**Abstract**

# Formal End-to-End Verification of Information-Flow Security for Complex Systems

David Costanzo

2016

Protecting the confidentiality of information manipulated by a computing system is one of the most important challenges facing today's cybersecurity community. Many complex systems, such as operating systems, hypervisors, web browsers, and distributed systems, require a user to trust that private information is properly isolated from other users. Real-world systems are full of bugs, however, so this assumption of trust is not reasonable.

The goal of this dissertation is to apply formal methods to complex security-sensitive systems, in such a way that we can guarantee to users that these systems really are trustworthy. Unfortunately, there are numerous prohibitive challenges standing in the way of achieving this goal.

One challenge is how to specify the desired security policy of a complex system. In the real world, pure noninterference is too strong to be useful. It is crucial to support more lenient security policies that allow for certain well-specified information flows between users, such as explicit declassifications. Furthermore, the specified policy must be comprehensible to users at a high level of abstraction, but also must apply to the low-level system implementation.

A second challenge is that real-world systems are usually written in low-level languages like C and assembly, but these languages are traditionally difficult to reason about. Additionally, even if we successfully verify individual C and assembly functions, how do we go about linking them together? The obvious answer is to do the linking after the C code gets compiled into assembly, but this requires trusting that

the compiler did not accidentally or maliciously introduce security bugs. This is a very difficult problem, especially considering that a compiler may fail to preserve security even when it correctly preserves functional behavior.

A third challenge is how to actually go about conducting a security proof over low-level code. Traditional security type systems do not work well since they require a strongly-typed language, so how can a security violation be detected in untyped C or assembly code? In fact, it is actually common for code to temporarily violate a security policy, perhaps for performance reasons, but then to not actually perform any observable behavior influenced by the violation; how can we reason that this kind of code is acceptably secure? Finally, how do we conduct the proof in a unified way that allows us to link everything together into a system-wide guarantee?

In this dissertation, we make two major contributions that achieve our goal by overcoming all of these challenges. The first contribution is the development of a novel methodology allowing us to formally specify, prove, and propagate information-flow security policies using a single unifying mechanism, called the "observation function". A policy is specified in terms of an expressive generalization of classical noninterference, proved using a general method that subsumes both security-label proofs and information-hiding proofs, and propagated across layers of abstraction using a special kind of simulation that is guaranteed to preserve security.

To demonstrate the effectiveness of our new methodology, our second major contribution is an actual end-to-end security proof, fully formalized and machine-checked in the Coq proof assistant, of a nontrivial operating system kernel. Our artifact is the first ever guaranteed-secure kernel involving both C and assembly code, including compilation from the C code into assembly. Our final result guarantees the following notion of isolation: as long as direct inter-process communication is not used, user processes executing over the kernel cannot influence each others' executions in any way. During the verification effort, we successfully discovered and fixed some inter-

esting security holes in the kernel, such as one that exploits child process IDs as a side channel for communication.

We also demonstrate the generality and extensibility of our methodology by extending the kernel with a virtualized time feature allowing user processes to time their own executions. With a relatively minor amount of effort, we successfully prove that this new feature obeys our isolation policy, guaranteeing that user processes cannot exploit virtualized time as an information channel.

# Formal End-to-End Verification of Information-Flow Security for Complex Systems

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
David Costanzo

Dissertation Director: Zhong Shao

December 2016

# Contents

# List of Figures

# Acknowledgements

I owe thanks to many people for supporting me throughout my particularly lengthy graduate school endeavor. First and foremost, my advisor Zhong Shao, who has worked diligently to keep me on the graduating track. Whenever I became discouraged or disillusioned with my work, he would always have fresh ideas on how to make it more exciting. He has a remarkable ability to visualize the ideal destination of where research should be heading; this was extremely helpful whenever I found myself bogged down in gory technical details and Coq proofs.

I am thankful to all of my friends throughout grad school, both for being colleagues to discuss research with, and for being companions to relax and play games with. I especially thank my friend and officemate Shu-Chun Weng, who always seemed to have a clean solution every time I ran into difficulty with Coq proofs.

I am also grateful to my loving family: my parents, siblings, nieces, and nephew, who have always been supportive of my decision to pursue graduate school — despite the fact that it requires me being on the opposite side of the country. I look forward to finally joining you all in California!

Finally, I thank all of the colleagues in Yale's FLINT group that I have had the pleasure of working with throughout the years. I am particularly thankful to Ronghui Gu and Newman Wu, who conducted the bulk of the effort in getting an initial verified version of the mCertiKOS kernel up and running. This initial version was a crucial testbed for demonstrating the applicability and usability of my new

security verification framework; without it, I would only be able to hope that my theory works well in practice.

# Chapter 1

# Introduction

## 1.1 Challenges in Security Reasoning

Information-flow security is highly 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. Security issues have become even more of a concern in recent times with the advent of cloud computing and distributed architecture, since many mutually-distrusting users execute over the same physical hardware.

It is extremely difficult to justify that a piece of software protects users' secrets. Virtually any bug could lead to an exploitable security hole, and reasoning about security is still difficult even when software is known to be functionally correct. It is therefore logical to look to formal methods for providing a foundation for complex security reasoning. There have been many diverse attempts in the literature, especially over the last two decades, at formally guaranteeing software security. These attempts include techniques like adding dynamic security monitors to runtime environments (e.g., [4, 22, 53, 56]), statically bounding software with security-aware type systems or logics (e.g., [5, 24, 27, 41]), and reasoning about the implications of various kinds of high-level security policies (e.g., [42, 50]). The state-of-the-art is not entirely

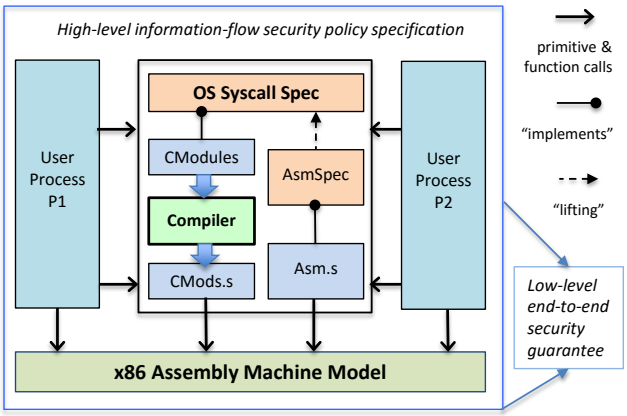



Figure 1.1: An end-to-end software system that consists of both OS modules (in C and assembly) and user processes.

satisfactory, however, as each of these individual security-reasoning methodologies is relatively limited in scope. Our ultimate goal in this dissertation is to demonstrate the possibility of a highly general methodology, that allows for both formally specifying any desired information-flow security policy, as well as formally guaranteeing that the low-level code of a complex system conforms to this high-level policy.

There are all kinds of challenges that we must overcome to achieve such a lofty goal. Consider the example setup of Figure 1.1, where a large operating system kernel consists of many separate functions (e.g., system call primitives) written in either C or assembly. Each primitive has a high-level specification, and there is a compiler that converts all C code into assembly. User processes execute arbitrary assembly code over the kernel, and they may occasionally invoke the kernel system calls. In order to achieve our desired goal, we must be able to clearly and formally specify a high-level security policy over this entire system, and we must be able to provide an *end-to-end* guarantee that all of the code, when linked together and executed over the x86 machine model, conforms to the security policy. We now describe the major challenges involved, contextualized with the help of this example.

**Challenge 1: Policy Specification** How do we specify a clear and precise security policy, describing how information is allowed to flow between various users? If we express the policy in terms of the high-level syscall specifications, then what will this imply for the whole-program assembly execution? We need some way of interpreting and enforcing policies at different levels of abstraction. Furthermore, it is crucial that the high-level policy language is expressive enough to handle more than just pure isolation. In the real world, users often wish to communicate with each other in certain ways; thus we must support policies which allow certain well-specified forms of communication, including controlled declassifications of data from a high security level to a lower one.

**Challenge 2: Reasoning About Low-Level Code** Assuming we can specify a security policy at a low level of abstraction, how should we go about proving that some low-level C or assembly code conforms to the policy? Security-aware type systems like Jif [41] do not work well for untyped languages, while dynamic security monitors incur undesirable execution overhead. Incompleteness is also problematic: a low-level program may execute an action that temporarily violates a security policy, but then the program does not end up producing any observable behaviors influenced by the violation. For example, for performance reasons, a program might read an entire block of a file into memory, despite not actually needing to know any secret data stored within that block. The end-to-end behavior of such a program is secure, but many line-by-line security enforcement mechanisms will deem it insecure.

**Challenge 3: Linking Everything Together** Multiple aspects of Figure 1.1 require linking: C code must be linked with assembly code within the kernel, low-level C and assembly code must be linked with their high-level specifications, and user code must be linked with kernel code. Even if we manage to prove security of each of the individual pieces (kernel C functions, kernel assembly functions, arbitrary user

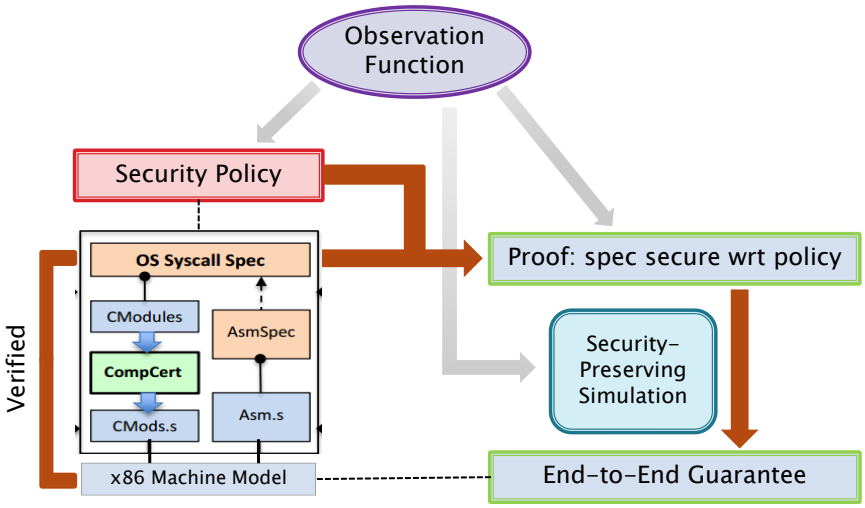



Figure 1.2: Using an observation function to verify end-to-end security.

code, and high-level specifications), how are we going to guarantee end-to-end security when all of these pieces are combined together? Simulations or refinements are generally used to link low-level code with high-level specifications, but it is well known [29, 36] that these may not soundly propagate security properties like noninterference. Furthermore, linking C code with assembly code requires compilation; modern C compilers usually have bugs causing program functionality to not be correctly preserved, and they certainly do not make any attempts to preserve security.

## 1.2 Contributions

In this dissertation, we successfully achieve our goal by showing how all of the challenges mentioned above can be cleanly handled. We make two major contributions: we design a novel methodology that achieves our goal, and we apply the methodology to guarantee security of an actual system.

**Contribution 1: Novel Methodology** The first contribution is the development of a novel and highly-general methodology allowing us to formally specify, verify,

and link security policies. Figure 1.2 gives a broad overview of our design. We first require that functional correctness is verified independently from security: all of the code must be shown to conform to the high-level specifications. Then, for the security verification, we begin by defining an "observation function", which essentially describes each user's view of program state. This observation function automatically induces a high-level security policy. Next, the system's high-level specification is verified to conform to the induced security policy. Finally, this high-level security property is automatically propagated down to a low-level, end-to-end guarantee by exploiting a special kind of simulation that soundly preserves security.

**Contribution 2: Applying the Methodology** We demonstrate the effectiveness of this novel methodology by applying it to real operating system kernel, called mCertiKOS. As described in [21], mCertiKOS is already fully verified to be functionally correct with respect to high-level specifications; thus our security verification effort does not require the first step described in Figure 1.2. Our resulting artifact, called mCertiKOS-secure, is the first ever guaranteed-secure kernel involving both C and assembly code, including compilation from the C code into assembly (which is handled by the CompCert verified compiler [33]). Our final result guarantees the following notion of isolation: as long as direct inter-process communication is not used, user processes executing over the kernel cannot influence each others' executions in any way. During the verification effort, we successfully discovered and fixed some interesting security holes in the kernel, such as one that exploits child process IDs as a side channel. After completing the verification, we tested the extensibility of our methodology by adding a new feature to the kernel providing users with a notion of time. With relatively little effort we were able to prove security of the new feature, guaranteeing the absence of any timing-based information-flow side channel.

**Previous Efforts** While the most important contributions in this dissertation are the two just described, we spent multiple years trying other strategies before arriving at our new methodology. We will devote two chapters to describing these earlier efforts, as they illustrate numerous important concepts that are used as stepping stones toward developing our ultimate contribution. The first of these chapters describes our work published in APLAS [12], in which we argue that systems should enforce a strong notion of locality, guaranteeing that unused resources never affect program behaviors. While we did not have security in mind at the time, it turns out that this strong locality has close ties with composability of secure systems. The second chapter describes our work published in POST [14], in which we present a new program logic that allows one to specify and prove very general security policies over C-like code. The program logic only applies to code written in the C-like language, so it only solves a small portion of the challenges described in Section 1.1.

**Machine-Checked Verification** All of the work throughout this dissertation is fully formalized and verified in the Coq proof assistant [55]. This means that all proofs are machine-checked, and therefore bug-free. The Coq formalizations, including the entirety of the secure mCertiKOS kernel, can be found online at this dissertation's companion website [11].

## 1.3 Principals and Policies

In this section, we informally introduce some basic terminology that will be heavily used throughout the dissertation. We also describe some motivating example security policies that will occasionally be revisited in later chapters.

Assume we have a complex system like the one from Figure 1.1, which is composed of many lines of both C and assembly code. As described previously, we wish to prove the end-to-end property that, when the C code is compiled into assembly and

linked with the existing assembly code, the resulting system executes securely. More specifically, we will prove that a system executes securely with respect to a particular *principal's* point-of-view; this particular principal is called the *observer*. Principals represent actors or users of the system (e.g., processes $P1$ and $P2$ are principals of the system in Figure 1.1), and we will assume they come from some abstract set $\mathcal{P}$.

For each potential observer $p$, we will specify a *security policy* describing precisely how information can flow to $p$. A system is then deemed secure if and only if it obeys all observers' policies. As described in Section 1.1 above, exact formalization of a security policy represents a major challenge in any information-flow system. To support real-world systems, it is crucial that policies allow for certain well-specified flows of information (e.g., declassifications); however, it is in general not obvious how to define an end-to-end security guarantee with respect to such lenient policies. To develop some intuition regarding this challenge, we will now consider some example security policies that are important to support.

**Public Parity** Suppose principal Alice owns some secret value $v$. Alice wishes to release only a single bit of her secret publicly, the parity $v\%2$. Describing this as a somewhat more formal security policy, we say that the observations made by any observer $p$ (excluding Alice) must not be influenced by anything relating to $v$ other than its parity. This statement can be clearly expressed as a noninterference-like property: if we were to hypothetically change Alice's secret from $v$ to any other value $w$ with the same parity as $v$, then observer $p$ must make identical observations over the system executing with secret $w$ as it does over the execution with secret $v$.

Notice that there is some implicit subtlety in the described policy: if an observer can learn the value $v\%2$, then clearly he can also learn, for example, the value $(v+1)\%2$. This implicitly-derived kind of information flow can become difficult to understand for extremely complex policies; we will assume throughout the disserta-

tion that it is the system specifier's burden to write a policy that is simple enough for users of the system to fully comprehend.

**Public Average** Suppose Alice runs a company and stores all employees' salaries in a database. One reasonable security policy is to publicly release only the average of these salaries. That is, there should be no flow of information from the salaries to public observers other than the value of the average salary. As a noninterference statement, this means that if we were to change the values of any subset of salaries in such a way that the overall average remains the same, then a public observer must not see any change in observation. Furthermore, one might reasonably extend the policy to apply to an employee $p$ of the company by allowing $p$ to learn information only about $p$'s own salary, in addition to the average salary. Note that, once again, there is some implicit information embedded in this policy: if, for example, $p$ happens to know that there are only two employees in total at the company, then he can learn the exact value of the other employee's salary just by looking at his own salary along with the average. Thus this policy would only be a reasonable one for security purposes if there were many employees at the company.

**Shared Calendar** As a more intricate example, suppose Alice owns a calendar on which she marks down the details of various events occurring at various time slots. Further suppose that Alice wishes to expose an API for her calendar that allows other principals to schedule a meeting with her at an open time slot. What kind of security policy might Alice wish to enforce over her calendar? An initial guess might be that Alice should not release any information about her calendar; this is incompatible with the desired API, however, since a principal who successfully schedules a meeting with Alice will obviously learn that the scheduled time was available in her calendar. Instead, a more reasonable policy is that a caller of Alice's API can learn which time slots are available/unavailable in the calendar, but cannot learn any information

about the events contained within the unavailable slots.

In terms of noninterference, this policy says that if we were to arbitrarily change the events within Alice's calendar without changing any times at which those events occur, then a caller of Alice's API would not observe any difference. One example implementation that clearly satisfies this security policy is for Alice to always schedule the meeting at the first available time slot.

**Dynamic Label Tainting** One common and important example involves attaching security labels to principals and data within a system, and then dynamically propagating ("tainting") these labels as the system executes. Many existing security frameworks are based upon this scheme (e.g., [4, 25, 31, 60, 61]). Security labels are arranged into a lattice structure where, for example, $L_1 \sqsubseteq L_2$ means that the security level $L_2$ is "at least as secure as" level $L_1$. An element of this lattice is assigned to each principal and each piece of data within the system. The standard security policy is then that information is only allowed to flow up this lattice — that is, an observer $p$ with label $L_p$ can only learn information about data with label less than or equal to $L_p$ in the lattice. As a noninterference statement, this means that changing any data with a label $L \not\sqsubseteq L_p$ will not affect $p$'s observation.

A simple method for enforcing this security policy is to dynamically taint data as it propagates during an execution. For example, if the program $z = x + y$ executes, then the resulting security label of $z$ will be assigned the least upper bound ($\sqcup$) of the labels of $x$ and $y$. In this way, the system can automatically guarantee that the resulting value of $z$ can flow to some principal $p$ (i.e., $L_x \sqcup L_y \sqsubseteq L_p$) if and only if the values of $x$ and $y$ can also flow to $p$ (i.e., $L_x \sqsubseteq L_p$ and $L_y \sqsubseteq L_p$).

**Aside on Declassification Terminology** Traditionally, declassification is a term used in a context like the dynamic label tainting just described, where the security label of a piece of data is explicitly changed from some $L$ to some $L'$ with $L \not\sqsubseteq L'$.

Declassifications may violate the policy that information can only flow up the lattice, and therefore many systems must carefully specify precise conditions under which declassifications are allowed to occur. Throughout this dissertation, we will refer to this concept as an "explicit declassification". One thesis of our work is that it is generally difficult to express a formal end-to-end security property for a system that allows such explicit declassifications. In this work, we instead support what we call "implicit declassifications" — a concept which we will show yields a clean and descriptive end-to-end security property. For example, in the public parity example described above, we say that Alice is "implicitly declassifying" the value $v\%2$ from secret to public. That example is not an explicit declassification because there are no explicit labels attached to the data.

In general, it is not possible to give any kind of formal definition to our notion of implicit declassification. As described above, an implicit declassification of $v\%2$ automatically implies implicit declassification of $(v+1)\%2$. Furthermore, we will see an example later in the dissertation, relating to the mCertiKOS security verification, where a policy seems to describe an implicit declassification, but actually does not allow information to flow from high security to lower security. In other words, there is no clear relationship between implicit declassifications and the allowed information flows. As a result, all uses of the term "declassification" in this dissertation should be interpreted in an informal context; our security policies will formally and precisely specify how information is allowed to flow between principals, but they will not formally specify whether or not declassifications are allowed.

## 1.4 Chapter Organization

The rest of this dissertation is organized as follows. Chapter 2 is an abridged version of our work published in [12], and discusses strong locality. There are some

interesting connections between locality and security, but overall the chapter is fairly orthogonal to the rest of the dissertation. Chapter 3 is an abridged version of our work from [14], and attempts to tackle security verification by using a program logic. The chapter finishes with a discussion regarding the various limitations of using a specific program logic; this leads directly into Chapter 4, where we move away from a specific program logic and informally describe our novel and more general methodology for security verification. Chapter 5 then completely formalizes the methodology and proves the main theorem that security can be automatically propagated across simulations from a high-level specification to a low-level implementation. Chapter 6 then introduces mCertiKOS and its security policy specification, and Chapter 7 continues with many technical details about the security verification effort. Chapter 8 presents our new mCertiKOS feature implementing virtualized time, and shows how we prove the feature secure. Chapter 9 discusses various assumptions and limitations of our methodology, and mentions how these open up opportunities for future work. Finally, Chapter 10 concludes with an in-depth discussion of related work.

