semantics must be abstracted further. For example, in our mCertiKOS security verification, we found that the TSysCall semantics did not preserve indistinguishability on a step-by-step basis; we therefore abstracted it further into the TSysCall-local semantics that hides the executions of non-observer processes. We are unsure if this requirement for single-step indistinguishability preservation could be problematic for other systems. In our experience, however, repeated abstraction to the point of atomicity is highly desirable, as it yields a clear view of what is really going on in the system.

- Indistinguishability-preserving simulations must be established to connect the various levels of abstraction. While the main simulation property can require significant effort, we have not found the indistinguishability preservation property to be difficult to establish in practice. The property generally feels closer to a sanity check than a significant restriction. Consider, for instance, the example of the swap primitive from Section 2.3. That example failed to preserve security across simulation because the local variable $z$ was being considered observable. A caller of the swap primitive should have no knowledge of $z$, however. Thus this is just a poorly-constructed observation function; a reasonable notion of observation would hide the local variable, and indistinguishability preservation would follow naturally.

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

### 7. Related Work and Conclusions

**Observations and Indistinguishability** Our flexible notion of observation is similarly powerful to purely semantic and relational views of indistinguishability, such as the ones used in Sabelfeld et al.’s PER model [24] and Nanevski et al.’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 $\sigma$ to be the set of states related to $\sigma$ by $R$. We have not systematically explored the precise extent of policy expressiveness in our methodology; this could be an interesting direction for future work.

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

Our approach is also a generalization of Delimited Release [22] 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 $z$ 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 [10, 16, 17]. The one that is most closely related to our setup is Murray et al.’s seL4 security proof [18], where the main security properties are shown to be preserved across refinement. As we mentioned in Section 2, we employ a similar strategy for security preservation in our framework, disallowing high-level specifications from exhibiting domain-visible nondeterminism. Because we use an extremely flexible notion of observation, however, we encounter another difficulty involved in preserving security across simulation; this is resolved with the natural solution of requiring simulation relations to preserve state indistinguishability.

**Comparison with mCertiKOS-base** Our verified secure kernel builds directly over the “base” version of mCertiKOS presented in Gu et al. [8]. In that version, the many layers of mCertiKOS are connected using CompCert-style simulations, and CompCertX is used to integrate C primitives with assembly primitives. However, that version does not have general notions of observations, events, or behaviors. Technically, CompCert expresses external events using traces that appear on the transition functions of operational semantics, and then defines whole-execution behaviors in terms of events; however, mCertiKOS does not make use of these events (the LAsm semantics completely ignores CompCert traces).

Separately from the security verification effort, a large portion of our work was devoted to developing the framework of generalized observations and indistinguishability-preserving simulations described in Sections 2 and 3 (over 2000 lines...
of Coq code, as shown in Figure 5), and integrating these ideas into mCertiKOS. The previous mCertiKOS soundness theorem in Gu et al. [8] only claimed a standard simulation between TSysCall and MBoot. We integrated observation functions into the mCertiKOS layers, modified this soundness theorem to establish an indistinguishability-preserving simulation between TSysCall and MBoot, and then defined whole-execution behaviors and proved an extended soundness theorem guaranteeing that the behaviors of executions at the TSysCall level are identical to those of corresponding executions at the MBoot level.

**Security of seL4** An important work in the area of formal operating system security is the seL4 verified kernel [11, 18, 19, 25]. There are some similarities between the security proof of seL4 and that of mCertiKOS, as both proofs are conducted over a high-level specification and then propagated down to a concrete implementation. Our work, however, has three important novelties over the seL4 work.

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

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

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

**Security of Other OS Kernels** 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.

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

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


A. Appendix

We present the full proof for transferring security across simulation here, even when the type of observations changes across machines. The definitions of machines and high-level security are identical to those presented in Section 3. The definition of low-level security is also the same, but we need to define whole-execution behaviors more formally.