# Chapter 2

# Locality and Behavior Preservation

In this chapter, we present our work published in APLAS 2012 [12]. In the context of Separation Logic [9, 28, 46, 58], we define and defend a strong notion of locality, deemed "behavior preservation". Locality concerns the relationship between a program's execution over a small footprint state and its execution over a larger state containing unused resources. Our key idea is to require a behavior-preserving formulation of locality, where the program's behavior is completely unchanged by the extra unused resources.

We describe here (and in the paper) how behavior preservation simplifies numerous metatheoretical difficulties in Separation Logic. This is mostly orthogonal to the rest of the dissertation, as it does not concern security. However, we will discuss some interesting connections to security at the end of the chapter, by relating behavior preservation with classical noninterference.

## 2.1 Local Reasoning and the Frame Rule

Separation Logic is a program logic that allows for formal reasoning about the behavior of heap-manipulating C-like programs. The most important concept is the *separating conjunction* — the assertion $P * Q$ holds on a program state if that state

can be separated into two *disjoint* portions, one satisfying $P$ and the other satisfying $Q$. The following *frame rule* is used to facilitate local reasoning:

$$\frac{\vdash \{P\}\, C\, \{Q\}}{\vdash \{P * R\}\, C\, \{Q * R\}}$$

That is, if a program $C$ is verified to satisfy a pre/post condition pair $(P, Q)$, then we can automatically infer that it also satisfies the pair $(P * R, Q * R)$, where $R$ describes program state that is disjoint from both $P$ and $Q$. Intuitively, this seems to indicate that the unused resources in $R$ do not affect $C$'s behavior. Formally, however, the notion of locality required by the frame rule is actually weaker than this intuition. There are three locality-related properties that together imply soundness of the frame rule, commonly called Safety Monotonicity, the Frame Property, and Termination Monotonicity (the latter is only needed for termination-sensitive reasoning).

To describe these properties more formally, we first introduce some notations for combining disjoint program states. We define states $\sigma$ to be members of an abstract set $\Sigma$. We assume that whenever two states $\sigma_0$ and $\sigma_1$ are "disjoint," written $\sigma_0 \# \sigma_1$, they can be combined to form the larger state $\sigma_0 \bullet \sigma_1$. Intuitively, two states are disjoint when their heaps occupy disjoint areas of memory.

We represent the semantic meaning of a program $C$ by a binary relation $[\![C]\!]$, indicating all possible whole-execution behaviors of $C$. We use the common notational convention $aRb$ for a binary relation $R$ to denote $(a, b) \in R$. Intuitively, $\sigma [\![C]\!] \sigma'$ means that, when executing $C$ on initial state $\sigma$, it is possible to terminate in state $\sigma'$. Note that if $\sigma$ is related by $[\![C]\!]$ to more than one state, this simply means that $C$ is a nondeterministic program. We also define two special behaviors `fault` and `div`:

- $\sigma [\![C]\!]$`fault` means that $C$ can crash or get stuck when executed on $\sigma$
- $\sigma [\![C]\!]$`div` means that $C$ can diverge (execute forever) when executed on $\sigma$

As a notational convention, we use $\tau$ to range over elements of $\Sigma \cup \{\texttt{fault}, \texttt{div}\}$. We require that for any state $\sigma$ and program $C$, there is always at least one $\tau$ such that $\sigma[\![C]\!]\tau$. In other words, every execution must either crash, go on forever, or terminate in some state.

Now we define the properties mentioned above. Following are definitions of Safety Monotonicity, the Frame Property, and Termination Monotonicity, respectively (when not explicitly mentioned, assume all variables are universally quantified):

1.) $\neg\sigma_0[\![C]\!]\texttt{fault} \wedge \sigma_0 \# \sigma_1 \Longrightarrow \neg(\sigma_0 \bullet \sigma_1)[\![C]\!]\texttt{fault}$

2.) $\neg\sigma_0[\![C]\!]\texttt{fault} \wedge (\sigma_0 \bullet \sigma_1)[\![C]\!]\sigma' \Longrightarrow \exists\sigma'_0 \,.\, \sigma' = \sigma'_0 \bullet \sigma_1 \wedge \sigma_0[\![C]\!]\sigma'_0$

3.) $\neg\sigma_0[\![C]\!]\texttt{fault} \wedge \neg\sigma_0[\![C]\!]\texttt{div} \wedge \sigma_0 \# \sigma_1 \Longrightarrow \neg(\sigma_0 \bullet \sigma_1)[\![C]\!]\texttt{div}$

Safety Monotonicity says that any time a program executes safely in a certain state, the same program must also execute safely in any larger state — in other words, unused resources cannot cause a program to crash. The Frame Property says that if a program executes safely on a small state, then any terminating execution of the program on a larger state can be tracked back to some terminating execution on the small state by assuming that the extra added state has no effect and is unchanged. Termination Monotonicity says that if a program executes safely and never diverges on a small state, then it cannot diverge on any larger state.

In standard Separation Logic, these three properties are required to hold for all programs $C$, and the frame rule is then automatically guaranteed to be sound. The properties represent the *minimum* requirement needed to make the frame rule sound — they are as weak as they can possibly be without breaking the logic. They are not defined to correspond with any intuitive notion of locality. As a result, there are two subtleties in the definition that might seem a bit odd. We will now describe these subtleties and the changes we make to get rid of them. Note that we

are not arguing in this section that there is any benefit to changing locality in this way (other than the arguably vacuous benefit of corresponding to our "intuition" of locality) — the benefit will become clear when we discuss how our change simplifies the metatheory in Section 2.4.

The first subtlety is that Termination Monotonicity only applies in one direction. This means that we could have a program $C$ that runs forever on a state $\sigma$, but when we add unused state, we suddenly lose the ability for that infinite execution to occur. We can easily get rid of this subtlety by replacing Termination Monoticity with the following Termination Equivalence property:

$$\neg\sigma_0[\![C]\!]\texttt{fault} \wedge \sigma_0\#\sigma_1 \implies (\sigma_0[\![C]\!]\texttt{div} \iff (\sigma_0 \bullet \sigma_1)[\![C]\!]\texttt{div})$$

The second subtlety is that locality gives us a way of tracking an execution on a large state back to a small one, but it does not allow for the other way around. This means that there can be an execution on a state $\sigma$ that becomes invalid when we add unused state. This subtlety is a little trickier to remedy than the other. If we think of the Frame Property as really being a "Backwards Frame Property," in the sense that it only works in the direction from large state to small state, then we clearly need to require a corresponding Forwards Frame Property. We would like to say that if $C$ takes $\sigma_0$ to $\sigma_0'$ and we add the unused state $\sigma_1$, then $C$ takes $\sigma_0 \bullet \sigma_1$ to $\sigma_0' \bullet \sigma_1$:

$$\sigma_0[\![C]\!]\sigma_0' \wedge \sigma_0\#\sigma_1 \implies (\sigma_0 \bullet \sigma_1)[\![C]\!](\sigma_0' \bullet \sigma_1)$$

Unfortunately, there is no guarantee that $\sigma_0' \bullet \sigma_1$ is defined, as the states might not occupy disjoint areas of memory. In fact, if $C$ causes our initial state to grow, say by allocating memory, then there will always be some $\sigma_1$ that is disjoint from $\sigma_0$ but not from $\sigma_0'$ (e.g., take $\sigma_1$ to be exactly that allocated memory). Therefore, it seems as if we are doomed to lose behavior in such a situation upon adding unused state.

There is, however, a solution worth considering: we could disallow programs from

ever increasing state. In other words, we can require that whenever $C$ takes $\sigma_0$ to $\sigma_0'$, the area of memory occupied by $\sigma_0'$ must be a subset of that occupied by $\sigma_0$. In this way, anything that is disjoint from $\sigma_0$ must also be disjoint from $\sigma_0'$, so we will not lose any behavior. Formally, we express this property as:

$$\sigma_0[\![C]\!]\sigma_0' \Longrightarrow (\forall \sigma_1 \,.\, \sigma_0 \# \sigma_1 \Rightarrow \sigma_0' \# \sigma_1)$$

We can conveniently combine this property with the previous one to express the Forwards Frame Property as the following condition:

$$\sigma_0[\![C]\!]\sigma_0' \wedge \sigma_0 \# \sigma_1 \Longrightarrow \sigma_0' \# \sigma_1 \wedge (\sigma_0 \bullet \sigma_1)[\![C]\!](\sigma_0' \bullet \sigma_1)$$

At first glance, it may seem imprudent to impose this requirement, as it apparently disallows memory allocation. However, it is in fact still possible to model memory allocation — we just have to be a little clever about it. Specifically, we can include a set of memory locations in our state that we designate to be the "free list". When memory is allocated, all allocated cells must be taken from the free list. Because the free list is represented explicitly in program state, any extra unused resources must be disjoint not only from the heap, but also from the free list. Contrast this to standard Separation Logic, in which newly-allocated heap cells are taken from outside the program state. In the next section, we will show that we can add a free list in this way to the model of Separation Logic without requiring a change to any of the inference rules.

We conclude this section with a brief justification of the term "behavior preservation." Given that $C$ runs safely on a state $\sigma_0$, we think of a behavior of $C$ on $\sigma_0$ as a particular execution, which can either diverge or terminate at some state $\sigma_0'$. The Forwards Frame Property tells us that execution on a larger state $\sigma_0 \bullet \sigma_1$ simulates execution on the smaller state $\sigma_0$, while the Backwards (standard) Frame Property says that execution on the smaller state simulates execution on the larger one. Since

standard locality only requires simulation in one direction, it is possible for a program to have fewer valid executions, or behaviors, when executing on $\sigma_0 \bullet \sigma_1$ as opposed to just $\sigma_0$. Our stronger locality disallows this from happening, enforcing a *bisimulation* under which all behaviors (including divergence) are preserved when extra resources are added.

## 2.2 Impact on a Concrete Separation Logic

In this section, we will demonstrate how behavior-preserving locality can be enforced in a standard model of Separation Logic without any negative impact on using the program logic. In the standard Reynolds' Separation Logic model [46], a program state consists of two components: a variable store and a heap. When new memory is allocated, the memory is taken from outside the state and added into the heap. As mentioned in Section 2.1, this notion of memory allocation violates our Forwards Frame Property, so we will instead include an explicit free list inside the program state. Thus a state is now is a triple $(s, h, f)$ consisting of a store, a heap, and a free list, with the heap and free list occupying disjoint areas of memory. Newly-allocated memory always comes from the free list, while deallocated memory goes back into the free list. Since the standard formulation of Separation Logic assumes that memory is infinite and hence that allocation never fails, we similarly require the free list to be infinite. More specifically, we require that there is some location $n$ such that all locations above $n$ are in the free list. Formally, states are defined as follows:

$$\text{Var } V \triangleq \{x, y, z, \ldots\} \qquad \text{Store } S \triangleq V \to \mathbb{Z} \qquad \text{Heap } H \triangleq \mathbb{N} \underset{\text{fin}}{\rightharpoonup} \mathbb{Z}$$

$$\text{Free List } F \triangleq \{N \in \mathbb{P}(\mathbb{N}) \mid \exists n \,.\, \forall k \geq n \,.\, k \in N\}$$

$$\text{State } \Sigma \triangleq \{(s, h, f) \in S \times H \times F \mid \text{dom}(h) \cap f = \emptyset\}$$

As a point of clarification, we are not claiming here that including the free list in the state model is a novel idea. Other systems (e.g., [45]) have made use of a

$$
\begin{array}{rl}
E & ::= E + E' \mid E - E' \mid \ldots \mid -1 \mid 0 \mid 1 \mid \ldots \mid x \mid y \mid \ldots \\
B & ::= E = E' \mid \texttt{false} \mid B \Rightarrow B' \\
P, Q & ::= B \mid \texttt{false} \mid \texttt{emp} \mid E \mapsto E' \mid P \Rightarrow Q \mid P * Q \\
C & ::= \texttt{skip} \mid x := E \mid x := [E] \mid [E] := E' \\
 & \quad \mid x := \texttt{cons}(E_1, \ldots, E_n) \mid \texttt{free}\,(E) \mid C; C' \\
 & \quad \mid \texttt{if}\, B\, \texttt{then}\, C\, \texttt{else}\, C' \mid \texttt{while}\, B\, \texttt{do}\, C
\end{array}
$$

Figure 2.1: Assertion and Program Syntax

very similar idea. The two novel contributions that we will show in this section are: (1) that a state model which includes an explicit free list can provide a behavior-preserving semantics, and (2) that the corresponding program logic can be made to be completely backwards-compatible with standard Separation Logic (meaning that any valid Separation Logic derivation is also a valid derivation in our logic).

We adopt the following standard notations: $\lfloor h \rfloor$ is the domain of the heap $h$; $s[x \mapsto v]$ is the store which is identical to $s$, except that the value of variable $x$ is updated to $v$; $h[l \mapsto v]$ is the heap which is identical to $h$, except that location $l$ is either added to $h$ with value $v$ if it does not exist in $h$, or updated with value $v$ if it does exist; $h \backslash l$ is the heap resulting from removing location $l$ from $h$; $h_0 \# h_1$ is true just when $\lfloor h_0 \rfloor$ and $\lfloor h_1 \rfloor$ do not overlap; $h_0 \bullet h_1$ is equal to the union of $h_0$ and $h_1$ if $h_0 \# h_1$, and is undefined otherwise. We also overload the disjointness ($\#$) operator to work with free lists — e.g., $h \# f$ says that $\lfloor h \rfloor$ and $f$ are disjoint.

Assertion syntax and program syntax are given in Figure 2.1, and are exactly the same as in the standard model for Separation Logic. This syntax includes expressions $E$ and boolean expressions $B$, both of which can be evaluated under a given variable store, without any knowledge of the heap. These valuations are denoted by $[\![E]\!]s$ and $[\![B]\!]s$ for a given store $s$; the former evaluates to an integer, while the latter evaluates to a boolean. Their formal definitions are omitted here, but are straightforward and standard in the literature.

Our satisfaction judgement $(s, h, f) \models P$ for an assertion $P$ is defined by ig-

$$
\begin{array}{rcl}
(s,h) \models B & \iff & [\![B]\!]s = \texttt{true} \\
(s,h) \models \texttt{false} & \iff & \text{never} \\
(s,h) \models \texttt{emp} & \iff & \lfloor h \rfloor = \emptyset \\
(s,h) \models E \mapsto E' & \iff & \lfloor h \rfloor = \{[\![E]\!]s\} \wedge h([\![E]\!]s) = [\![E']\!]s \\
(s,h) \models P \Rightarrow Q & \iff & \text{if } (s,h) \models P, \text{ then } (s,h) \models Q \\
(s,h) \models P * Q & \iff & \left( \begin{array}{c} \exists h_0, h_1 \,.\, h_0 \# h_1 \wedge h_0 \bullet h_1 = h \wedge \\ (s,h_0) \models P \wedge (s,h_1) \models Q \end{array} \right)
\end{array}
$$

Figure 2.2: Satisfaction of Assertions

noring the free list and only considering whether $(s,h)$ satisfies $P$. The definition of $(s,h) \models P$ is identical to that of standard Separation Logic, and is given in Figure 2.2. The most important cases are $E \mapsto E'$ and $P * Q$. $E \mapsto E'$ says that the current heap consists *only* of the memory cell at address $[\![E]\!]s$, and that the cell at that address maps to the value $[\![E']\!]s$. $P * Q$ says that we can separate the current heap into two disjoint subheaps $h_0$ and $h_1$, with $h_0$ satisfying $P$ and $h_1$ satisfying $Q$. We also define the standard syntactic sugars $E \mapsto E_0, \ldots, E_n$ to be $(E \mapsto E_0) * \ldots * (E + n \mapsto E_n)$, and $E \mapsto -$ to be $\exists x. E \mapsto x$ (where $x$ is not free in $E$).

Figure 2.3 defines the small-step operational semantics for our machine. $x := [E]$ and $[E] := E'$ correspond to reading from and writing to the heap, respectively. $x := \texttt{cons}(E_1, \ldots, E_n)$ allocates a nondeterministically-chosen contiguous block of $n$ heap cells from the free list. The most interesting rules are those for allocation and deallocation, since they make use of the free list. Note that none of the operations make use of any memory existing outside the program state — this is the key for obtaining behavior-preservation.

To see how out state model fits into the structure defined in Section 2.1, we need to define the state combination operator. Given two states $\sigma_1 = (s_1, h_1, f_1)$ and $\sigma_2 = (s_2, h_2, f_2)$, the combined state $\sigma_1 \bullet \sigma_2$ is equal to $(s_1, h_1 \uplus h_2, f_1)$ if $s_1 = s_2$, $f_1 = f_2$, and the domains of $h_1$ and $h_2$ are disjoint; otherwise, the combination is undefined.

$$\frac{}{\sigma, \texttt{skip}; C \longrightarrow \sigma, C} \text{ (SKIP)}$$

$$\frac{}{(s, h, f), x := E \longrightarrow (s[x \mapsto [\![E]\!]s], h, f), \texttt{skip}} \text{ (ASSGN)}$$

$$\frac{[\![E]\!]s \in \lfloor h \rfloor}{(s, h, f), x := [E] \longrightarrow (s[x \mapsto h([\![E]\!]s)], h, f), \texttt{skip}} \text{ (HEAP-READ)}$$

$$\frac{[\![E]\!]s \in \lfloor h \rfloor}{(s, h, f), [E] := E' \longrightarrow (s, h[[\![E]\!]s \mapsto [\![E']\!]s], f), \texttt{skip}} \text{ (HEAP-WRITE)}$$

$$\frac{\forall i \in [1, n] \,.\, l + i - 1 \in f}{\begin{array}{c}(s, h, f), x := \texttt{cons}(E_1, \ldots, E_n) \longrightarrow \\ (s[x \mapsto l], h[l \mapsto [\![E_1]\!]s] \ldots [l + n - 1 \mapsto [\![E_n]\!]s], f - \{l, \ldots, l + n - 1\}), \texttt{skip}\end{array}} \text{ (CONS)}$$

$$\frac{[\![E]\!]s \in \lfloor h \rfloor}{(s, h, f), \texttt{free}\,(E) \longrightarrow (s, h \backslash [\![E]\!]s, f \cup \{[\![E]\!]s\}), \texttt{skip}} \text{ (FREE)}$$

$$\frac{\sigma, C \longrightarrow \sigma', C'}{\sigma, C; C'' \longrightarrow \sigma', C'; C''} \text{ (SEQ)} \qquad \frac{[\![B]\!]s = \texttt{true}}{\sigma, \texttt{if}\, B\, \texttt{then}\, C_1\, \texttt{else}\, C_2 \longrightarrow \sigma, C_1} \text{ (IF-TRUE)}$$

$$\frac{[\![B]\!]s = \texttt{false}}{\sigma, \texttt{if}\, B\, \texttt{then}\, C_1\, \texttt{else}\, C_2 \longrightarrow \sigma, C_2} \text{ (IF-FALSE)}$$

$$\frac{[\![B]\!]s = \texttt{true}}{\sigma, \texttt{while}\, B\, \texttt{do}\, C \longrightarrow \sigma, C; \texttt{while}\, B\, \texttt{do}\, C} \text{ (WHILE-TRUE)}$$

$$\frac{[\![B]\!]s = \texttt{false}}{\sigma, \texttt{while}\, B\, \texttt{do}\, C \longrightarrow \sigma, \texttt{skip}} \text{ (WHILE-FALSE)}$$

Figure 2.3: Small-Step Operational Semantics

Note that this combined state satisfies the requisite condition $\mathrm{dom}(h_1 \uplus h_2) \cap f_1 = \emptyset$ because $h_1$, $h_2$, and $f_1$ are pairwise disjoint by assumption. The most important aspect of this definition of state combination is that we can never change the free list when adding extra resources. This guarantees behavior preservation of the nondeterministic memory allocator because the allocator's set of possible behaviors is precisely defined by the free list.

In order to formally compare our logic to standard Separation Logic, we need to provide the standard version of the small-step operational semantics, denoted as $(s, h), C \rightsquigarrow (s', h'), C'$. This definition is nearly identical to Figure 2.3, except that all free lists are removed from program state, and the (CONS) rule precondition is modified to only require that the newly-allocated locations are not in $\lfloor h \rfloor$. It is then possible to show the following relationship between the two operational semantics (note that the full proofs for all of the following lemmas and theorems can be found in our Coq implementation [11]):

**Lemma 1.**

$$(s, h), C \overset{n}{\rightsquigarrow} (s', h'), C' \iff \exists f, f' \,.\, (s, h, f), C \xrightarrow{n} (s', h', f'), C'$$

*Proof.* The backwards direction is a straightforward proof by induction. For the forwards direction, we actually prove a stronger statement by picking our $f$ and $f'$ to be exactly $\mathbb{N} - \lfloor h \rfloor$ and $\mathbb{N} - \lfloor h' \rfloor$, respectively. The proof of this stronger statement is then straightforward by induction. Picking the free lists in this way showcases how the Separation Logic model can be interpreted as having an implicit free list containing everything not in the heap. □

The inference rules in the form $\vdash \{P\}\, C\, \{Q\}$ for our logic are exactly the same as those used in standard Separation Logic. We give many of these inference rules in Figure 2.4; the reader may refer to [46] for more.

$$\frac{}{\vdash \{\mathtt{emp}\}\,\mathtt{skip}\,\{\mathtt{emp}\}}\ (\text{SKIP}) \qquad \frac{}{\vdash \{x = y \wedge \mathtt{emp}\}\,x := E\,\{x = E[y/x] \wedge \mathtt{emp}\}}\ (\text{ASSGN})$$

$$\frac{}{\vdash \{x = y \wedge E \mapsto z\}\,x := [E]\,\{x = z \wedge E[y/x] \mapsto z\}}\ (\text{HEAP-READ})$$

$$\frac{}{\vdash \{E \mapsto -\}\,[E] := E'\,\{E \mapsto E'\}}\ (\text{HEAP-WRITE})$$

$$\frac{}{\vdash \{x = y \wedge \mathtt{emp}\}\,x := \mathtt{cons}(E_1, \ldots, E_k)\,\{x \mapsto E_1[y/x], \ldots, E_k[y/x]\}}\ (\text{CONS})$$

$$\frac{}{\vdash \{E \mapsto -\}\,\mathtt{free}\,(E)\,\{\mathtt{emp}\}}\ (\text{FREE}) \qquad \frac{\vdash \{P\}\,C_1\,\{Q\} \qquad \vdash \{Q\}\,C_2\,\{R\}}{\vdash \{P\}\,C_1; C_2\,\{R\}}\ (\text{SEQ})$$

$$\frac{\vdash \{B \wedge P\}\,C_1\,\{Q\} \qquad \vdash \{\neg B \wedge P\}\,C_2\,\{Q\}}{\vdash \{P\}\,\mathtt{if}\,B\,\mathtt{then}\,C_1\,\mathtt{else}\,C_2\,\{Q\}}\ (\text{IF})$$

$$\frac{\vdash \{B \wedge P\}\,C\,\{P\}}{\vdash \{P\}\,\mathtt{while}\,B\,\mathtt{do}\,C\,\{\neg B \wedge P\}}\ (\text{WHILE})$$

$$\frac{P' \Rightarrow P \qquad Q \Rightarrow Q' \qquad \vdash \{P\}\,C\,\{Q\}}{\vdash \{P'\}\,C\,\{Q'\}}\ (\text{CONSEQ})$$

$$\frac{\vdash \{P_1\}\,C\,\{Q_1\} \qquad \vdash \{P_2\}\,C\,\{Q_2\}}{\vdash \{P_1 \wedge P_2\}\,C\,\{Q_1 \wedge Q_2\}}\ (\text{CONJ})$$

$$\frac{\vdash \{P_1\}\,C\,\{Q_1\} \qquad \vdash \{P_2\}\,C\,\{Q_2\}}{\vdash \{P_1 \vee P_2\}\,C\,\{Q_1 \vee Q_2\}}\ (\text{DISJ})$$

$$\frac{\vdash \{P\}\,C\,\{Q\} \qquad \mathtt{modifies}(C) \cap \mathtt{vars}(R) = \emptyset}{\vdash \{P * R\}\,C\,\{Q * R\}}\ (\text{FRAME})$$

Figure 2.4: Some Separation Logic Inference Rules

We next define safe execution and semantic triples. A configuration $(\sigma, C)$ is *safe* if it can never get stuck in a non-halting state:

$$\texttt{safe}(\sigma, C) \stackrel{\triangle}{=} \forall \sigma', C' \,.\, \sigma, C \xrightarrow{*} \sigma', C' \wedge C' \neq \texttt{skip} \implies \exists \sigma'', C'' \,.\, \sigma', C' \longrightarrow \sigma'', C''$$

A triple $\models \{P\}\, C\, \{Q\}$ is then *semantically valid* when, for all $\sigma$, $\sigma'$:

1.) if $\sigma \models P$, then $\texttt{safe}(\sigma, C)$

2.) if $\sigma \models P$ and $\sigma, C \xrightarrow{*} \sigma', \texttt{skip}$, then $\sigma' \models Q$

Semantic validity of standard Separation Logic triples is defined in the same way, but using the operational semantics for Separation Logic. We will write this as $\models_{SL} \{P\}\, C\, \{Q\}$. Note that we are only considering a *partial correctness* definition of validity here, meaning that programs are not required to terminate.

We now formally relate semantic validity of our logic with standard Separation Logic, with the help of a minor technical lemma:

**Lemma 2.**

$$(s, h), C \rightsquigarrow (s', h'), C' \implies \forall f \,.\, (f \# h \Rightarrow \exists \sigma \,.\, (s, h, f), C \longrightarrow \sigma, C')$$

*Proof.* Straightforward by induction on the rules for stepping. □

**Theorem 1** (Equivalence of Semantic Validity)**.**

$$\models_{SL} \{P\}\, C\, \{Q\} \iff \models \{P\}\, C\, \{Q\}$$

*Proof.* First, suppose that $\models_{SL} \{P\}\, C\, \{Q\}$. To prove the first property of semantic validity, suppose that $(s, h, f) \models P$, and consider some execution $(s, h, f), C \xrightarrow{*} (s', h', f'), C'$ with $C' \neq \texttt{skip}$. Then we need to show that $(s', h', f'), C'$ can take

another step. By Lemma 1, we have that $(s,h), C \overset{*}{\rightsquigarrow} (s',h'), C'$. Since $(s,h) \models P$, we know that $\texttt{safe}((s,h), C)$, and so $(s',h'), C' \rightsquigarrow (s'',h''), C''$ for some $s''$, $h''$, $C''$. Therefore Lemma 2 tells us that $(s',h',f'), C'$ can indeed take a step. For the second property, suppose that $(s,h,f) \models P$ and $(s,h,f), C \xrightarrow{*} (s',h',f'), \texttt{skip}$. Then Lemma 1 tells us that $(s,h), C \overset{*}{\rightsquigarrow} (s',h'), \texttt{skip}$, meaning that $(s',h') \models Q$, and so $(s',h',f') \models Q$.

Now suppose that $\models \{P\}\, C\, \{Q\}$. For the first property, suppose that $(s,h) \models P$ and $(s,h), C \overset{*}{\rightsquigarrow} (s',h'), C'$ with $C' \neq \texttt{skip}$. Lemma 1 gives us $(s,h,f), C \xrightarrow{*} (s',h',f'), C'$ for some $f$ and $f'$, which means that $(s',h',f'), C' \longrightarrow (s'',h'',f''), C''$ for some $s''$, $h''$, $f''$, $C''$ (since $(s,h,f) \models P$). Therefore Lemma 1 gives us $(s',h'), C' \rightsquigarrow (s'',h''), C''$, as desired. For the second property, suppose $(s,h) \models P$ and $(s,h), C \overset{*}{\rightsquigarrow} (s',h'), \texttt{skip}$. By Lemma 1, we have

$$(s,h,f), C \overset{*}{\rightsquigarrow} (s',h',f'), \texttt{skip}$$

for some $f$ and $f'$. Since $(s,h,f) \models P$, this means that $(s',h',f') \models Q$, and so $(s',h') \models Q$. □

**Theorem 2** (Soundness and Completeness)**.**

$$\vdash \{P\}\, C\, \{Q\} \iff \models \{P\}\, C\, \{Q\}$$

*Proof.* Note that $\vdash \{P\}\, C\, \{Q\}$ has the same definition in both our logic and in Separation Logic, since we use the same assertion language and inference rules. Therefore, because Separation Logic is known to be sound and complete, we have that $\vdash \{P\}\, C\, \{Q\} \iff \models_{SL} \{P\}\, C\, \{Q\}$. Applying Theorem 1 gives the desired result. □

We have thus shown that our new model does not cause any complications in the usage of Separation Logic. Any specification that can be proved using the standard

model can also be proved using our model, with the exact same application of inference rules (since they completely ignore the free list within program state). We now only need to show that our model enjoys the stronger, behavior-preserving notion of locality. As described in Section 2.1, this locality is composed of Safety Monotonicity, Termination Equivalence, and the Forward and Backwards Frame Properties. We first prove that the two frame properties hold:

**Theorem 3** (Frame Properties)**.**

$$\begin{aligned}
&\textit{1.)}\ (s, h_0, f), C \xrightarrow{n} (s', h_0', f'), C' \wedge h_0 \# h_1 \wedge f \# h_1 \Longrightarrow \\
&\qquad h_0' \# h_1 \wedge (s, h_0 \bullet h_1, f), C \xrightarrow{n} (s', h_0' \bullet h_1, f'), C' \\
&\textit{2.)}\ \texttt{safe}((s, h_0, f), C) \wedge (s, h_0 \bullet h_1, f), C \xrightarrow{n} (s', h', f'), C' \Longrightarrow \\
&\qquad \exists h_0' \,.\, h' = h_0' \bullet h_1 \wedge (s, h_0, f), C \xrightarrow{n} (s', h_0', f'), C'
\end{aligned}$$

*Proof.* Straightforward by induction on the derivation rules for stepping. For details, see the Coq implementation. □

It is then easy to show that these Frame Properties imply both Safety Monotonicity and Termination Equivalence.

**Lemma 3** (Safety Monotonicity)**.**

$$\texttt{safe}((s, h_0, f), C) \wedge h_0 \# h_1 \wedge f \# h_1 \Longrightarrow \texttt{safe}((s, h_0 \bullet h_1, f), C)$$

*Proof.* Suppose that $\texttt{safe}((s, h_0, f), C)$, and consider an execution on the large state $(s, h_0 \bullet h_1, f), C \xrightarrow{n} (s', h', f'), C'$ with $C' \neq \texttt{skip}$. Then the Backwards Frame Property tells us that $h' = h_0' \bullet h_1$ and $(s, h_0, f), C \xrightarrow{n} (s', h_0', f'), C'$. Since $\texttt{safe}((s, h_0, f), C)$ and $C' \neq \texttt{skip}$, we see that $(s', h_0', f'), C' \longrightarrow (s'', h_0'', f''), C''$ for some $s''$, $h_0''$, $f''$, $C''$. Thus we can now use the Forwards Frame Property (clearly $h_1 \# f'$ since $(s', h_0' \bullet h_1, f')$

is a well-typed state) to obtain $(s', h'_0 \bullet h_1, f'), C' \longrightarrow (s'', h''_0 \bullet h_1, f''), C''$, and so $\texttt{safe}((s, h_0 \bullet h_1, f), C)$ does indeed hold. □

In order to define Termination Equivalence, we first need to define divergence. We say that $\sigma$ diverges on $C$, written $\sigma, C \uparrow$, if there exists an infinite path of steps starting from $\sigma, C$. More formally:

$$\sigma, C \uparrow \stackrel{\triangle}{=} \forall n \,.\, \exists \sigma', C' \,.\, \sigma, C \xrightarrow{n} \sigma', C'$$

**Lemma 4** (Termination Equivalence)**.**

$$\texttt{safe}((s, h_0, f), C) \wedge h_0 \# h_1 \wedge f \# h_1 \Rightarrow ((s, h_0, f), C \uparrow \iff (s, h_0 \bullet h_1, f), C \uparrow)$$

*Proof.* First, suppose $(s, h_0, f), C \uparrow$, and pick any $n$. Then there are some $s'$, $h'_0$, $f'$, $C'$ such that $(s, h_0, f), C \xrightarrow{n} (s', h'_0, f'), C'$. Thus the Forwards Frame Property tells us that $h'_0 \# h_1$ and $(s, h_0 \bullet h_1, f), C \xrightarrow{n} (s', h'_0 \bullet h_1, f'), C'$, as desired. For the other direction, suppose $(s, h_0 \bullet h_1, f), C$ and pick any $n$. Then $(s, h_0 \bullet h_1, f), C \xrightarrow{n} (s', h', f'), C'$ for some $s'$, $h'$, $f'$, $C'$. Since $\texttt{safe}((s, h_0, f), C)$, the Backwards Frame Property tells us that $h' = h'_0 \bullet h_1$ for some $h'_0$, and $(s, h_0, f), C \xrightarrow{n} (s', h'_0, f'), C'$, as desired. □

## 2.3 The Abstract Logic

The previous section demonstrated how we can impose behavior preservation in the context of Separation Logic, without making Separation Logic any more difficult to use or any less powerful. We next need to show how behavior preservation can provide benefits over the standard, weaker notion of locality. In order to do this, it will help to have a formal, abstract view of behavior-preserving Separation Logic. This section will describe how our strong locality fits into a context similar to that of Abstract

Separation Logic [9]. With a minor amount of work, the logic of Section 2.2 can be molded into a particular instance of the abstract logic presented here.

We define a *separation algebra* to be a set of states $\Sigma$, along with a partial associative and commutative operator $\bullet : \Sigma \to \Sigma \rightharpoonup \Sigma$. The disjointness relation $\sigma_0 \# \sigma_1$ holds iff $\sigma_0 \bullet \sigma_1$ is defined, and the substate relation $\sigma_0 \preceq \sigma_1$ holds iff there is some $\sigma_0'$ such that $\sigma_0 \bullet \sigma_0' = \sigma_1$. A particular element of $\Sigma$ is designated as a unit state, denoted $u$, with the property that for any $\sigma$, $\sigma \# u$ and $\sigma \bullet u = \sigma$. We require the $\bullet$ operator to be cancellative, meaning that $\sigma \bullet \sigma_0 = \sigma \bullet \sigma_1 \Rightarrow \sigma_0 = \sigma_1$.

An *action* is a set of pairs of type $\Sigma \cup \{\texttt{fault}, \texttt{div}\} \times \Sigma \cup \{\texttt{fault}, \texttt{div}\}$. We require the following two properties: (1) actions always relate `fault` to `fault` and `div` to `div`, and never relate `fault` or `div` to anything else; and (2) actions are total, in the sense that for any $\tau$, there exists some $\tau'$ such that $\tau \, [A] \, \tau'$ (recall from Section 2.1 that we use $\tau$ to range over elements of $\Sigma \cup \{\texttt{fault}, \texttt{div}\}$). Note that these two requirements are preserved over the standard composition of relations, as well as over both finitary and infinite unions. We write `Id` to represent the identity action $\{(\tau, \tau) \mid \tau \in \Sigma \cup \{\texttt{fault}, \texttt{div}\}\}$.

Note that it is more standard in the literature to have the domain of actions range only over $\Sigma$ — we use $\Sigma \cup \{\texttt{fault}, \texttt{div}\}$ here because it has the pleasant effect of making $[\![C_1; C_2]\!]$ correspond precisely to standard composition. Intuitively, once an execution goes wrong, it continues to go wrong, and once an execution diverges, it continues to diverge.

A *local action* is an action $A$ that satisfies the following four properties, which respectively correspond to Safety Monotonicity, Termination Equivalence, the Forwards

Frame Property, and the Backwards Frame Property from Section 2.1:

$$1.)\ \neg\sigma_0\,[A]\,\mathtt{fault} \wedge \sigma_0\#\sigma_1 \Longrightarrow \neg(\sigma_0 \bullet \sigma_1)\,[A]\,\mathtt{fault}$$

$$2.)\ \neg\sigma_0\,[A]\,\mathtt{fault} \wedge \sigma_0\#\sigma_1 \Longrightarrow (\sigma_0\,[A]\,\mathtt{div} \iff (\sigma_0 \bullet \sigma_1)\,[A]\,\mathtt{div})$$

$$3.)\ \sigma_0\,[A]\,\sigma_0' \wedge \sigma_0\#\sigma_1 \Longrightarrow \sigma_0'\#\sigma_1 \wedge (\sigma_0 \bullet \sigma_1)\,[A]\,(\sigma_0' \bullet \sigma_1)$$

$$4.)\ \neg\sigma_0\,[A]\,\mathtt{fault} \wedge (\sigma_0 \bullet \sigma_1)\,[A]\,\sigma' \Longrightarrow \exists\sigma_0' \,.\, \sigma' = \sigma_0' \bullet \sigma_1 \wedge \sigma_0\,[A]\,\sigma_0'$$

We denote the set of all local actions by **LocAct**. We now show that the set of local actions is closed under composition and (possibly infinite) union. We use the notation $A_1; A_2$ to denote composition, and $\bigcup \mathcal{A}$ to denote union (where $\mathcal{A}$ is a possibly infinite set of actions). The formal definitions of these operations follow. Note that we require that $\mathcal{A}$ be non-empty. This is necessary because $\bigcup \emptyset$ is $\emptyset$, which is not a valid action. Unless otherwise stated, whenever we write $\bigcup \mathcal{A}$, there will always be an implicit assumption that $\mathcal{A} \neq \emptyset$.

$$\tau\,[A_1; A_2]\,\tau' \iff \exists\tau'' \,.\, \tau\,[A_1]\,\tau'' \wedge \tau''\,[A_2]\,\tau'$$

$$\tau\left[\bigcup \mathcal{A}\right]\tau' \iff \exists A \in \mathcal{A} \,.\, \tau\,[A]\,\tau' \qquad (\mathcal{A} \neq \emptyset)$$

**Lemma 5.** *If $A_1$ and $A_2$ are local actions, then $A_1; A_2$ is a local action.*

*Proof.* It will be useful to first note that $\sigma\,[A_1; A_2]\,\mathtt{fault}$ iff either $\sigma\,[A_1]\,\mathtt{fault}$ or there exists some $\sigma'$ such that $\sigma\,[A_1]\,\sigma'$ and $\sigma'\,[A_2]\,\mathtt{fault}$. This is due to the fact that we know $\mathtt{fault}\,[A_2]\,\mathtt{fault}$ and $\neg\mathtt{div}\,[A_2]\,\mathtt{fault}$. Similarly, it is also the case that $\sigma\,[A_1; A_2]\,\mathtt{div}$ iff either $\sigma\,[A_1]\,\mathtt{div}$ or there exists some $\sigma'$ such that $\sigma\,[A_1]\,\sigma'$ and $\sigma'\,[A_2]\,\mathtt{div}$.

For Safety Monotonicity, suppose that $\sigma_0\#\sigma_1$ and $\neg\sigma_0\,[A_1; A_2]\,\mathtt{fault}$. Suppose by way of contradiction that $(\sigma_0 \bullet \sigma_1)\,[A_1; A_2]\,\mathtt{fault}$. Since $\neg\sigma_0\,[A_1; A_2]\,\mathtt{fault}$ and $\mathtt{fault}\,[A_2]\,\mathtt{fault}$, we have $\neg\sigma_0\,[A_1]\,\mathtt{fault}$. Thus by Safety Monotonicity of $A_1$, $\neg(\sigma_0 \bullet$

$\sigma_1)\,[A_1]\,\texttt{fault}$. By our note above, we see that there must be some $\sigma$ such that $(\sigma_0 \bullet \sigma_1)\,[A_1]\,\sigma$ and $\sigma\,[A_2]\,\texttt{fault}$. By the Backwards Frame Property of $A_1$, there must be a $\sigma_0'$ such that $\sigma = \sigma_0' \bullet \sigma_1$ and $\sigma_0\,[A_1]\,\sigma_0'$. Thus we have that $(\sigma_0' \bullet \sigma_1)\,[A_2]\,\texttt{fault}$, and so Safety Monotonicity of $A_2$ tells us that $\sigma_0'\,[A_2]\,\texttt{fault}$. Hence $\sigma_0\,[A_1; A_2]\,\texttt{fault}$, which is a contradiction.

For Termination Equivalence, suppose that $\sigma_0 \# \sigma_1$ and $\neg\sigma_0\,[A_1; A_2]\,\texttt{fault}$. Then we also have $\neg\sigma_0\,[A_1]\,\texttt{fault}$, since we have $\texttt{fault}\,[A_2]\,\texttt{fault}$.

For the forward direction, suppose that $\sigma_0\,[A_1; A_2]\,\texttt{div}$. By the note above, there are two possible situations. In the first situation, we have $\sigma_0\,[A_1]\,\texttt{div}$. By Termination Equivalence of $A_1$, this implies that $(\sigma_0 \bullet \sigma_1)\,[A_1]\,\texttt{div}$, and so $(\sigma_0 \bullet \sigma_1)\,[A_1; A_2]\,\texttt{div}$, as desired. In the second situation, there is a state $\sigma$ such that $\sigma_0\,[A_1]\,\sigma$ and $\sigma\,[A_2]\,\texttt{div}$. By the Forwards Frame Property of $A_1$, we see that $\sigma \# \sigma_1$ and $(\sigma_0 \bullet \sigma_1)\,[A_1]\,(\sigma \bullet \sigma_1)$. Now note that we must have $\neg\sigma\,[A_2]\,\texttt{fault}$, because otherwise we would be able to derive $\sigma_0\,[A_1; A_2]\,\texttt{fault}$, which is a contradiction. Therefore, by Termination Equivalence of $A_2$, we have $(\sigma \bullet \sigma_1)\,[A_2]\,\texttt{div}$. Hence we get $(\sigma_0 \bullet \sigma_1)\,[A_1; A_2]\,\texttt{div}$, as desired.