Definition 8 (Behavioral State). Given a machine $M$ and a partial order $\leq$ over the observation type $\Omega_M$, we say that a program state $\sigma$ is behavioral for principal $l$, written $\triangleright M^l(\sigma)$, if all executions starting from $\sigma$ respect the partial order; i.e., the following monotonicity property holds:

$$\forall \sigma'. \sigma \rightarrow^* \sigma' \implies O_l(\sigma) \leq O_l(\sigma')$$

Definition 9 (Behavioral Machine). We say that a machine $M$ is behavioral for principal $l$, written $\triangleright M^l$, when the machine has the following components:

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

We will give a semiformal 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 10 (Whole-Execution Behaviors). Given a machine $M$ with a partial order defined over $\Omega_M$, and a state $\sigma$ that is behavioral for principal $l$, we write $B_{M,l}(\sigma)$ to represent the (potentially infinite) set of whole-execution behaviors that can arise from some execution of $M$ starting from $\sigma$. 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. $\text{Fault}(o) \in B_{M,l}(\sigma)$ indicates that there is an execution $\sigma \rightarrow^* \sigma'$ where $\sigma'$ is not a final state, $\sigma'$ cannot take a step to any state, and $o = O_l(\sigma')$.
2. $\text{Term}(o) \in B_{M,l}(\sigma)$ indicates that there is an execution $\sigma \rightarrow^* \sigma'$ where $\sigma'$ is a final state and $o = O_l(\sigma')$.
3. $\text{Silent}(o) \in B_{M,l}(\sigma)$ indicates that there is an execution $\sigma \rightarrow^* \sigma'$ where $o = O_l(\sigma')$ and there is an infinite execution starting from $\sigma'$ for which all states in that infinite execution have identical observations.
4. $\text{React}(os) \in B_{M,l}(\sigma)$ indicates that there is an infinite execution starting from $\sigma$ 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 $\sigma' \rightarrow \sigma''$ with $o = O_l(\sigma')$ and $O_l(\sigma') \neq O_l(\sigma'')$.

Next we define simulations that can relate machines with different observation types. As a first attempt, we start with the same simulation definition as presented in Section 3, but we introduce a second relation that relates observations of one machine to observations of the other.

Definition 11 (Generalized Simulation, Attempt 1). 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. $\forall \sigma, \sigma' \in \Sigma_M, s \in \Sigma_m$. $\sigma \rightarrow \sigma' \land R(\sigma, s)$ $\implies \exists s' \in \Sigma_m. s \rightarrow^* s' \land R'(\sigma', s')$
2. $\forall \sigma \in \Sigma_M, s \in \Sigma_m$. $\sigma \in F_M \land R(\sigma, s)$ $\implies s \in F_m$
3. $\forall \sigma \in \Sigma_M, s \in \Sigma_m$. $R(\sigma, s) \implies \bar{R}(O_{M,l}(\sigma), O_{m,l}(s))$
4. $\forall \sigma_1, \sigma_2 \in \Sigma_M, s_1, s_2 \in \Sigma_m$. $O_{M,l}(\sigma_1) = O_{M,l}(\sigma_2)$ $\land \bar{R}(O_{M,l}(\sigma_1), O_{m,l}(s_1))$ $\land \bar{R}(O_{M,l}(\sigma_2), O_{m,l}(s_2))$ $\implies O_{m,l}(s_1) = O_{m,l}(s_2)$

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 the actual definition for simulation:

Definition 12 (Generalized Simulation). Given two machines $M, m$, a principal $l$, a relation $R$ between states of $M$ and states of $m$, and a function $f$ from observations of $M$ to observations of $m$, we say that there is a simulation from $M$ to $m$ using $R$ and $f$, written $M \sqsubseteq_{R,f;l} m$, when:

1. $\forall \sigma, \sigma' \in \Sigma_M, s \in \Sigma_m$. $\sigma \rightarrow \sigma' \land R(\sigma, s)$ $\implies \exists s' \in \Sigma_m. s \rightarrow^* s' \land R(\sigma', s')$
2. $\forall \sigma \in \Sigma_M, s \in \Sigma_m$. $\sigma \in F_M \land R(\sigma, s)$ $\implies s \in F_m$
3. $\forall \sigma \in \Sigma_M, s \in \Sigma_m$. $R(\sigma, s) \implies f(O_{M,l}(\sigma)) = O_{m,l}(s)$
We next define a bisimulation, and state a lemma that reformulates high-level security as a bisimulation (the proof is omitted here, but it is completely straightforward).

**Definition 13 (Bisimulation).** Given two machines $M$, $m$, a principal $l$, a relation $R$ between states of $M$ and states of $m$, and an invertible function $f$ from observations of $M$ to observations of $m$, we say that there is a bisimulation from $M$ to $m$ using $R$ and $f$, written $M \equiv_{R,f,l} m$, when $M \subseteq_{R,f,l} m$ and $m \subseteq_{R^{-1},f^{-1},l} M$.