For the backward direction, suppose that $(\sigma_0 \bullet \sigma_1)\,[A_1; A_2]\,\texttt{div}$. Again by the note above, there are two possible situations. In the first situation, we have $(\sigma_0 \bullet \sigma_1)\,[A_1]\,\texttt{div}$. By Termination Equivalence of $A_1$, this implies that $\sigma_0\,[A_1]\,\texttt{div}$, and so $\sigma_0\,[A_1; A_2]\,\texttt{div}$, as desired. In the second situation, there is a state $\sigma$ such that $(\sigma_0 \bullet \sigma_1)\,[A_1]\,\sigma$ and $\sigma\,[A_2]\,\texttt{div}$. By the Backwards Frame Property of $A_1$, there must be a $\sigma_0'$ such that $\sigma = \sigma_0' \bullet \sigma_1$ and $\sigma_0\,[A_1]\,\sigma_0'$. Now note that we must have $\neg\sigma_0'\,[A_2]\,\texttt{fault}$, because otherwise we would be able to derive $\sigma_0\,[A_1; A_2]\,\texttt{fault}$, which is a contradiction. Therefore, by Termination Equivalence of $A_2$, we have $\sigma_0'\,[A_2]\,\texttt{div}$. Hence we get $\sigma_0\,[A_1; A_2]\,\texttt{div}$, as desired.

For the Forwards Frame Property, suppose that $\sigma_0 \# \sigma_1$ and $\sigma_0\,[A_1; A_2]\,\sigma_0'$. Then there exists a $\tau$ such that $\sigma_0\,[A_1]\,\tau$ and $\tau\,[A_2]\,\sigma_0'$. Furthermore, $\tau$ cannot be $\texttt{fault}$ or

$\texttt{div}$ since $\tau\,[A_2]\,\sigma_0'$ — thus let $\tau$ be $\sigma_0''$. By the Forwards Frame Property of $A_1$, we have $\sigma_0''\#\sigma_1$ and $(\sigma_0 \bullet \sigma_1)\,[A_1]\,(\sigma_0'' \bullet \sigma_1)$. Therefore, by the Forwards Frame Property of $A_2$, we have $\sigma_0'\#\sigma_1$ and $(\sigma_0'' \bullet \sigma_1)\,[A_2]\,(\sigma_0' \bullet \sigma_1)$. Hence $\sigma_0'\#\sigma_1$ and $(\sigma_0 \bullet \sigma_1)\,[A_1; A_2]\,(\sigma_0' \bullet \sigma_1)$, as desired.

For the Backwards Frame Property, suppose that $\neg\sigma_0\,[A_1; A_2]\,\texttt{fault}$ and $(\sigma_0 \bullet \sigma_1)\,[A_1; A_2]\,\sigma'$. Then, repeating some reasoning from earlier in this proof, we have $\neg\sigma_0\,[A_1]\,\texttt{fault}$, and there exists a $\sigma$ such that $(\sigma_0 \bullet \sigma_1)\,[A_1]\,\sigma$ and $\sigma\,[A_2]\,\sigma'$. By the Backwards Frame Property of $A_1$, we get $\sigma = \sigma_0' \bullet \sigma_1$ and $\sigma_0\,[A_1]\,\sigma_0'$. Now note that $\neg\sigma_0'\,[A_2]\,\texttt{fault}$, because otherwise we would be able to derive $\sigma_0\,[A_1; A_2]\,\texttt{fault}$, which is a contradiction. Therefore, by the Backwards Frame Property of $A_2$, we get $\sigma' = \sigma_0'' \bullet \sigma_1$ and $\sigma_0'\,[A_2]\,\sigma_0''$. Hence $\sigma' = \sigma_0'' \bullet \sigma_1$ and $\sigma_0\,[A_1; A_2]\,\sigma_0''$, as desired. □

**Lemma 6.** *If every $A$ in the set $\mathcal{A}$ is a local action, then $\bigcup\mathcal{A}$ is a local action.*

*Proof.* For Safety Monotonicity, suppose $\sigma_0\#\sigma_1$ and $\neg\sigma_0\,[\bigcup\mathcal{A}]\,\texttt{fault}$. Suppose by way of contradiction that $(\sigma_0 \bullet \sigma_1)\,[\bigcup\mathcal{A}]\,\texttt{fault}$. Then there is some $A \in \mathcal{A}$ such that $(\sigma_0 \bullet \sigma_1)\,[A]\,\texttt{fault}$. By Safety Monotonicity of $A$, we get $\sigma_0\,[A]\,\texttt{fault}$. But this means that $\sigma_0\,[\bigcup\mathcal{A}]\,\texttt{fault}$, which is a contradiction.

For Termination Equivalence, suppose that $\sigma_0\#\sigma_1$ and $\neg\sigma_0\,[\bigcup\mathcal{A}]\,\texttt{fault}$. This means that for every $A \in \mathcal{A}$, $\neg\sigma_0\,[A]\,\texttt{fault}$. For the forward direction, suppose that $\sigma_0\,[\bigcup\mathcal{A}]\,\texttt{div}$. Then $\sigma_0\,[A]\,\texttt{div}$ for some $A \in \mathcal{A}$. Thus Termination Equivalence of $A$ gives us $(\sigma_0 \bullet \sigma_1)\,[A]\,\texttt{div}$, and so we get the desired $(\sigma_0 \bullet \sigma_1)\,[\bigcup\mathcal{A}]\,\texttt{div}$. For the backward direction, suppose that $(\sigma_0 \bullet \sigma_1)\,[\bigcup\mathcal{A}]\,\texttt{div}$. Then $(\sigma_0 \bullet \sigma_1)\,[A]\,\texttt{div}$ for some $A \in \mathcal{A}$. Thus Termination Equivalence of $A$ gives us $\sigma_0\,[A]\,\texttt{div}$, and so we get the desired $\sigma_0\,[\bigcup\mathcal{A}]\,\texttt{div}$.

For the Forwards Frame Property, suppose that $\sigma_0\#\sigma_1$ and $\sigma_0\,[\bigcup\mathcal{A}]\,\sigma_0'$. Then $\sigma_0\,[A]\,\sigma_0'$ for some $A \in \mathcal{A}$, and so by the Forwards Frame Property of $A$, we have $\sigma_0'\#\sigma_1$ and $(\sigma_0 \bullet \sigma_1)\,[A]\,(\sigma_0' \bullet \sigma_1)$, which in turn implies the desired result.

For the Backwards Frame Property, suppose that $\neg\sigma_0\,[\bigcup\mathcal{A}]\,\texttt{fault}$ and $(\sigma_0 \bullet$

$$C \ ::= c \mid C_1; C_2 \mid C_1 + C_2 \mid C^*$$

$$\forall c \,.\, [\![c]\!] \in \mathbf{LocAct} \qquad [\![C_1; C_2]\!] \triangleq [\![C_1]\!]; [\![C_2]\!]$$

$$[\![C_1 + C_2]\!] \triangleq [\![C_1]\!] \cup [\![C_2]\!] \qquad [\![C^*]\!] \triangleq \bigcup_{n \in \mathbb{N}} [\![C]\!]^n$$

$$[\![C]\!]^0 \triangleq \mathtt{Id} \qquad [\![C]\!]^{n+1} \triangleq [\![C]\!]; [\![C]\!]^n$$

Figure 2.5: Command Definition and Denotational Semantics

$\sigma_1) [\bigcup \mathcal{A}] \sigma'$. Then $(\sigma_0 \bullet \sigma_1) [A] \sigma'$ for some $A \in \mathcal{A}$, and for all $A \in \mathcal{A}$ we have $\neg \sigma_0 [A] \, \mathtt{fault}$. Hence the Backwards Frame Property of $A$ tells us that $\sigma' = \sigma'_0 \bullet \sigma_1$ and $\sigma_0 [A] \sigma'_0$, which implies the desired result. □

Figure 2.5 defines our abstract program syntax and semantics. The language consists of primitive commands, sequencing ($C_1; C_2$), nondeterministic choice ($C_1 + C_2$), and finite iteration ($C^*$). The semantics of primitive commands are abstracted — the only requirement is that they are local actions. Therefore, from the two previous lemmas and the trivial fact that `Id` is a local action, it is clear that the semantics of *every* program is a local action.

Note that our concrete language in Section 2.2 used if statements and while loops. As shown in [9], it is possible to represent if and while constructs with finite iteration and nondeterministic choice by including a primitive command $\mathtt{assume}(B)$, which does nothing if the boolean expression $B$ is true, and diverges otherwise. Given this setup, we can define the primitive command $\mathtt{assume}(B)$ as follows:

$$\begin{aligned}[\![\mathtt{assume}(B)]\!] \triangleq \; & \{(\mathtt{fault}, \mathtt{fault}), (\mathtt{div}, \mathtt{div})\} \cup \\ & \{(\sigma, \sigma) \mid [\![B]\!]\sigma = \mathtt{true}\} \cup \{(\sigma, \mathtt{div}) \mid [\![B]\!]\sigma = \mathtt{false}\} \cup \\ & \{(\sigma, \mathtt{fault}) \mid [\![B]\!]\sigma \text{ undefined}\}\end{aligned}$$

It is a simple matter to show that this is a local action. We can then syntactically

$$\frac{\neg\sigma[\![c]\!]\texttt{fault}}{\vdash \{\{\sigma\}\}\, c\, \{\{\sigma' \mid \sigma[\![c]\!]\sigma'\}\}}\ (\text{PRIM}) \qquad \frac{\vdash \{P\}\, C_1\, \{Q\} \qquad \vdash \{Q\}\, C_2\, \{R\}}{\vdash \{P\}\, C_1; C_2\, \{R\}}\ (\text{SEQ})$$

$$\frac{\vdash \{P\}\, C_1\, \{Q\} \qquad \vdash \{P\}\, C_2\, \{Q\}}{\vdash \{P\}\, C_1 + C_2\, \{Q\}}\ (\text{PLUS}) \qquad \frac{\vdash \{P\}\, C\, \{P\}}{\vdash \{P\}\, C^*\, \{P\}}\ (\text{STAR})$$

$$\frac{\vdash \{P\}\, C\, \{Q\}}{\vdash \{P * R\}\, C\, \{Q * R\}}\ (\text{FRAME}) \qquad \frac{P' \subseteq P \qquad \vdash \{P\}\, C\, \{Q\} \qquad Q \subseteq Q'}{\vdash \{P'\}\, C\, \{Q'\}}\ (\text{CONSEQ})$$

$$\frac{\forall i \in I\ .\ \vdash \{P_i\}\, C\, \{Q_i\}}{\vdash \{\bigcup P_i\}\, C\, \{\bigcup Q_i\}}\ (\text{DISJ}) \qquad \frac{\forall i \in I\ .\ \vdash \{P_i\}\, C\, \{Q_i\} \qquad I \neq \emptyset}{\vdash \{\bigcap P_i\}\, C\, \{\bigcap Q_i\}}\ (\text{CONJ})$$

Figure 2.6: Inference Rules

define if and while statements as follows:

$$\texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2 \triangleq (\texttt{assume}(B); C_1) + (\texttt{assume}(\neg B); C_2)$$

$$\texttt{while}\ B\ \texttt{do}\ C \triangleq (\texttt{assume}(B); C)^*; \texttt{assume}(\neg B)$$

Technically, these definitions only correctly implement if and while statements in terms of which states they can terminate at — they do not correctly implement divergence behavior since they allow for arbitrary divergence. Thus these definitions should only be used if we do not care about divergence behavior. It is certainly still possible to define fully correct if and while statements, but the technical details are outside the scope of this work.

Now that we have defined the interpretation of programs as local actions, we can talk about the meaning of a triple $\{P\}\, C\, \{Q\}$. We define an assertion $P$ to be a set of states, and we say that a state $\sigma$ satisfies $P$ iff $\sigma \in P$. We can then define the separating conjunction as follows:

$$P * Q \triangleq \{\sigma \in \Sigma \mid \exists \sigma_0 \in P, \sigma_1 \in Q\ .\ \sigma = \sigma_0 \bullet \sigma_1\}$$

Given an assignment of primitive commands to local actions, we say that a triple is valid, written $\models \{P\}\, C\, \{Q\}$, just when the following two properties hold for all states $\sigma$ and $\sigma'$:

$$1.)\ \sigma \in P \Longrightarrow \neg\sigma[\![C]\!]\texttt{fault}$$

$$2.)\ \sigma \in P \wedge \sigma[\![C]\!]\sigma' \Longrightarrow \sigma' \in Q$$

The inference rules of the logic are given in Figure 2.6. Note that we are taking a significant presentation shortcut here in the inference rule for primitive commands. Specifically, we assume that we know the exact local action $[\![c]\!]$ of each primitive command $c$. This assumption makes sense when we define our own primitive commands, as we do in the logic of Section 2.2. However, in a more general setting, we might be provided with an opaque function along with a specification (precondition and postcondition) for the function. Since the function is opaque, we must consider it to be a primitive command in the abstract setting. Yet we do not know how it is implemented, so we do not know its precise local action. In [9], the authors provide a method for inferring a "best" local action from the function's specification. With a decent amount of technical development, we can do something similar here, using our stronger definition of locality. These details can be found in the technical report of our APLAS publication [13].

Given this setup, we can now prove soundness and completeness of our abstract logic. The details of the proof can be found in our Coq implementation [11].

**Theorem 4** (Soundness and Completeness)**.**

$$\vdash \{P\}\, C\, \{Q\} \iff \models \{P\}\, C\, \{Q\}$$

## 2.4 Applications of Behavior Preservation

Now that we have an abstracted formalism of our behavior-preserving local actions, we can demonstrate some ways in which behavior preservation yields significant benefits and simplifications. We will first present four Separation Logic metatheory issues which are greatly simplified by enforcing behavior preservation; these four issues are described in full detail (including proofs) in the technical report of our APLAS publication [13]. Then we will conclude by connecting this chapter to the dissertation's main contributions, with a discussion of how behavior preservation is important for security verification.

### 2.4.1 Footprints and Smallest Safe States

Consider a situation in which we are handed a program $C$ along with a specification of what this program does. The specification consists of a set of axioms; each axiom has the form $\{P\}\, C\, \{Q\}$ for some precondition $P$ and postcondition $Q$. A common question to ask would be: is this specification *complete*? In other words, if the triple $\models \{P\}\, C\, \{Q\}$ is valid for some $P$ and $Q$, then is it possible to derive $\vdash \{P\}\, C\, \{Q\}$ from the provided specification?

In standard Separation Logic, it can be extremely difficult to answer this question. In [45], the authors conduct an in-depth study of various conditions and circumstances under which it is possible to prove that certain specifications are complete. However, in the general case, there is no easy way to prove this.

We can show that under our assumption of behavior preservation, there is a very easy way to guarantee that a specification is complete. In particular, a specification that describes the exact behavior of $C$ on all of its *smallest safe states* is always complete. Formally, a smallest safe state is a state $\sigma$ such that $\neg\sigma[\![C]\!]\texttt{fault}$ and, for all $\sigma' \prec \sigma$, $\sigma'[\![C]\!]\texttt{fault}$.

To see that such a specification may not be complete in standard Separation Logic, we borrow an example from [45]. Consider the program $C$, defined as $x := \texttt{cons}(0); \texttt{free}(x)$. This program simply allocates a single cell and then frees it. Under the standard model, the smallest safe states are those of the form $(s, \emptyset)$ for any store $s$. For simplicity, assume that the only variables in the store are $x$ and $y$. Define the specification to be the infinite set of triples that have the following form, for any $a$, $b$ in $\mathbb{Z}$, and any $a'$ in $\mathbb{N}$:

$$\{x = a \wedge y = b \wedge \texttt{emp}\}\, C\, \{x = a' \wedge y = b \wedge \texttt{emp}\}$$

Note that $a'$ must be in $\mathbb{N}$ because only valid unallocated memory addresses can be assigned into $x$. It should be clear that this specification describes the exact behavior on all smallest safe states of $C$. Now we claim that the following triple is valid, but there is no way to derive it from the given specification.

$$\{x = a \wedge y = b \wedge y \mapsto -\}\, C\, \{x = a' \wedge y = b \wedge y \mapsto - \wedge a' \neq b\}$$

The triple is clearly valid because $a'$ must be a memory address that was initially unallocated, while address $b$ was initially allocated. Nevertheless, there will not be any way to derive this triple, even if we come up with new assertion syntax or inference rules. The behavior of $C$ on the larger state is different from the behavior on the small one, but there is no way to recover this fact once we make $C$ opaque. It can be shown (see [45]) that if we add triples of the above form to our specification, then we will obtain a complete specification for $C$. Yet there is no straightforward way to see that such a specification is complete.

When behavior preservation is enforced, there is a clean canonical form for complete specification. We say that a specification $\Psi$ is *complete for* $C$ if, whenever $\models \{P\}\, C\, \{Q\}$ is valid, the triple $\vdash \{P\}\, C\, \{Q\}$ is derivable using only the inference

rules that are not specific to the structure of $C$ (i.e., the frame, consequence, disjunction, and conjunction rules), plus the following axiom rule:

$$\frac{\{P\}\, C\, \{Q\} \in \Psi}{\vdash \{P\}\, C\, \{Q\}}$$

For any $\sigma$, let $\sigma[\![C]\!]$ denote the set of all $\sigma'$ such that $\sigma[\![C]\!]\sigma'$. For any set of states $S$, we define a *canonical specification on* $S$ as the set of triples of the form $\{\{\sigma\}\}\, C\, \{\sigma[\![C]\!]\}$ for any state $\sigma \in S$. If there exists a canonical specification on $S$ that is complete for $C$, then we say that $S$ forms a *footprint* for $C$.

**Theorem 5.** *For any program $C$, the set of all smallest safe states of $C$ forms a footprint for $C$.*

Note that while this theorem guarantees that the canonical specification is complete, we may not actually be able to write down the specification simply because the assertion language is not expressive enough. This would be the case for the behavior-preserving nondeterministic memory allocator if we used the assertion language presented in Section 2.2. We could, however, express canonical specifications in that system by extending the assertion language to talk about the free list.

### 2.4.2 Data Refinement

In [18], the goal is to formalize the concept of having a concrete module correctly implement an abstract one, within the context of Separation Logic. Specifically, the authors prove that as long as a client program "behaves nicely," any execution of the program using the concrete module can be tracked to a corresponding execution using the abstract module. The client states in the corresponding executions are identical, so the proof shows that a well-behaved client cannot tell the difference between the concrete and abstract modules.

To get their proof to work out, the authors require two somewhat odd properties to hold. The first is called *contents independence*, and is an extra condition on top of the standard locality conditions. The second is called a *growing relation* — it refers to the relation connecting a state of the abstract module to its logically equivalent state(s) in the concrete module. All relations connecting the abstract and concrete modules in this way are required to be growing, which means that the domain of memory used by the abstract state must be a subset of that used by the concrete state. This is a somewhat unintuitive and restrictive requirement which is needed for purely technical reasons. It turns out that behavior preservation completely eliminates the need for both contents independence and growing relations.

We now provide a formal setting for the data refinement theory. This formal setting is similar to the one in [18], but we will make some minor alterations to simplify the presentation. The programming language is defined as:

$$\begin{aligned} C \ ::= & \ \texttt{skip} \mid c \mid \texttt{m} \mid C_1; C_2 \mid \texttt{if}\, B\, \texttt{then}\, C_1\, \texttt{else}\, C_2 \\ & \mid \texttt{while}\, B\, \texttt{do}\, C \end{aligned}$$

$c$ is a primitive command (sometimes referred to as "client operation" in this context). `m` is a *module command* taken from an abstracted set **MOp** (e.g., a memory manager might implement the two module commands `cons` and `free`).

The primitive client and module commands are assumed to have a semantics mapping them to particular local actions. We of course use our behavior-preserving notion of "local" here, whereas in [18], the authors use the three properties of safety monotonicity, the (backwards) frame property, and a new property called contents independence. It is trivial to show that behavior preservation implies contents independence, as contents independence is essentially a forwards frame property that can only be applied under special circumstances.

A *module* is a pair $(p, \eta)$ representing a particular implementation of the module

commands in **MOp**; the state predicate $p$ describes the module's *invariant* (e.g., that a valid free list is stored starting at a location pointed to by a particular head pointer), while $\eta$ is a function mapping each module command to a particular local action. The predicate $p$ is required to be *precise* [44], meaning that no state can have more than one substate satisfying $p$ (if a state $\sigma$ does have a substate satisfying $p$, then we refer to that uniquely-defined state as $\sigma_p$). Additionally, all module operations are required to preserve the invariant $p$:

$$\neg\sigma\,[\eta\mathtt{m}]\,\mathtt{fault} \wedge \sigma \in p * \mathtt{true} \wedge \sigma\,[\eta\mathtt{m}]\,\sigma' \Longrightarrow \sigma' \in p * \mathtt{true}$$

We define a big-step operational semantics parameterized by a module $(p, \eta)$. This semantics is fundamentally the same as the one defined in [18]; the full details can be found in the APLAS technical report [13]. The only aspect that is important to mention here is that the semantics is equipped with a special kind of faulting called "access violation." Intuitively, an access violation occurs when a client operation's execution depends on the module's portion of memory. More formally, it occurs when the client operation executes safely on a state where the module's memory is present (i.e., a state satisfying $p * \mathtt{true}$), but faults when that memory is removed.

The main theorem that we get out of this setup is a refinement simulation between a program being run in the presence of an abstract module $(p, \eta)$, and the same program being run in the presence of a concrete module $(q, \mu)$ that implements the same module commands (i.e., $\lfloor\eta\rfloor = \lfloor\mu\rfloor$, where the floor notation indicates domain). Suppose we have a binary relation $R$ relating states of the abstract module to those of the concrete module. For example, if our modules are memory managers, then $R$ might relate a particular set of memory locations available for allocation to all lists containing that set of locations in some order. To represent that $R$ relates abstract module states to concrete module states, we require that whenever $\sigma_1 R \sigma_2$, $\sigma_1 \in p$

and $\sigma_2 \in q$. Given this relation $R$, we can make use of the separating conjunction of Relational Separation Logic [57] and write $R * \texttt{Id}$ to indicate the relation relating any two states of the form $\sigma_p \bullet \sigma_c$ and $\sigma_q \bullet \sigma_c$, where $\sigma_p R \sigma_q$.

Now, for any module $(p, \eta)$, let $C[(p, \eta)]$ be notation for the program $C$ whose semantics have $(p, \eta)$ filled in for the parameter module. Then our main theorem says that, if $\eta(f)$ simulates $\mu(f)$ under relation $R * \texttt{Id}$ for all $f \in \lfloor \eta \rfloor$, then for any program $C$, $C[(p, \eta)]$ also simulates $C[(q, \mu)]$ under relation $R * \texttt{Id}$. More formally, say that $C_1$ simulates $C_2$ under relation $R$ (written $R; C_2 \subseteq C_1; R$) when, for all $\sigma_1$, $\sigma_2$ such that $\sigma_1 R \sigma_2$:

1.) $\sigma_1 [\![C_1]\!] \texttt{fault} \iff \sigma_2 [\![C_2]\!] \texttt{fault}$, and

2.) $\neg\, \sigma_1 [\![C_1]\!] \texttt{fault} \implies (\forall \sigma_2' \,.\, \sigma_2 [\![C_2]\!] \sigma_2' \Rightarrow \exists \sigma_1' \,.\, \sigma_1 [\![C_1]\!] \sigma_1' \wedge \sigma_1' R \sigma_2')$

**Theorem 6.** *Suppose we have modules $(p, \eta)$ and $(q, \mu)$ with $\lfloor \eta \rfloor = \lfloor \mu \rfloor$ and a refinement relation $R$ as described above, such that $R * \texttt{Id}; \mu(f) \subseteq \eta(f); R * \texttt{Id}$ for all $f \in \lfloor \eta \rfloor$. Then, for any program $C$, $R * \texttt{Id}; C[(q, \mu)] \subseteq C[(p, \eta)]; R * \texttt{Id}$.*

While the full proof can be found in [13], we will semi-formally describe here the one case that highlights why behavior preservation eliminates the need for contents independence and growing relations: when $C$ is simply a client command $c$.

We wish to prove that $C[(p, \eta)]$ simulates $C[(q, \mu)]$, so suppose we have related states $\sigma_1$ and $\sigma_2$, and executing $c$ on $\sigma_2$ results in $\sigma_2'$. Since $\sigma_1$ and $\sigma_2$ are related by $R * \texttt{Id}$, we have that $\sigma_1 = \sigma_p \bullet \sigma_c$ and $\sigma_2 = \sigma_q \bullet \sigma_c$. We know that (1) $\sigma_q \bullet \sigma_c \xrightarrow{c} \sigma_2'$, (2) $c$ is local, and (3) $c$ runs safely on $\sigma_c$ because the client operation's execution must be independent of the module state $\sigma_q$ (i.e., there is no access violation); thus the backwards frame property tells us that $\sigma_2' = \sigma_q \bullet \sigma_c'$ and $\sigma_c \xrightarrow{c} \sigma_c'$. Now, if $c$ is behavior-preserving, then we can simply apply the forwards frame property, framing

on the state $\sigma_p$, to get that $\sigma_p \# \sigma'_c$ and $\sigma_p \bullet \sigma_c \xrightarrow{c} \sigma_p \bullet \sigma'_c$, completing the proof for this case. However, without behavior preservation, contents independence and growing relations are used in [18] to finish the proof. Specifically, because we know that $\sigma_q \bullet \sigma_c \xrightarrow{c} \sigma_q \bullet \sigma'_c$ and that $c$ runs safely on $\sigma_c$, contents independence says that $\sigma \bullet \sigma_c \xrightarrow{c} \sigma \bullet \sigma'_c$ for any $\sigma$ whose domain is a subset of the domain of $\sigma_q$. Therefore, we can choose $\sigma = \sigma_p$ because $R$ is a growing relation.

### 2.4.3 Relational Separation Logic

Relational Separation Logic [57] allows for simple reasoning about the relationship between two executions. Instead of deriving triples $\{P\}\, C\, \{Q\}$, a user of the logic derives *quadruples* of the form:

$$\{R\} \quad \begin{matrix} C \\ C' \end{matrix} \quad \{S\}$$

$R$ and $S$ are binary relations on states, rather than unary predicates. Semantically, a quadruple says that if we execute the two programs in states that are related by $R$, then both executions are safe, and any final states will be related by $S$. Furthermore, we want to be able to use this logic to prove program equivalence, so we also require that initial states related by $R$ have the same divergence behavior. Formally, we say that the above quadruple is valid if, for any states $\sigma_1$, $\sigma_2$, $\sigma'_1$, $\sigma'_2$:

$$1.)\ \sigma_1 R \sigma_2 \implies \neg\sigma_1[\![C]\!]\texttt{fault} \wedge \neg\sigma_2[\![C']\!]\texttt{fault}$$

$$2.)\ \sigma_1 R \sigma_2 \implies (\sigma_1[\![C]\!]\texttt{div} \iff \sigma_2[\![C']\!]\texttt{div})$$

$$3.)\ \sigma_1 R \sigma_2 \wedge \sigma_1[\![C]\!]\sigma'_1 \wedge \sigma_2[\![C']\!]\sigma'_2 \implies \sigma'_1 S \sigma'_2$$

Relational Separation Logic extends the separating conjunction to work for rela-

tions, breaking related states into disjoint, correspondingly-related pieces:

$$\sigma_1 (R * S) \sigma_2 \iff \exists\ \sigma_{1r},\ \sigma_{1s},\ \sigma_{2r},\ \sigma_{2s}\ .$$
$$\sigma_1 = \sigma_{1r} \bullet \sigma_{1s} \wedge \sigma_2 = \sigma_{2r} \bullet \sigma_{2s} \wedge \sigma_{1r} R \sigma_{2r} \wedge \sigma_{1s} S \sigma_{2s}$$

Just as Separation Logic has a frame rule for enabling local reasoning, Relational Separation Logic has a frame rule with the same purpose. This frame rule says that, given that we can derive the quadruple above involving $R$, $S$, $C$, and $C'$, we can also derive the following quadruple for any relation $T$:

$$\{R * T\} \begin{matrix} C \\ \\ C' \end{matrix} \{S * T\}$$

In [57], it is shown that the frame rule is sound when all programs are deterministic but it is unsound if nondeterministic programs are allowed, so this frame rule cannot be used when we have a nondeterministic memory allocator.

To deal with nondeterministic programs, a solution is proposed in [57], in which the interpretation of quadruples is strengthened. The new interpretation for a quadruple containing $R$, $S$, $C$, and $C'$ is that, for any $\sigma_1$, $\sigma_2$, $\sigma_1'$, $\sigma_2'$, $\sigma$, $\sigma'$:

1.) $\sigma_1 R \sigma_2 \implies \neg \sigma_1 [\![C]\!] \texttt{fault} \wedge \neg \sigma_2 [\![C']\!] \texttt{fault}$

2.) $\sigma_1 R \sigma_2 \wedge \sigma_1 \# \sigma \wedge \sigma_2 \# \sigma' \implies ((\sigma_1 \bullet \sigma) [\![C]\!] \texttt{div} \iff (\sigma_2 \bullet \sigma') [\![C']\!] \texttt{div})$

3.) $\sigma_1 R \sigma_2 \wedge \sigma_1 [\![C]\!] \sigma_1' \wedge \sigma_2 [\![C']\!] \sigma_2' \implies \sigma_1' S \sigma_2'$

Note that this interpretation is the same as before, except that the second property is strengthened to say that divergence behavior must be equivalent not only on the initial states, but also on any larger states. It can be shown that the frame rule becomes sound under this stronger interpretation of quadruples.

In our behavior-preserving setting, it is possible to use the simpler interpretation of quadruples without breaking soundness of the frame rule. We could prove this by directly proving frame rule soundness, but instead we will take a shorter route in which we show that, when actions are behavior-preserving, a quadruple is valid under the first interpretation above if and only if it is valid under the second interpretation — i.e., the two interpretations are the same in our setting.

Clearly, validity under the second interpretation implies validity under the first, since it is a direct strengthening. To prove the inverse, suppose we have a quadruple (involving $R$, $S$, $C$, and $C'$) that is valid under the first interpretation. Properties 1 and 3 of the second interpretation are identical to those of the first, so all we need to show is that Property 2 holds. Suppose that $\sigma_1 R \sigma_2$, $\sigma_1 \# \sigma$, and $\sigma_2 \# \sigma'$. By Property 1 of the first interpretation, we know that $\neg \sigma_1 [\![C]\!] \texttt{fault}$ and $\neg \sigma_2 [\![C']\!] \texttt{fault}$. Therefore, Termination Equivalence tells us that $\sigma_1 [\![C]\!] \texttt{div} \iff (\sigma_1 \bullet \sigma) [\![C]\!] \texttt{div}$, and that $\sigma_2 [\![C']\!] \texttt{div} \iff (\sigma_2 \bullet \sigma') [\![C']\!] \texttt{div}$. Furthermore, we know by Property 2 of the first interpretation that $\sigma_1 [\![C]\!] \texttt{div} \iff \sigma_2 [\![C']\!] \texttt{div}$. Hence we obtain our desired result.

**Note** In case the reader is curious, the reason that the frame rule under the first interpretation is sound when all programs are deterministic is simply that determinism (along with standard locality) implies Termination Equivalence. To see this, we only need to check the forwards direction, since standard locality requires the backwards one. Consider a situation where $\sigma_0 \, [A] \, \texttt{div}$, $\sigma_0 \# \sigma_1$, and $\neg \sigma_0 \, [A] \, \texttt{fault}$. By Safety Monotonicity, we have $\neg (\sigma_0 \bullet \sigma_1) \, [A] \, \texttt{fault}$. Furthermore, suppose there is some $\sigma$ such that $(\sigma_0 \bullet \sigma_1) \, [A] \, \sigma$. Then by the Backwards Frame Property, we have $\sigma = \sigma_0' \bullet \sigma_1$ and $\sigma_0 \, [A] \, \sigma_0'$. But we already know that $\sigma_0 \, [A] \, \texttt{div}$, so this contradicts the fact that $A$ is deterministic. Therefore, $A$ does not relate $\sigma_0 \bullet \sigma_1$ to $\texttt{fault}$ or to any state $\sigma$. Since $A$ is required to be total, we conclude that $(\sigma_0 \bullet \sigma_1) \, [A] \, \texttt{div}$.

### 2.4.4 Finite Memory

Since standard locality allows the program state to increase during execution, it does not play nicely with a model in which memory is finite. Consider any command that grows the program state in some way. Such a command is safe on the empty state but, if we extend this empty state to the larger state consisting of all available memory, then the command becomes unsafe. Hence such a command violates Safety Monotonicity.

There is one commonly-used solution for supporting finite memory without enforcing behavior preservation: say that, instead of faulting on the state consisting of all of memory, a state-growing command diverges. Furthermore, to satisfy Termination Monotonicity, we also need to allow the command to diverge on *any* state. The downside of this solution, therefore, is that it is only reasonable when we are not interested in the termination behavior of programs.

When behavior preservation is enforced, we no longer have any issues with finite memory models because program state cannot increase during execution. The initial state is obviously contained within the finite memory, so all states reachable through execution must also be contained within memory.

### 2.4.5 Security

Although we did not have security in mind at the time of our APLAS publication, it turns out that behavior preservation has some close ties with security reasoning. In particular, both behavior preservation and noninterference are properties regarding equality of a program's behaviors. The former relates executions on a small state with those on a larger state, while the latter relates executions on some state with those for which the values of high-security data have been altered.

More formally, let the notation $[\![C]\!](\sigma)$ represent the set of possible behaviors when executing $C$ on initial state $\sigma$ (in the context of earlier discussion, this behavior set

could include final states or the special behaviors `fault` or `div`). Suppose that our program state can be represented as $\sigma_H \bullet \sigma_L$, where $\sigma_H$ contains high-security data and $\sigma_L$ contains low-security data. Then classical noninterference says that changing the values within $\sigma_H$ will not influence the possible behaviors: $[\![C]\!](\sigma_H \bullet \sigma_L) = [\![C]\!](\sigma'_H \bullet \sigma_L)$ (where $\sigma_H$ and $\sigma'_H$ have equal domains). Consider what happens if we prove that $C$ satisfies this property, and then execute $C$ in a larger context with extra unused resources. We will now have behavior sets $[\![C]\!](\sigma \bullet \sigma_H \bullet \sigma_L)$ and $[\![C]\!](\sigma \bullet \sigma'_H \bullet \sigma_L)$; standard locality will not guarantee these sets are equal. This means that seemingly-unused resources could maliciously reveal information about the secrets in $\sigma_H$! Thus standard local reasoning is fundamentally incompatible with noninterference.

If we enforce behavior preservation, however, then this incompatibility disappears. Behavior preservation guarantees that a program will exhibit exactly the same set of behaviors when extra resources are added to state, so the two sets $[\![C]\!](\sigma \bullet \sigma_H \bullet \sigma_L)$ and $[\![C]\!](\sigma \bullet \sigma'_H \bullet \sigma_L)$ will be equal in the example above. In Chapter 3, we will present a security-aware program logic that guarantees a certain notion of noninterference. This program logic is based on Separation Logic and thus has a frame rule for local reasoning. Given the insight above, it should not be surprising that we found behavior preservation to be extremely important for soundness of the security-aware program logic. Indeed, inspecting the Coq proofs of that program logic at the dissertation's companion website [11], one will notice the lemmas `hstep_ff` and `lstep_ff` are exactly the Forwards Frame property presented in this chapter; one will also notice that these lemmas are used in the proof of soundness of the frame rule.

# Chapter 3

# Security via Program Logic

Our work published in POST 2014 [14] presents a program logic for proving that code is secure with respect to a general, expressive security policy. This chapter will present the formalization of this program logic, and will show how a particular nontrivial program is verified secure using the logic. However, after considering the implications of using the logic, as well as its relative incompleteness, we will arrive at a crucial conclusion: it would be far more ideal to verify security in a way that is not specific to a particular program logic. Later chapters then demonstrate that this ideal is in fact achievable. Hence this chapter should be taken with a grain of salt; it is important for developing a solid understanding of formal security verification (especially with respect to policy specification), but this dissertation ultimately advocates *against* relying upon any specific program logic such as the one presented here.

## 3.1 Program Logic Overview

The ultimate goal of our security-aware program logic is to verify interesting security policies over low-level systems code. For convenience we will use a toy C-like language here that supports heap manipulation and pointer arithmetic; it should generally be clear how to extend the toy language with other C features. The language will not

support unstructured control flow (e.g., "goto"), however, so it is not designed for directly reasoning about assembly code. This is one reason why we will eventually decide to move away from a specific program logic and instead design a more general methodology that works for any code written in any language.

Our program logic will perform security reasoning by directly modeling dynamic label tainting (as described in Section 1.3). For simplicity, we will use a label lattice consisting only of the two labels `Lo` and `Hi`, with $\texttt{Lo} \sqsubseteq \texttt{Hi}$. This lattice can easily be mapped to a more intricate one, however: for a given observer principal $p$ with label $L_p$, the `Lo` label represents all labels in the more intricate lattice which are equal to or below $L_p$, while `Hi` represents all other labels.

Label propagation is done in a mostly obvious way. All values in the variable store and the global memory heap are associated with a label (`Lo` or `Hi`). If we have a direct assignment such as $x := y$, then the label of $y$'s data propagates into $x$ along with the data itself. We compute the composite label of an expression such as $2 * x + z$ to be the least upper bound of the labels of its constituent parts (for the two-element lattice of `Lo` and `Hi`, this will be `Lo` if and only if each constituent label is `Lo`). For the heap-read command $x := [E]$, we must propagate both the label of $E$ and the label of the data located at heap address $E$ into $x$. In other words, if we read some low-security data from the heap using a high-security pointer, the result must be tainted as high security in order for our information flow tracking to be accurate. Similarly, the heap-write command $[E] := E'$ must propagate both the label of $E'$ and the label of pointer $E$ into the location $E$ in the heap. As a general rule for any of these atomic commands, we compute the composite label of the entire read-set, and propagate that into all locations in the write-set.

### 3.1.1 Security Formulation

Our security guarantee is based on pure noninterference, which says that the initial values of `Hi` data have no effect on the "observable behavior" of a program's execution. We choose to define observable behavior in terms of a special output channel. We include an output command in our language, and an execution's observable behavior is defined to be exactly the sequence of values that the execution outputs.

The standard way to express this noninterference property formally is in terms of two executions: a program is deemed to be noninterfering if two executions of the program from *observably equivalent* initial states always yield identical outputs. Two states are defined to be observably equivalent when only their high-security values differ. Thus this property describes what one would expect: changing the value of any high-security data in the initial state will cause no change in the program's output. As hinted at during the examples of Section 1.3, we will actually use a weaker version of noninterference to allow for security policies with certain well-specified flows from high security to low (e.g., declassification).

We accomplish this weakening of noninterference by requiring a precondition to hold on the initial state of an execution. That is, we alter the noninterference property to say that two executions will yield identical outputs if they start from two observably equivalent states that both satisfy some state predicate $P$. This weakening is interesting for two reasons. First, it provides a clean interface between information-flow reasoning and Hoare Logic (a program logic that derives pre/postconditions as state predicates). Second, this weaker property describes a certain level of dependency between high-security inputs and low-security outputs, rather than the complete independence of pure noninterference. This is key for allowing interesting, real-world security policies. To better understand this property, let us revisit the examples of Section 1.3.

**Public Parity** Suppose we have a variable $x$ that contains Alice's secret value $v$, and Alice wishes to only publicly release the parity of $v$. We can prove that some program $C$ obeys this security policy by verifying it using our program logic, with respect to a precondition $P$ saying "$x$ contains high data, $y$ contains low data, and $y = x\%2$". Our weakened noninterference property described above says that if we have an execution from some initial state satisfying this $P$, then changing the value of $x$ will not affect the program's output as long as the new state also satisfies $P$. Since $y$'s value is $v\%2$ and is unchanged in the two executions, this means that as long as we change $x$'s value to something *with the same parity*, the output will be unchanged. Indeed, this is exactly the desired property for enforcing Alice's security policy. In this way, the precondition $P$ used for verifying $C$ serves as a formal description of the security policy.

**Public Average** Suppose we have $n$ salaries stored in variables $x_1$ through $x_n$, and we are only willing to release their average as public. This is similar to the previous example, except that we now have multiple secrets. The precondition $P$ says that $x_1$ through $x_n$ all contain `Hi` data, $a$ contains `Lo` data, and $a = (\Sigma\ x_i)/n$. In this situation, noninterference of a verified program $C$ says that we can change any of the values contained within the *set* of variables $\{x_1, ..., x_n\}$ in any way, and $C$'s output will be unaffected as long as the average of all the values is unchanged.

**Shared Calendar** Suppose Alice has a calendar represented as an array of $n$ time slots at heap locations $l$ through $l + n - 1$. Each time slot may either contain a value of 0 if Alice has not scheduled an event at that time, or some nonzero value encoding the details of an event. The precondition $P$ says that, for each time slot $t$ in the calendar, either the value at $t$ is 0 and the label is `Lo`, or the value is not 0 and the label is `Hi`. This is an example of what we call a *conditional label*: a label that is dependent on values within the program state (notice that the previous examples

did not use conditional labels). Interpreting this precondition as our security policy, noninterference of a program $C$ says that changing any *nonzero* (i.e., high security) values within the calendar will not have any effect on the output produced by $C$ as long as $P$ still holds; in order for $P$ to still hold, we must be changing these values into other *nonzero* values. Hence we get exactly the desired policy: the observable behavior is independent from the details of any scheduled events (i.e., we cannot distinguish one nonzero value from another, but we *can* determine which time slots contain 0 and which do not). Conditional labels are a novelty of our program logic; they allow for highly expressive security policies, and we will show below how they can be a useful and powerful tool in conducting the security verification.