**Definition 14 (Invariant-Aware Indistinguishability).**

$\Theta_l^I(\sigma_1, \sigma_2) \triangleq \sigma_1 \in I \land \sigma_2 \in I \land O_l(\sigma_1) = O_l(\sigma_2)$

**Lemma 1 (High-Level Security Bisimulation).**

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

The next step is to define the method of comparing sets of behaviors between machines that have different observation types. In the following, we will overload the $f$ function to apply to behaviors in the obvious way (e.g., $f(\text{Term}(a)) = \text{Term}(f(a))$). Given a simulation $M \subseteq_{R,f,l} m$, with both $M$ and $m$ behavioral for $l$, we define subset and equality relations between sets of behaviors by applying $f$ to every element of the first set:

**Definition 15 (Behavior Subset).**

$B_{M,l}(\sigma) \subseteq_{f} B_{m,l}(\sigma) \triangleq f(\sigma) \in B_{m,l}(s)$

**Definition 16 (Behavior Equality).**

$B_{M,l}(\sigma) \equiv_{f} B_{m,l}(\sigma) \triangleq f(\sigma) \in B_{m,l}(s)$

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

We now state some important 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. We begin with notations for the standard properties of safety and determinism, and then we state the various lemmas using these notations.

**Definition 17 (Safety).** We say that a machine $M$ is safe under state predicate $I$, written $\text{safe}(M, I)$, when the following progress and preservation properties hold:

1. $\forall \sigma \in I - F_M . \exists \sigma' . \sigma \rightarrow \sigma'$
2. $\sigma', \sigma'' . \sigma \in I \land \sigma \rightarrow \sigma' \implies \sigma' \in I$

**Definition 18 (Determinism).** We say that a machine $M$ is deterministic, written $\text{determ}(M)$, when the following properties hold:

1. $\forall \sigma, \sigma' . \sigma \rightarrow \sigma' \land \sigma \rightarrow \sigma'' \implies \sigma' = \sigma''$
2. $\forall \sigma \in F_M . \exists \sigma' . \sigma \rightarrow \sigma'$

**Lemma 2 (Behavior Exists).**

$\forall M, l, \sigma . \beta M_l^I(\sigma) \implies B_{M,l}(\sigma) \neq \emptyset$

**Lemma 3 (Behavior Determinism).**

$\forall M, l, \sigma . \beta M_l^I(\sigma) \land \text{determ}(M) \implies |B_{M,l}(\sigma)| = 1$

**Lemma 4 (Simulation Implies Behavior Subset).**

$\forall M, m, I, R, f, l, \sigma, s . \quad \beta M_l^I(\sigma) \land \beta m_l^I(s) \land \text{safe}(M, I) \land M \subseteq_{R,f,l} m \land \sigma \in I \land R(s, s) \implies B_{M,l}(\sigma) \subseteq_{f} B_{m,l}(s)$

**Lemma 5 (Bisimulation Implies Behavior Equality).**

$\forall M, m, R, f, l, \sigma, s . \quad \beta M_l^I(\sigma) \land \beta m_l^I(s) \land \text{safe}(M, I) \land M \subseteq_{R,f,l} m \land \sigma \in I \land R(s, s) \implies B_{M,l}(\sigma) \equiv_{f} B_{m,l}(s)$

Finally, we state the end-to-end security theorem. 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 (like we do in Section 3).

**Theorem 2 (Generalized End-to-End Security).**

$\forall M, m, I, R, f, l, \sigma_1, \sigma_2, s_1, s_2 . \quad \Delta M_l^I \land \beta m_l^I(s_1) \land R(\sigma_1, s_1) \land R(\sigma_2, s_2) \land \sigma_1 \in I \land \sigma_2 \in I \land O_{M,l}(\sigma_1) = O_{M,l}(\sigma_2) \implies B_{m,l}(s_1) = B_{m,l}(s_2)$

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 19 (Low-Level Indistinguishability).**

$\phi(M, l, I, R) \triangleq \lambda s_1, s_2 . \exists \sigma_1, \sigma_2 \in I . \quad O_{M,l}(\sigma_1) = O_{M,l}(\sigma_2) \land R(\sigma_1, s_1) \land R(\sigma_2, s_2)$

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

**Corollary 1 (Generalized End-to-End Security, Final).**

$\forall M, m, I, R, f, l . \quad \beta m_l^I \land \phi(m_l^I) \land M \subseteq_{R,f,l} m \implies (\Delta M_l^I \implies \nabla m_l^I \phi(M,l,I,R))$