**Dynamic Label Tainting** We can trivially represent standard dynamic label tainting, since our program logic is based upon directly modeling labels and their propagation throughout an execution. It is important to note, however, that this modeling of labels is purely logical. The labels are there to help specify the security policy and to help conduct the security verification; when the actual program executes, no labels exist within the machine state, and no computations occur for propagating labels. We enforce this formally by defining two different machine operational semantics. The low-level, "true" semantics is completely unaware of security labels, while the higher level instrumented semantics deals with labels and propagation. We prove standard theorems to relate the two semantics together, so that our final end-to-end security guarantee applies to the true semantics of execution.

## 3.2 Language and Semantics

We can now get into the formal definitions of our program logic. The entire program logic, along with its theorems, have been encoded in the Coq proof assistant language and can be found at the companion website [11]. Our simple, C-like toy programming

language is defined as follows:

$$
\begin{aligned}
\text{(Exp)} \quad & E ::= x \mid c \mid E + E \mid \cdots \\
\text{(BExp)} \quad & B ::= \texttt{false} \mid E = E \mid B \wedge B \mid \cdots \\
\text{(Cmd)} \quad & C ::= \texttt{skip} \mid \texttt{output}\, E \mid x := E \mid x := [E] \mid [E] := E \\
& \qquad \mid C; C \mid \texttt{if}\, B\, \texttt{then}\, C\, \texttt{else}\, C \mid \texttt{while}\, B\, \texttt{do}\, C
\end{aligned}
$$

Valid code includes variable assignment, heap load/store, if statements, while loops, and output. Our model of a program state, consisting of a variable store and a heap, is given by:

$$
\begin{aligned}
\text{(Lbl)} \quad & L ::= \texttt{Lo} \mid \texttt{Hi} \\
\text{(Val)} \quad & V ::= \mathbb{Z} \times \text{Lbl} \\
\text{(Store)} \quad & s ::= \text{Var} \to \text{option Val} \\
\text{(Heap)} \quad & h ::= \mathbb{N} \to \text{option Val} \\
\text{(State)} \quad & \sigma ::= \text{Store} \times \text{Heap}
\end{aligned}
$$

Given a variable store $s$, we define a denotational semantics $[\![E]\!]s$ that evaluates an expression to a pair of integer and label, with the label being the least upper bound of the labels of the constituent parts. The denotation of an expression also may evaluate to `None`, indicating that the program state does not contain the necessary resources to evaluate. We have a similar denotational semantics for boolean expressions. The formal definitions of these semantics are omitted here as they are standard and straightforward. Note that we will sometimes write $[\![E]\!]\sigma$ as shorthand for $[\![E]\!]$ applied to the store of state $\sigma$.

Figure 3.1 defines our instrumented operational semantics. The semantics is security-aware, meaning that it keeps track of security labels on data and propagates these labels throughout execution in order to track which values might have been influenced by some high-security data. The semantics operates on machine con-

$$\frac{[\![E]\!]s = \texttt{Some}\ (n, l)}{\langle (s,h),\ x := E,\ K\rangle \underset{l'}{\longrightarrow} \langle (s[x \mapsto (n, l \sqcup l')], h),\ \texttt{skip},\ K\rangle}\ \text{(ASSGN)}$$

$$\frac{[\![E]\!]s = \texttt{Some}\ (n_1, l_1) \qquad h(n_1) = \texttt{Some}\ (n_2, l_2)}{\langle (s,h),\ x := [E],\ K\rangle \underset{l'}{\longrightarrow} \langle (s[x \mapsto (n_2, l_1 \sqcup l_2 \sqcup l')], h),\ \texttt{skip},\ K\rangle}\ \text{(READ)}$$

$$\frac{[\![E]\!]s = \texttt{Some}\ (n_1, l_1) \qquad h(n_1) \neq \texttt{None} \qquad [\![E']\!]s = \texttt{Some}\ (n_2, l_2)}{\langle (s,h),\ [E] := E',\ K\rangle \underset{l'}{\longrightarrow} \langle (s, h[n_1 \mapsto (n_2, l_1 \sqcup l_2 \sqcup l')]),\ \texttt{skip},\ K\rangle}\ \text{(WRITE)}$$

$$\frac{[\![E]\!]\sigma = \texttt{Some}\ (n, \texttt{Lo})}{\langle \sigma,\ \texttt{output}\ E,\ K\rangle \xrightarrow[\texttt{Lo}]{[n]} \langle \sigma,\ \texttt{skip},\ K\rangle}\ \text{(OUTPUT)}$$

$$\frac{[\![B]\!]\sigma = \texttt{Some}\ (\texttt{true}, l) \qquad l \sqsubseteq l'}{\langle \sigma,\ \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2,\ K\rangle \underset{l'}{\longrightarrow} \langle \sigma,\ C_1,\ K\rangle}\ \text{(IF-TRUE)}$$

$$\frac{[\![B]\!]\sigma = \texttt{Some}\ (\texttt{false}, l) \qquad l \sqsubseteq l'}{\langle \sigma,\ \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2,\ K\rangle \underset{l'}{\longrightarrow} \langle \sigma,\ C_2,\ K\rangle}\ \text{(IF-FALSE)}$$

$$\frac{\begin{array}{c}[\![B]\!]\sigma = \texttt{Some}\ (\_, \texttt{Hi}) \\ \langle \texttt{mark\_vars}(\sigma, \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2),\ \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2,\ []\rangle \underset{\texttt{Hi}}{\longrightarrow}_n \langle \sigma',\ \texttt{skip},\ []\rangle\end{array}}{\langle \sigma,\ \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2,\ K\rangle \underset{\texttt{Lo}}{\longrightarrow} \langle \sigma',\ \texttt{skip},\ K\rangle}\ \text{(IF-HI)}$$

$$\frac{[\![B]\!]\sigma = \texttt{Some}\ (\texttt{true}, l) \qquad l \sqsubseteq l'}{\langle \sigma,\ \texttt{while}\ B\ \texttt{do}\ C,\ K\rangle \underset{l'}{\longrightarrow} \langle \sigma,\ C; \texttt{while}\ B\ \texttt{do}\ C,\ K\rangle}\ \text{(WHILE-TRUE)}$$

$$\frac{[\![B]\!]\sigma = \texttt{Some}\ (\texttt{false}, l) \qquad l \sqsubseteq l'}{\langle \sigma,\ \texttt{while}\ B\ \texttt{do}\ C,\ K\rangle \underset{l'}{\longrightarrow} \langle \sigma,\ \texttt{skip},\ K\rangle}\ \text{(WHILE-FALSE)}$$

$$\frac{\begin{array}{c}[\![B]\!]\sigma = \texttt{Some}\ (\_, \texttt{Hi}) \\ \langle \texttt{mark\_vars}(\sigma, \texttt{while}\ B\ \texttt{do}\ C),\ \texttt{while}\ B\ \texttt{do}\ C,\ []\rangle \underset{\texttt{Hi}}{\longrightarrow}_n \langle \sigma',\ \texttt{skip},\ []\rangle\end{array}}{\langle \sigma,\ \texttt{while}\ B\ \texttt{do}\ C,\ K\rangle \underset{\texttt{Lo}}{\longrightarrow} \langle \sigma',\ \texttt{skip},\ K\rangle}\ \text{(WHILE-HI)}$$

$$\frac{}{\langle \sigma,\ C_1; C_2,\ K\rangle \underset{l}{\longrightarrow} \langle \sigma,\ C_1,\ C_2 :: K\rangle}\ \text{(SEQ)} \qquad \frac{}{\langle \sigma,\ \texttt{skip},\ C :: K\rangle \underset{l}{\longrightarrow} \langle \sigma,\ C,\ K\rangle}\ \text{(SKIP)}$$

$$\frac{}{\langle \sigma,\ C,\ K\rangle \underset{l}{\longrightarrow}_0 \langle \sigma,\ C,\ K\rangle}\ \text{(ZERO)}$$

$$\frac{\langle \sigma,\ C,\ K\rangle \xrightarrow[l]{o} \langle \sigma',\ C',\ K'\rangle \qquad \langle \sigma',\ C',\ K'\rangle \xrightarrow[l]{o'}_n \langle \sigma'',\ C'',\ K''\rangle \qquad n > 0}{\langle \sigma,\ C,\ K\rangle \xrightarrow[l]{o+o'}_{n+1} \langle \sigma'',\ C'',\ K''\rangle}\ \text{(SUCC)}$$

Figure 3.1: Security-Aware Operational Semantics

figurations, which consist of program state, code, and a list of commands called the continuation stack (we use a continuation-stack approach solely for the purpose of simplifying some proofs). The transition arrow of the semantics is annotated with a *program counter label*, which is a standard IFC construct used to keep track of information flow resulting from the control flow of the execution. Whenever an execution enters a conditional construct, it raises the pc label by the label of the boolean expression evaluated; the pc label then taints any assignments made within the conditional construct (see variable $l'$ in the (ASSGN), (READ), and (WRITE) rules). The transition arrow is also annotated with a list of outputs (equal to the empty list when not explicitly written) and the number of steps (equal to 1 when not explicitly written). We sometimes annotate the arrow with an asterisk ($\longrightarrow_*$) to indicate zero or more steps.

Two of the rules for conditional constructs make use of a function called `mark_vars`. `mark_vars`$(\sigma, C)$ alters $\sigma$ by setting the label of each variable in `modifies`$(C)$ to `Hi`, where `modifies`$(C)$ is a syntactic function returning an overapproximation of the store variables that may be modified by $C$. Thus, whenever we raise the pc label to `Hi`, our semantics taints all store variables that appear on the left-hand side of an assignment or heap-read command within the conditional construct, even if some of those commands do not actually get executed. Note that regardless of which branch of an if statement is taken, the semantics taints all the variables in *both* branches. This is required for noninterference, due to the well-known fact that the *lack* of assignment in a branch of an if statement can leak information about the branching expression. Consider, for example, the following program:

```
1    y := 1;
2    if (x = 0) then y := 0 else skip;
3    if (y = 0) then skip else output 1;
```

Suppose $x$ contains `Hi` data initially, while $y$ contains `Lo` data. If $x$ is 0, then $y$ will

be assigned 0 at line 2 and tainted with a `Hi` label (by the pc label). Then nothing happens at line 3, and the program produces no output. If $x$ is nonzero, however, nothing happens at line 2, so $y$ still has a `Lo` label at line 3. Thus the output command at line 3 executes without issue. Therefore the output of this program depends on the `Hi` data in $x$, even though our instrumented semantics executes safely. We choose to resolve this issue by using the `mark_vars` function in the semantics. Then $y$ will be tainted at line 2 regardless of the value of $x$, and so the semantics will get stuck at line 3 when $x$ is nonzero. In other words, we would only be able to verify this program with a precondition saying that $x = 0$ — the program is indeed noninterfering with respect to this degenerate security policy.

The operational semantics presented here is mixed-step and manipulates security labels directly. As mentioned above, we need to relate it to a more standard semantics that does not make use of security labels. A standard, single-step semantics is defined in Figure 3.2. This semantics operates on states without labels, and it does not use continuation stacks. Given a state $\sigma$ with labels, we write $\bar{\sigma}$ to represent the same state with all labels erased from both the store and heap. We will also use $\tau$ to range over states without labels. Then the following two theorems hold:

**Theorem 7.** *Suppose* $\langle \sigma, C, [] \rangle \xrightarrow[\texttt{Lo}]{o}{}_* \langle \sigma', \texttt{skip}, [] \rangle$ *in the instrumented semantics. Then there exists* $\tau$ *such that* $\langle \bar{\sigma}, C \rangle \xrightarrow{o}{}_* \langle \tau, \texttt{skip} \rangle$ *in the standard semantics.*

**Theorem 8.** *Suppose* $\langle \bar{\sigma}, C \rangle \xrightarrow{o}{}_* \langle \tau, \texttt{skip} \rangle$ *in the standard semantics, and suppose* $\langle \sigma, C, [] \rangle$ *never gets stuck when executed in the instrumented semantics. Then there exists* $\sigma'$ *such that* $\bar{\sigma}' = \tau$ *and* $\langle \sigma, C, [] \rangle \xrightarrow[\texttt{Lo}]{o}{}_* \langle \sigma', \texttt{skip}, [] \rangle$ *in the instrumented semantics.*

These theorems together guarantee that the two semantics produce identical observable behaviors (outputs) on terminating executions, as long as the instrumented semantics does not get stuck. Our program logic will of course guarantee that the

$$\frac{[\![E]\!]s = \texttt{Some}\ n}{\langle (s,h),\ x := E\rangle \longrightarrow \langle (s[x \mapsto n], h),\ \texttt{skip}\rangle}\ (\text{ASSGN})$$

$$\frac{[\![E]\!]s = \texttt{Some}\ n_1 \qquad h(n_1) = \texttt{Some}\ n_2}{\langle (s,h),\ x := [E]\rangle \longrightarrow \langle (s[x \mapsto n_2], h),\ \texttt{skip}\rangle}\ (\text{READ})$$

$$\frac{[\![E]\!]s = \texttt{Some}\ n_1 \qquad h(n_1) \neq \texttt{None} \qquad [\![E']\!]s = \texttt{Some}\ n_2}{\langle (s,h),\ [E] := E'\rangle \longrightarrow \langle (s, h[n_1 \mapsto n_2]),\ \texttt{skip}\rangle}\ (\text{WRITE})$$

$$\frac{[\![E]\!]\tau = \texttt{Some}\ n}{\langle \tau,\ \texttt{output}\ E\rangle \xrightarrow{[n]} \langle \tau,\ \texttt{skip}\rangle}\ (\text{OUTPUT})$$

$$\frac{[\![B]\!]\tau = \texttt{Some true}}{\langle \tau,\ \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2\rangle \longrightarrow \langle \tau,\ C_1\rangle}\ (\text{IF-TRUE})$$

$$\frac{[\![B]\!]\tau = \texttt{Some false}}{\langle \tau,\ \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2\rangle \longrightarrow \langle \tau,\ C_2\rangle}\ (\text{IF-FALSE})$$

$$\frac{[\![B]\!]\tau = \texttt{Some true}}{\langle \tau,\ \texttt{while}\ B\ \texttt{do}\ C\rangle \longrightarrow \langle \tau,\ C; \texttt{while}\ B\ \texttt{do}\ C\rangle}\ (\text{WHILE-TRUE})$$

$$\frac{[\![B]\!]\tau = \texttt{Some false}}{\langle \tau,\ \texttt{while}\ B\ \texttt{do}\ C\rangle \longrightarrow \langle \tau,\ \texttt{skip}\rangle}\ (\text{WHILE-FALSE})$$

$$\frac{\langle \tau,\ C_1\rangle \xrightarrow{o} \langle \tau',\ C_1'\rangle}{\langle \tau,\ C_1; C_2\rangle \xrightarrow{o} \langle \tau',\ C_1'; C_2\rangle}\ (\text{SEQ}) \qquad \frac{}{\langle \tau,\ \texttt{skip}; C\rangle \longrightarrow \langle \tau,\ C\rangle}\ (\text{SKIP})$$

$$\frac{}{\langle \tau,\ C\rangle \longrightarrow_0 \langle \tau,\ C\rangle}\ (\text{ZERO})$$

$$\frac{\langle \tau,\ C\rangle \xrightarrow{o} \langle \tau',\ C'\rangle \qquad \langle \tau',\ C'\rangle \xrightarrow{o'}_n \langle \tau'',\ C''\rangle \qquad n > 0}{\langle \tau,\ C\rangle \xrightarrow{o +\!+ o'}_{n+1} \langle \tau'',\ C''\rangle}\ (\text{SUCC})$$

Figure 3.2: Standard Operational Semantics

$$
\begin{array}{c}
P, Q \ ::= \ \texttt{emp} \mid E \mapsto \_ \mid E \mapsto (n, l) \mid B \mid x.\texttt{lbl} = l \mid x.\texttt{lbl} \sqsubseteq l \\
\mid \texttt{lbl}(E) = l \mid \exists X \,.\, P \mid P \wedge Q \mid P \vee Q \mid P * Q
\end{array}
$$

$$
\begin{array}{rcl}
[\![P]\!] & : & \mathbb{P}(\texttt{state}) \\
(s,h) \in [\![\texttt{emp}]\!] & \iff & h = \emptyset \\
(s,h) \in [\![E \mapsto \_]\!] & \iff & \exists a, n, l \,.\, [\![E]\!]s = \texttt{Some}\ a \wedge h = [a \mapsto (n,l)] \\
(s,h) \in [\![E \mapsto (n,l)]\!] & \iff & \exists a \,.\, [\![E]\!]s = \texttt{Some}\ a \wedge h = [a \mapsto (n,l)] \\
(s,h) \in [\![B]\!] & \iff & [\![B]\!]s = \texttt{Some true} \\
(s,h) \in [\![x.\texttt{lbl} = l]\!] & \iff & \exists n \,.\, s(x) = \texttt{Some}\ (n,l) \\
(s,h) \in [\![x.\texttt{lbl} \sqsubseteq l]\!] & \iff & \exists n, l' \,.\, s(x) = \texttt{Some}\ (n,l') \text{ and } l' \sqsubseteq l \\
(s,h) \in [\![\texttt{lbl}(E) = l]\!] & \iff & \bigsqcup_{x \in \texttt{vars}(E)} \mathrm{snd}(s(x)) = l \\
(s,h) \in [\![\exists X \,.\, P]\!] & \iff & \exists v \in \mathbb{Z} + \mathrm{Lbl} \,.\, (s,h) \in [\![P[v/X]]\!] \\
(s,h) \in [\![P \wedge Q]\!] & \iff & (s,h) \in [\![P]\!] \cap [\![Q]\!] \\
(s,h) \in [\![P \vee Q]\!] & \iff & (s,h) \in [\![P]\!] \cup [\![Q]\!] \\
(s,h) \in [\![P * Q]\!] & \iff & \begin{pmatrix} \exists h_0, h_1 \,.\, h_0 \uplus h_1 = h \\ \wedge\, (s, h_0) \in [\![P]\!] \\ \wedge\, (s, h_1) \in [\![Q]\!] \end{pmatrix}
\end{array}
$$

Figure 3.3: Assertion Syntax and Semantics

instrumented semantics does not get stuck in any execution satisfying the precondition.

## 3.3 The Program Logic

We next present the formal inference rules used for verifying the security of a program. A logic judgment takes the form $l \vdash \{P\}\, C\, \{Q\}$. $P$ and $Q$ are the pre- and postconditions, $C$ is the program to be executed, and $l$ is the pc label under which the program is verified. $P$ and $Q$ are *state assertions*, whose syntax and semantics are given in Figure 3.3.

**Note** We allow assertions to contain logical variables, but we elide the details here to avoid complicating the presentation. In Figure 3.3, we claim that the type of $[\![P]\!]$ is a set of states — in reality, the type is a function from logical variable environments to sets of states. In an assertion like $E \mapsto (n, l)$, the $n$ and $l$ may be logical variables rather than constants.

**Definition 1** (Sound judgment)**.** *We say that a judgment* $l \vdash \{P\}\, C\, \{Q\}$ *is* sound *if, for any state* $\sigma \in [\![P]\!]$*, the following two properties hold:*

1. *The operational semantics cannot get stuck when executed from initial configuration* $\langle \sigma, C, [] \rangle$ *with pc label* $l$*.*

2. *If the operational semantics executes from initial configuration* $\langle \sigma, C, [] \rangle$ *with pc label* $l$ *and terminates at state* $\sigma'$*, then* $\sigma' \in [\![Q]\!]$*.*

Selected inference rules for our logic are shown in Figure 3.4. The rules make use of two auxiliary syntactic functions, $P \backslash x$ and $P \backslash x.\texttt{lbl}$ (read the backslash operator as "delete"). $P \backslash x$ replaces any atomic assertions within $P$ referring to $x$ by the assertion `true`. Similarly, $P \backslash x.\texttt{lbl}$ replaces atomic assertions referring to $x.\texttt{lbl}$ by `true`. We also sometimes abuse notation and write $P \backslash S$ or $P \backslash S.\texttt{lbl}$, where $S$ is a set of variables, to indicate the iterative folding of these functions over the set $S$. The important fact about these auxiliary functions is that, if $P$ holds on some state and we perform an assigment into $x$, then $P \backslash x$ will hold on the resulting state. Furthermore, if we change only the label of $x$ without touching its data (this is done by the `mark_vars` function described in Section 3.2), then $P \backslash x.\texttt{lbl}$ will hold on the resulting state.

Here are a few interesting points to note about these inference rules:

- While the rules shown here mostly involve detailed reasoning about label propagation, we can also prove the soundness of simpler versions of the rules that

$$\texttt{mark\_vars}(P, S, l, l') \triangleq \begin{cases} P & , \quad \text{if } l \sqsubseteq l' \\ P \backslash S.\texttt{lbl} \wedge \left( \bigwedge_{x \in S} l \sqcup l' \sqsubseteq x.\texttt{lbl} \right) & , \quad \text{otherwise} \end{cases}$$

$$\frac{}{l \vdash \{P\}\, \texttt{skip}\, \{P\}}\ (\text{SKIP}) \qquad \frac{P \Rightarrow \texttt{lbl}(E) = \texttt{Lo}}{\texttt{Lo} \vdash \{P\}\, \texttt{output}\, E\, \{P\}}\ (\text{OUTPUT})$$

$$\frac{P \Rightarrow \texttt{lbl}(E) = l}{l' \vdash \{P\}\, x := E\, \{(P \backslash x)[E/x] \wedge x.\texttt{lbl} = l \sqcup l'\}}\ (\text{ASSIGN})$$

$$\frac{P \Rightarrow \texttt{lbl}(E) = l_1 \qquad P \Rightarrow E \mapsto (n, l_2)}{l \vdash \{P\}\, x := [E]\, \{P \backslash x \wedge x = n \wedge x.\texttt{lbl} = l_1 \sqcup l_2 \sqcup l\}}\ (\text{READ})$$

$$\frac{P \Rightarrow \texttt{lbl}(E) = l_1 \qquad P \Rightarrow \texttt{lbl}(E') = l_2 \qquad P \Rightarrow E \mapsto \_}{l \vdash \{P\}\, [E] := E'\, \{P \wedge \exists n\, .\, E \mapsto (n, l_1 \sqcup l_2 \sqcup l) \wedge E' = n\}}\ (\text{WRITE})$$

$$\frac{\begin{array}{c} P \Rightarrow B \vee \neg B \qquad B \wedge P \Rightarrow \texttt{lbl}(B) = l_t \\ \neg B \wedge P \Rightarrow \texttt{lbl}(B) = l_f \qquad S = \texttt{modifies}(\texttt{if}\, B\, \texttt{then}\, C_1\, \texttt{else}\, C_2) \\ l_t \sqcup l' \vdash \{B \wedge \texttt{mark\_vars}(P, S, l_t, l')\}\, C_1\, \{Q\} \\ l_f \sqcup l' \vdash \{\neg B \wedge \texttt{mark\_vars}(P, S, l_f, l')\}\, C_2\, \{Q\} \end{array}}{l' \vdash \{P\}\, \texttt{if}\, B\, \texttt{then}\, C_1\, \texttt{else}\, C_2\, \{Q\}}\ (\text{IF})$$

$$\frac{\begin{array}{c} P \Rightarrow \texttt{lbl}(B) = l \qquad S = \texttt{modifies}(\texttt{while}\, B\, \texttt{do}\, C) \\ l \sqcup l' \vdash \{B \wedge \texttt{mark\_vars}(P, S, l, l')\}\, C\, \{\texttt{mark\_vars}(P, S, l, l')\} \end{array}}{l' \vdash \{P\}\, \texttt{while}\, B\, \texttt{do}\, C\, \{\neg B \wedge \texttt{mark\_vars}(P, S, l, l')\}}\ (\text{WHILE})$$

$$\frac{l \vdash \{P\}\, C_1\, \{Q\} \qquad l \vdash \{Q\}\, C_2\, \{R\}}{l \vdash \{P\}\, C_1; C_2\, \{R\}}\ (\text{SEQ})$$

$$\frac{P' \Rightarrow P \qquad Q \Rightarrow Q' \qquad l \vdash \{P\}\, C\, \{Q\}}{l \vdash \{P'\}\, C\, \{Q'\}}\ (\text{CONSEQ})$$

$$\frac{l \vdash \{P_1\}\, C\, \{Q_1\} \qquad l \vdash \{P_2\}\, C\, \{Q_2\}}{l \vdash \{P_1 \wedge P_2\}\, C\, \{Q_1 \wedge Q_2\}}\ (\text{CONJ})$$

$$\frac{l \vdash \{P\}\, C\, \{Q\} \qquad \texttt{modifies}(C) \cap \texttt{vars}(R) = \emptyset}{l \vdash \{P * R\}\, C\, \{Q * R\}}\ (\text{FRAME})$$

Figure 3.4: Selected Inference Rules for the Logic

do not reason about labels and, consequentially, do not have any label-related proof obligations.

- The (IF) and (WHILE) rules may look rather complex, but almost all of that is just describing how to reason about the `mark_vars` function that gets applied at the beginning of a conditional construct when the pc label increases.

- An additional complexity present in the (IF) rule involves the labels $l_t$ and $l_f$. In fact, these labels describe a novel and interesting feature of our system: when verifying an if statement, it might be possible to reason that the pc label gets raised by $l_t$ in one branch and by $l_f$ in the other, based on the fact that $B$ holds in one branch but not in the other. This is interesting if $l_t$ and $l_f$ are different labels. In every other static-analysis IFC system we are aware of, a particular pc label must be determined at the entrance to the conditional, and this pc label will propagate to both branches. We will see an example program shortly that illustrates this novelty.

Given our logic inference rules, the following theorem holds:

**Theorem 9** (Soundness)**.** *If* $l \vdash \{P\}\, C\, \{Q\}$ *is derivable according to our inference rules, then it is a* sound judgment*, as defined in Definition 1.*

## 3.4 Example: Alice's Calendar

Before we delve into the noninterference guarantee provided by the inference rules, let us first see how the inference rules can be used to verify an interesting example. Consider the calendar example discussed previously. Figure 3.5 shows a program that we would like to prove is secure with respect to Alice's policy. Suppose Alice owns a calendar with 64 time slots beginning at some location designated by constant $A$. Each time slot is either 0 if she is free at that time, or some nonzero value representing

```
1  i := 0;
2  while (i < 64) do
3      x := [A+i];
4      if (x = 0)
5          then
6              output i
7          else
8              skip;
9      i := i+1
```

Figure 3.5: Example: Alice's Private Calendar

an event if she is busy. Alice does not want to reveal any details about her scheduled events; this policy still allows for others to schedule a meeting time with her, as they can determine when she is available. Indeed, the example program shown here simply prints out all free time slots.

Figure 3.6 gives an overview of the verification, omitting a few trivial details. In between each line of code, we show the current pc label and a state predicate that currently holds. The program is verified with respect to Alice's policy, described by the precondition $P$ defined in the figure. This precondition is the iterated separating conjunction of 64 calendar slots; each slot's label is `Lo` if its value is 0 and `Hi` otherwise. A major novelty of this verification regards the conditional statement at lines 4-8. As mentioned earlier, in other IFC systems, the label of the boolean expression "$x = 0$" would have to be determined at the time of entering the conditional, and its label would then propagate into both branches via the pc label. In our system, however, we can reason that the expression's label (and hence the resulting pc label) will be different depending on which branch is taken. If the "true" branch is taken, then we know that $x$ is 0, and hence we know from the state assertion that its label is `Lo`. This means that the pc label is `Lo`, and so the output statement within this branch will not leak high-security data. If the "false" branch is taken, however, then we can reason that the pc label will be `Hi`, meaning that an output statement could result in

$$P \triangleq \overset{63}{\underset{i=0}{\circledast}} (A + i \mapsto (n_i, l_i) \wedge n_i = 0 \iff l_i = \texttt{Lo})$$

```
    Lo ⊢ {P}
1   i := 0;
    Lo ⊢ {P ∧ 0 ≤ i ∧ i.lbl = Lo}
2   while (i < 64) do
        Lo ⊢ {P ∧ 0 ≤ i < 64 ∧ i.lbl = Lo}
3       x := [A+i];
        Lo ⊢ {P ∧ 0 ≤ i < 64 ∧ i.lbl = Lo ∧
                    (x = 0 ⟺ x.lbl = Lo)}
4       if (x = 0)
5           then
                Lo ⊢ {P ∧ 0 ≤ i < 64 ∧ i.lbl = Lo ∧
                              x = 0 ∧ x.lbl = Lo}
6               output i
                Lo ⊢ {P ∧ 0 ≤ i < 64 ∧ i.lbl = Lo ∧
                              x = 0 ∧ x.lbl = Lo}
                Lo ⊢ {P ∧ 0 ≤ i < 64 ∧ i.lbl = Lo}
7           else
                Hi ⊢ {P ∧ 0 ≤ i < 64 ∧ i.lbl = Lo ∧
                              x ≠ 0 ∧ x.lbl = Hi}
8               skip;
                Hi ⊢ {P ∧ 0 ≤ i < 64 ∧ i.lbl = Lo ∧
                              x ≠ 0 ∧ x.lbl = Hi}
                Hi ⊢ {P ∧ 0 ≤ i < 64 ∧ i.lbl = Lo}
        Lo ⊢ {P ∧ 0 ≤ i < 64 ∧ i.lbl = Lo}
9       i := i+1
        Lo ⊢ {P ∧ 0 ≤ i ∧ i.lbl = Lo}
    Lo ⊢ {P ∧ i ≥ 64 ∧ 0 ≤ i ∧ i.lbl = Lo}
```

Figure 3.6: Calendar Example Verification

a leaky program (e.g., if the value of $x$ were printed). This program does not attempt to output anything within this branch, so it is still valid.

Since the program is verified with respect to precondition $P$, the noninterference guarantee for this example says that if we change any high-security event in Alice's calendar to any other high-security event (i.e., nonzero value), then the output will be unaffected. In other words, an observer cannot infer any information about the scheduled events in Alice's calendar.

## 3.5 Noninterference

Finally, we can now discuss how we formally prove our program logic's security guarantee. Much of the work has already been done through careful design of the security-aware semantics and the inference rules of the program logic. The fundamental idea is that we can find a bisimulation relation for our `Lo`-context (i.e., pc label is `Lo`) instrumented semantics. This relation will guarantee that two executions operate in lock-step, always producing the same program continuation and output.

The bisimulation relation we will use is called *observable equivalence*. It intuitively says that the low-security portions of two states are identical; the relation is commonly used in many IFC systems as a tool for proving noninterference. In our system, states $\sigma_1$ and $\sigma_2$ are observably equivalent if: (1) they contain equal values at all locations that are present and `Lo` in both states; and (2) the presence and labels of all store variables are the same in both states. This may seem like a rather odd notion of equivalence (in fact, it is not even transitive, so "equivalence" is a misnomer here) — two states can be observably equivalent even if some heap location contains `Hi` data in one state and `Lo` data in the other. To see why we need to define observable equivalence in this way, consider a heap-write command $[x] := E$ where $x$ is a `Hi` pointer. If we vary the value of $x$, then we will end up writing to two different

locations in the heap. Suppose we write to location 100 in one execution and location 200 in the other. Then location 100 will contain `Hi` data in the first execution (as the `Hi` pointer taints the value written), but it may contain `Lo` data in the second since we never wrote to it. Thus we design observable equivalence so that this situation is allowed.

The following definitions describe observable equivalence formally:

**Definition 2** (Observable Equivalence of Stores)**.** *Suppose $s_1$ and $s_2$ are variable stores. We say that they are observably equivalent, written $s_1 \sim s_2$, if, for all program variables $x$:*

- *If* $s_1(x) = \texttt{None}$*, then* $s_2(x) = \texttt{None}$*.*
- *If* $s_1(x) = \texttt{Some}\ (v_1, \texttt{Hi})$*, then* $s_2(x) = \texttt{Some}\ (v_2, \texttt{Hi})$ *for some* $v_2$*.*
- *If* $s_1(x) = \texttt{Some}\ (v, \texttt{Lo})$*, then* $s_2(x) = \texttt{Some}\ (v, \texttt{Lo})$*.*

**Definition 3** (Observable Equivalence of Heaps)**.** *Suppose $h_1$ and $h_2$ are heaps. We say that they are observably equivalent, written $h_1 \sim h_2$, if, for all natural numbers $n$:*

- *If* $h_1(n) = \texttt{Some}\ (v_1, \texttt{Lo})$ *and* $h_2(n) = \texttt{Some}\ (v_2, \texttt{Lo})$*, then* $v_1 = v_2$*.*

We say that two states are observably equivalent (written $\sigma_1 \sim \sigma_2$) when both their stores and heaps are observably equivalent. Given this definition, we define a convenient relational denotational semantics for state assertions as follows:

$$(\sigma_1, \sigma_2) \in [\![P]\!]^2 \iff \sigma_1 \in [\![P]\!] \wedge \sigma_2 \in [\![P]\!] \wedge \sigma_1 \sim \sigma_2$$

In order to state noninterference cleanly, it helps to define a "bisimulation semantics" consisting of the following single rule (the side condition will be discussed

below):

$$\frac{\langle \sigma_1,\, C,\, K\rangle \xrightarrow[\texttt{Lo}]{o} \langle \sigma_1',\, C',\, K'\rangle \qquad \langle \sigma_2,\, C,\, K\rangle \xrightarrow[\texttt{Lo}]{o} \langle \sigma_2',\, C',\, K'\rangle \qquad \text{(side condition)}}{\langle \sigma_1,\, \sigma_2,\, C,\, K\rangle \longrightarrow \langle \sigma_1',\, \sigma_2',\, C',\, K'\rangle}$$

Note that this bisimulation semantics operates on configurations consisting of a *pair* of states and a program. With this definition, we can split noninterference into the following progress and preservation properties.

**Theorem 10** (Progress)**.** *Suppose we derive* $\texttt{Lo} \vdash \{P\}\, C\, \{Q\}$ *using our program logic. For any* $(\sigma_1, \sigma_2) \in [\![P]\!]^2$*, suppose we have*

$$\langle \sigma_1,\, \sigma_2,\, C,\, K\rangle \longrightarrow_* \langle \sigma_1',\, \sigma_2',\, C',\, K'\rangle,$$

*where* $\sigma_1' \sim \sigma_2'$ *and* $(C', K') \neq (\texttt{skip}, [])$*. Then there exist* $\sigma_1''$*,* $\sigma_2''$*,* $C''$*,* $K''$ *such that*

$$\langle \sigma_1',\, \sigma_2',\, C',\, K'\rangle \longrightarrow \langle \sigma_1'',\, \sigma_2'',\, C'',\, K''\rangle$$

**Theorem 11** (Preservation)**.** *Suppose we have* $\sigma_1 \sim \sigma_2$ *and* $\langle \sigma_1,\, \sigma_2,\, C,\, K\rangle \longrightarrow \langle \sigma_1',\, \sigma_2',\, C',\, K'\rangle$*. Then* $\sigma_1' \sim \sigma_2'$*.*

For the most part, the proofs of these theorems are relatively straightforward. Preservation requires proving the following two simple lemmas about `Hi`-context executions:

1. `Hi`-context executions never produce output.

2. If the initial and final values of some location differ across a `Hi`-context execution, then the location must have a `Hi` label in the final state.

There is one significant difficulty in the proof that requires discussion. If $C$ is a heap-read command $x := [E]$, then Preservation does not obviously hold. The reason

for this comes from our odd definition of observable equivalence; in particular, the requirements for a heap location to be observably equivalent are weaker than those for a store variable. Yet the heap-read command is copying directly from the heap to the store. In more concrete terms, the heap location pointed to by $E$ might have a `Hi` label in one state and `Lo` label in the other; but this means $x$ will now have different labels in the two states, violating the definition of observable equivalence for the store.

We resolve this difficulty via the side condition in the bisimulation semantics. The side condition says that the situation we just described does not happen. More formally, it says that if $C$ has the form $x := [E]$, then the heap location pointed to by $E$ in $\sigma_1$ has the same label as the heap location pointed to by $E$ in $\sigma_2$.

This side condition is sufficient for proving Preservation. However, we still need to show that the side condition holds in order to prove Progress. This fact comes from induction over the specific inference rules of our logic. For example, consider the (READ) rule from Section 3.3:

$$\frac{P \Rightarrow \texttt{lbl}(E) = l_1 \qquad P \Rightarrow E \mapsto (n, l_2)}{l \vdash \{P\}\, x := [E]\, \{P \backslash x \wedge x = n \wedge x.\texttt{lbl} = l_1 \sqcup l_2 \sqcup l\}} \text{ (READ)}$$

In order to use this rule, we are required to show that the precondition implies $E \mapsto (n, l_2)$. Since both states $\sigma_1$ and $\sigma_2$ satisfy the precondition, we see that the heap locations pointed to by $E$ both have label $l_2$, and so the side condition holds. Note that the side condition holds even if $l_2$ is a logical variable rather than a constant.

In order to prove that the side condition holds for *every* verified program, we need to show it holds for all inference rules involving a heap-read command. In particular, this means that no heap-read rule in our logic can have a precondition that only implies $E \mapsto \_$.

Now that we have the Progress and Preservation theorems, we can easily combine them to prove the overall noninterference theorem for our instrumented semantics:

**Theorem 12** (Noninterference, Instrumented Semantics)**.** *Suppose we derive* $\texttt{Lo} \vdash \{P\}\, C\, \{Q\}$ *using our program logic. Pick any state* $\sigma_1 \in [\![P]\!]$*, and consider changing the values of any* `Hi` *data in* $\sigma_1$ *to obtain some* $\sigma_2 \in [\![P]\!]$*. Suppose, in the instrumented semantics, we have*

$$\langle \sigma_1,\, C,\, [] \rangle \xrightarrow[\texttt{Lo}]{o_1}{}_* \langle \sigma_1',\, \texttt{skip},\, [] \rangle$$

*and*

$$\langle \sigma_2,\, C,\, [] \rangle \xrightarrow[\texttt{Lo}]{o_2}{}_* \langle \sigma_2',\, \texttt{skip},\, [] \rangle.$$

*Then* $o_1 = o_2$*.*

Finally, we can use the results from Section 3.2 along with the safety guaranteed by our logic to prove the final, end-to-end noninterference theorem:

**Theorem 13** (Noninterference, True Semantics)**.** *Suppose we derive* $\texttt{Lo} \vdash \{P\}\, C\, \{Q\}$ *using our program logic. Pick any state* $\sigma_1 \in [\![P]\!]$*, and consider changing the values of any* `Hi` *data in* $\sigma_1$ *to obtain some* $\sigma_2 \in [\![P]\!]$*. Suppose, in the true semantics, we have*

$$\langle \bar{\sigma}_1,\, C \rangle \xrightarrow{o_1}{}_* \langle \tau_1,\, \texttt{skip} \rangle$$

*and*

$$\langle \bar{\sigma}_2,\, C \rangle \xrightarrow{o_2}{}_* \langle \tau_2,\, \texttt{skip} \rangle.$$

*Then* $o_1 = o_2$*.*

## 3.6 Problems with the Program Logic Approach

The program logic presented in this chapter is a convenient tool for formally proving the security of a C-like program with respect to a detailed and general policy.

The calendar example program of Figure 3.6 shows that the logic is significantly more powerful than many previous information-flow tracking frameworks, as it supports clean policy specification using conditional labels, and it exploits the power of Hoare Logic to perform fine-grained reasoning on an if statement that branches upon conditionally-labeled data.

The program logic approach is not perfect, however. In the following, we will discuss various suboptimal aspects of using a program logic to verify security.

**Language-Specific** In order to define logical inference rules for a program, we must have a clear, formal definition of the programming language being used. We choose a toy C-like language here that is simple enough to formally specify, but general enough to easily extend to handle many standard C features. Nevertheless, the language is fundamentally bound to a C-level memory model consisting of program variable store and global memory heap, as well as structured control flow (if-then-else statements and while loops, as opposed to labels and goto statements). This means that there is no easy way to adapt the program logic to support higher-level features such as high-order functions, or lower-level features such as unstructured jumps in assembly code. The latter problem is particularly of concern for the ultimate goal of this dissertation, since real-world systems sometimes do require reasoning directly about assembly code. For example, the context switch operation of an OS kernel must always be written directly in assembly since, after restoring registers, it must long jump to the location of a process's saved code pointer. The C language and memory model are too high level to directly write code that performs such a jump. Our program logic therefore cannot be used to prove anything about the context switch implementation within an OS kernel.

**Access to Code Details** As a prerequisite to using a program logic, one must of course have direct access to the code in question. This could be a problem in real-

world systems if, for example, the creator of the system would like to outsource the security verification to a third party. The creator would ideally wish to avoid leaking company secrets by revealing all details of the system's implementation. Instead, it would be far more preferable to only provide the third party with a formal, high-level specification of the system's functionality. Furthermore, this strategy would have the additional standard benefit of abstraction: if the low-level code of the system were changed without affecting the high-level specification, then the security verification would not need to be redone.

**Functional Correctness vs Security** Our program logic combines functional correctness verification with security verification. While this can certainly be viewed positively, since it allows for security to be proved in a single pass through the code, it also has the potential to conflate orthogonal aspects of the verification. For example, we discussed above how we interpret the verified precondition $P$ as a security policy; the precondition also serves as a safety specification, however, saying on which machine states the program executes without getting stuck. We cannot distinguish between which aspects of $P$ refer only to security, and which aspects refer only to safety. For example, if $P$ does not happen to mention anything about security labels, then it really says nothing of interest regarding the program's security. It might be more desirable to completely separate the verification of the program's functionality from the verification of its security, as this could potentially yield a clearer and more precise description of the program's security policy.

**Incompleteness** While we proved that our program logic is sound with respect to noninterference, it is certainly not complete — there are plenty of programs that are noninterfering under some precondition, but cannot be verified in our logic using that precondition. For example, if we slightly modify the program of Figure 3.5 by changing line 8 to output $i$, then the program will always output all the numbers

from 0 to 63 in order, regardless of values of high-security data. We would not be able to verify the program, however, because the pc label is `Hi` at line 8 and thus disallows any output. In the POST paper [14], we mention an informal observation: in our experience, it is always possible to rewrite a secure-but-unverifiable program in such a way that it produces the same output and becomes verifiable. For the altered calendar example just mentioned, it suffices to rewrite the program to simply print out the numbers 0 through 63 (without branching on elements in Alice's calendar).

A rather more complex example of this observation can be obtained by swapping lines 6 and 8 in the code of Figure 3.5. This program prints out all the time slots that are *not* free. Changing any (nonzero) event to any other (nonzero) event will not change this output, so the program is still secure with respect to Alice's policy. It is not verifiable for the same reason as before — output is disallowed at line 8. Nevertheless, this program can be rewritten in the following way (assume we add to the precondition that we have allocated a 64-element array filled with `Lo` 0's starting at location $B$, which will be used for temporary scratch work):

```
1  i := 0;
2  while (i < 64) do
3      x := [A+i];
4      if (x = 0) then [B+i] := 1 else skip;
5      i := i+1;
6  i := 0;
7  while (i < 64) do
8      x := [B+i];
9      if (x = 0) then output i else skip;
10     i := i+1;
```

The ability to rewrite these safe-but-unverifiable programs leads to some interesting ideas. The most direct idea is that it might be fruitful to pursue a formal proof

of semi-completeness of the program logic, saying that for any secure program $C$, there exists a program $C'$ with the same behavior (i.e., mapping from initial states to output produced) as $C$, and $C'$ can be verified secure using the inference rules of our logic. An even more interesting idea, however, is to abstract away from the individual programs $C$ and $C'$ entirely. Instead of using inference rules to verify $C$, what if we could first formally abstract $C$ into a specification of its behavior $S$, and then simply prove that $S$ is secure? If we could successfully accomplish this, then this incompleteness issue would no longer be relevant.

**Conclusion** It should be clear by now that all of these problems are steering us toward a particular solution. In the coming chapters, we will present a new methodology for security verification, where we first abstract a program into a high-level specification, in a manner that is completely independent from security concerns. Then we define and prove a security policy over the specification only. The methodology is fairly independent from program language specifics, it hides low-level details of code, and it completely separates functional correctness concerns from security concerns. Crucially (and perhaps surprisingly), our methodology successfully guarantees that if a program's specification is proved secure, then the program itself (i.e., the implementation) will always execute in a secure fashion.

# Chapter 4

# Security Reasoning over Specifications

## 4.1 A New Methodology for Security Verification

In this chapter, we discuss the main ideas and challenges for defining and proving *secure specifications*, and for propagating security from a specification to its implementation. The fundamental idea is to connect everything using what we call an *observation function*. Recalling Figure 1.2, we use the observation function to (1) define a security policy at the specification level, (2) formalize and prove noninterference with respect to the policy, (3) define the observable whole-execution behaviors of a program, and finally (4) automatically propagate security from a high-level specification to a low-level implementation.

Intuitively, the observation function represents which parts of a program state are observable (i.e., low security) to each principal. The observation being made by a principal does not actually have to be a portion of the program state, however; it can be of *any* type, and it can be *any* arbitrary transformation on the program state. For a principal $p$ and program state $\sigma$, we express the observation function notationally

as $\mathcal{O}_p(\sigma)$. Occasionally, we will need to distinguish between observation functions of different abstract machines — we use the notation $\mathcal{O}_{M;p}(\sigma)$ to refer to the observation function of machine $M$.

Note that this chapter will only describe our methodology semiformally; full formalization of mathematical notations, definitions, and theorems will appear in Chapter 5.

### 4.1.1 High-Level Security Policies

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

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

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

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

program state), and the observation function is:

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

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

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

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

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

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

Notice how we use the observation function here to express a declassification policy. This is similar to how we used the logical precondition in Chapter 3, but it is more general since the observation can be of any type. In the program logic, the observation was fixed to be the set of data within program state that has a label of `Lo`.

The generality of our observation function allows for the expression of many dif-

ferent kinds of security policies. While we have not exhaustively studied the extent of policy expressibility, we have anecdotally found it to be similar to other frameworks that express observational equivalence in a purely semantic fashion, e.g., Sabelfeld et al.'s PER model [50] and Nanevski et al.'s Relational Hoare Type Theory [42]. Later in this chapter, we will revisit the various desirable security policies mentioned in Section 1.3, and show how each one can be expressed using an observation function. Before delving into those examples, however, we will first explain how an observation function induces a high-level security policy, as well as how the high-level policy can then be propagated across a security-preserving simulation to apply to a low-level implementation.

### 4.1.2 Security Formulation

**High-Level Security** We define our noninterference property over some specification $S$ exactly as one would expect given our discussions in Chapters 1 and 3. Specifically, for a given principal $p$, the property says that observable equivalence, or state indistinguishability, is preserved by the specification, where two states are said to be indistinguishable just when their observations are equal:

$$\sigma_1 \overset{p}{\sim} \sigma_2 \triangleq \mathcal{O}_p(\sigma_1) = \mathcal{O}_p(\sigma_2)$$

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

More formally, for any principal $p$ and specification $S$ expressed as a set of pairs of initial and final states, we say that $S$ is secure for $p$ if the following property holds

for all states $\sigma_1$, $\sigma_2$, $\sigma_1'$, and $\sigma_2'$:

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

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

Note that we will sometimes refer to this high-level security property as the "unwinding condition", since it is essentially the same as the standard unwinding condition in the literature [19, 20]. The term comes from the fact that this property is used to inductively prove end-to-end security for an entire execution.

**Low-Level Security** While the above property is used to prove security across entire specifications of functions like `add`, we ultimately require a security guarantee that applies to the small-step implementations of these specifications. Notice that both lines of code in the implementation of `add` satisfy the unwinding condition, so we could trivially apply transitivity to get noninterference of the implementation. However, this is not true in general. Consider an alternative implementation of `add` with the same specification:

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

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

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

**Low-Level Indistinguishability** For high-level security, we defined state indistinguishability to be equality of the state-parameterized observation functions. This definition may not work well at a lower level of abstraction, however, since security-relevant logical state may be hidden by refinement. For example, suppose we attach security labels to data in a high-level state, for the purpose of specifying the policy based on label tainting described in Chapters 1 and 3. Further suppose that we treat the labels as logical state, erasing them when simulating the high-level specification with its implementation (i.e., the low-level machine model does not contain any physical representation of the security labels). This means that, at the implementation level, we can no longer define the portion of program state belonging to a particular principal. Hence it becomes unclear what state indistinguishability should mean.

We resolve this difficulty by defining low-level state indistinguishability in terms of high-level indistinguishability and simulation. We say that, given a simulation

relation $R$ connecting specification to implementation, two low-level states are indistinguishable if there exist two indistinguishable high-level states that are related to the low-level states by $R$. This definition will be fully formalized in Chapter 5.

**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.

Instead of labeling the transition relation, we make use of 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 (i.e., the event is actually *observed*). 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 security 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.

### 4.1.3 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 simulations. One potential issue, known as the refinement paradox [29, 36, 37], 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` based on an unbiased, independent coin flip. $P$ 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 [39] avoids this issue in the same way. In that work, the authors frame their solution as a restriction that disallows specifications from exhibiting any *domain-visible* nondeterminism. Indeed, this can be seen clearly by specializing the

unwinding condition above such that states $\sigma_1$ and $\sigma_2$ are identical:

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

The successful security verifications of both seL4 and mCertiKOS provide 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 4.1 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 correctly 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

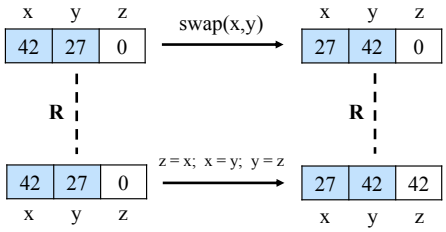



Figure 4.1: Security-Violating Simulation. The shaded part of state is unobservable, while the unshaded part is observable.

example of Figure 4.1, it is easy to demonstrate two indistinguishable program states that are related by $R$ to two distinguishable ones (since $R$ allows arbitrary change in the observable variable $z$). Thus our solution to this issue is to restrict simulations to require that state indistinguishability is preserved. More formally, given a principal $p$, in order to show that machine $m$ simulates $M$ under simulation relation $R$, the following property must be proved for all states $\sigma_1$, $\sigma_2$ of $M$, and states $s_1$, $s_2$ of $m$:

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

## 4.2 Representing Intricate Security Policies

Now that we have described how observation functions induce a high-level security policy and enforce a low-level security guarantee, let us revisit some of the interesting example policies of Section 1.3.

### 4.2.1 Declassify Parity

As a simple starting example, recall the `add` function mentioned above, and suppose we wish to enforce a security policy that declassifies the parity of secret data.

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

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

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

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

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

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

$$\begin{aligned}\mathcal{O}_A(\sigma_1) = \mathcal{O}_A(\sigma_2) \wedge (\sigma_1, \sigma_1') \in S_{\texttt{add}} \wedge (\sigma_2, \sigma_2') \in S_{\texttt{add}} \\ \implies \mathcal{O}_A(\sigma_1') = \mathcal{O}_A(\sigma_2')\end{aligned}$$

This reduces to:

$$\begin{aligned}&[a \hookrightarrow \sigma_1(a)\%2;\ x \hookrightarrow \sigma_1(x)\%2;\ y \hookrightarrow \sigma_1(y)\%2] = \\ &[a \hookrightarrow \sigma_2(a)\%2;\ x \hookrightarrow \sigma_2(x)\%2;\ y \hookrightarrow \sigma_2(y)\%2] \\ &\qquad \implies \\ &[a \hookrightarrow (\sigma_1(x) + \sigma_1(y))\%2;\ x \hookrightarrow \sigma_1(x)\%2;\ y \hookrightarrow \sigma_1(y)\%2] = \\ &[a \hookrightarrow (\sigma_2(x) + \sigma_2(y))\%2;\ x \hookrightarrow \sigma_2(x)\%2;\ y \hookrightarrow \sigma_2(y)\%2]\end{aligned}$$

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

### 4.2.2 Event Calendar Objects

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

$$\begin{aligned} \text{calendar } \mathcal{C} &\triangleq \mathbb{N} \to \texttt{option}\ \mathbb{Z} \\ \text{state } \Sigma &\triangleq \mathcal{P} \to \mathcal{C} \end{aligned}$$

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

1. Each principal can observe the entire contents of his or her own calendar.

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

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

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

This observation function only reveals details of scheduled events in a calendar to the calendar's owner, and therefore allows a principal to freely modify his or her own calendar securely. If different principals wish to collaborate in some way, we must verify that such collaboration is secure with respect to this observation function. For

example, consider a function `sched` that attempts to schedule some common event among a set of principals. Given a list of principals $P$ and an event $e$, the function will search for the earliest time slot $n$ that is free for all principals in $P$. If such a time slot is found, then all of the involved principals' calendars are updated with event $e$ scheduled at slot $n$. Otherwise, all calendars are unchanged. The following is pseudocode, and operates over a program state that contains an implementation of the per-principal calendars ($\Sigma$) in the array `cals`:

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

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

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

### 4.2.3 Security Labels and Dynamic Tainting

Our third example concerns dynamic labels and tainting, as described in Chapters 1 and 3. Even though the observation function is statically defined for an entire execution, we can exploit dynamic labels to change the observability of data during an execution. Assume we have a lattice of security labels $\mathbb{L}$, with the set of possible labels being a superset of principals $\mathcal{P}$. Let program state be a function mapping variables to a pair $(v, l)$ of integer value $v$ and security label $l$. For a given principal $p$, the observation function expresses the policy that all security labels are observable, but values are only observable if they have a label less than or equal to $p$ in the lattice:

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

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

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

The atomic specification of `add` is:

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

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

We could go further here and build an entire label-aware execution environment. Proving security of the high-level specifications is a similar process to proving soundness in other label-aware systems. We could then either treat the labels as purely logical state (like many statically-typed security systems), erasing them with a simulation relation, or we could verify a refinement to a machine like the one used in the SAFE system [10], where labels are actually implemented in the hardware and the physical machine performs dynamic label checks and tainting. Regardless of this choice of label representation, as long as we make sure our simulation relation preserves indistinguishability (as defined earlier), the security of the high-level specifications will automatically give us the whole-execution noninterference property for the low-level machine.

**Relation to Our Program Logic** This example can be viewed as a generalization of the strategy employed by the security-aware program logic of Chapter 3. The program logic directly modeled dynamic label tainting using a machine semantics instrumented with logical labels. There is, however, a significant difference: the small steps of the program logic's instrumented semantics do not satisfy the unwinding condition noninterference property presented in this chapter. Specifically, recall from Section 3.5 that the heap-read primitive instruction violates noninterference. Our solution presented in that section was to exploit properties of the specific inference rules of the program logic to establish a restriction on how the heap-read instruction can be used in the semantics.

It turns out that there is actually a way to define the instrumented semantics such that each step is automatically noninterfering, regardless of which particular inference rules are used for program verification. We discovered this fact by attempting to prove noninterference in Coq and figuring out precisely what goes wrong. We found that we could fix the proof by adding a specific (but rather unintuitive) dynamic label check to the instrumented heap-write instruction. The check uses the label of the *old* data in the heap ($l_3$ below) to enforce an upper bound:

$$\frac{\begin{array}{c}[\![E]\!]s = \texttt{Some}\ (n_1, l_1) \\ h(n_1) = \texttt{Some}\ (\_, l_3) \qquad [\![E']\!]s = \texttt{Some}\ (n_2, l_2) \qquad l_1 \sqcup l' \sqsubseteq l_3\end{array}}{\langle (s,h),\ [E] := E',\ K\rangle \xrightarrow[l']{} \langle (s, h[n_1 \mapsto (n_2, l_1 \sqcup l_2 \sqcup l')]),\ \texttt{skip},\ K\rangle}\ \text{(WRITE)}$$

While adding this check could potentially reduce the set of verifiably-secure programs (i.e., reduce completeness of the logic), it allows for noninterference to be entirely separated from the program logic inference rules. This key insight paved the way for us to develop the novel methodology presented in this chapter. Interestingly, after coming up with this additional label check, we later discovered that some other purely-dynamic (i.e., no program logic involved) security systems in the literature use the exact same check ([4, 26, 61]), and refer to it as the "no-sensitive-upgrade" requirement.

# Chapter 5

# Simulations and Security Propagation

In this chapter, we completely formalize everything discussed in Section 4.1, showing how security can be soundly propagated from a high-level specification to a low-level implementation. Figure 5.1 pictures the overall setup. We have many different machine semantics; the bottom one represents our lowest-level model of the actual systems code executing over physical hardware, while the top semantics represents our highest-level abstraction of the system, complete with logical state. We connect all of these semantics together with formal simulations, and show how the unwinding condition noninterference property at the highest abstraction level automatically guarantees an end-to-end, whole-execution noninterference property for the lowest level.

## 5.1 Machines with Observations

In the following, assume we have a set $\mathcal{P}$ of distinct principals or security domains.

**Definition 4** (Machine)**.** *A* state transition machine $M$ *consists of the following components (assume all sets may be finite or infinite):*

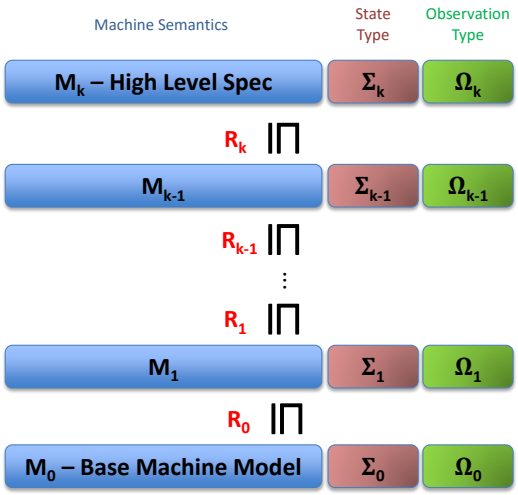



Figure 5.1: Basic Setup — Many simulations are chained together to incrementally refine a top-level specification semantics into a concrete implementation executing over a low-level assembly machine model.

- *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* $\mathbb{P}(\Sigma_M \times \Sigma_M)$
- *a type* $\Omega_M$ *of observations*
- *an observation function* $\mathcal{O}_{M;p}(\sigma)$ *of type* $\mathcal{P} \times \Sigma_M \to \Omega_M$

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 many 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 (which can contain both concrete and logical state) — this simplifies the theory, proofs, and policy specifications.

**Initial States vs Initialized States** Throughout our formalization, we do not require anything regarding initial states of a machine. The reason is related to how we will actually carry out security and simulation proofs in practice (described with respect to the mCertiKOS security proof in Chapters 6 and 7). 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.

## 5.2 High-Level Security

As described in Chapter 4, 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 5** (Safety)**.** *We say that a machine $M$ is safe under state predicate $I$, written* $\texttt{safe}(M, I)$*, when the following progress and preservation properties hold:*

$$\text{1.)} \quad \forall \sigma \in I - F_M \,.\, \exists \sigma' \,.\, \sigma \mapsto \sigma'$$

$$\text{2.)} \quad \forall \sigma, \sigma' \,.\, \sigma \in I \wedge \sigma \mapsto \sigma' \Longrightarrow \sigma' \in I$$

**Definition 6** (High-Level Security)**.** *Machine $M$ is secure for principal $p$ under invariant $I$, written $\Delta M_l^I$, just when:*

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

2.) $\forall \sigma_1, \sigma_2 \in I,\ \sigma_1', \sigma_2' \,.$

$$\mathcal{O}_p(\sigma_1) = \mathcal{O}_p(\sigma_2) \wedge \sigma_1 \mapsto \sigma_1' \wedge \sigma_2 \mapsto \sigma_2' \Longrightarrow \mathcal{O}_p(\sigma_1') = \mathcal{O}_p(\sigma_2')$$

3.) $\forall \sigma_1, \sigma_2 \in I \,.$

$$\mathcal{O}_p(\sigma_1) = \mathcal{O}_p(\sigma_2) \Longrightarrow (\sigma_1 \in F_M \iff \sigma_2 \in F_M)$$

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 initialization invariant $I$. The third property says that the finality of a state is observable to $p$ (again under invariant $I$); it is needed to close a potential termination-related security leak.

## 5.3 Low-Level Security

For low-level security, as discussed in Section 4.1, we first must define whole-execution behaviors with respect to a monotonic observation function.

**Definition 7** (Behavioral State)**.** *Given a machine $M$ and a partial order $\preceq$ over the observation type $\Omega_M$, we say that a program state $\sigma$ is behavioral for principal $p$, written $\flat M^p(\sigma)$, if all executions starting from $\sigma$ respect the partial order; i.e., the following monotonicity property holds:*

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

**Definition 8** (Behavioral Machine)**.** *We say that a machine $M$ is behavioral for principal $p$, written $\flat M^p$, when the machine has the following components:*

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

We next give a semiformal definition of whole-execution behaviors. 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 [34], except that we use state observations as the basic building block, while CompCert uses *traces*, which are input/output events labeled on transitions.

**Definition 9** (Whole-Execution Behaviors)**.** *Given a machine $M$ with a partial order defined over $\Omega_M$, and a state $\sigma$ that is behavioral for principal $p$, we write $\mathcal{B}_{M;p}(\sigma)$ 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. $\texttt{Fault}(o) \in \mathcal{B}_{M;p}(\sigma)$ *indicates that there is an execution* $\sigma \mapsto^* \sigma'$ *where* $\sigma'$ *is not a final state,* $\sigma'$ *cannot take a step to any state, and* $o = \mathcal{O}_p(\sigma')$.

2. $\texttt{Term}(o) \in \mathcal{B}_{M;p}(\sigma)$ *indicates that there is an execution* $\sigma \mapsto^* \sigma'$ *where* $\sigma'$ *is a final state and* $o = \mathcal{O}_p(\sigma')$.

3. $\texttt{Silent}(o) \in \mathcal{B}_{M;p}(\sigma)$ *indicates that there is an execution* $\sigma \mapsto^* \sigma'$ *where* $o = \mathcal{O}_p(\sigma')$ *and there is an infinite execution starting from* $\sigma'$ *for which all states in that infinite execution have identical observations (i.e., all observations are* $o$*).*

4. $\texttt{React}(os) \in \mathcal{B}_{M;p}(\sigma)$ *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* $\sigma' \mapsto \sigma''$ *with* $o = \mathcal{O}_p(\sigma'')$ *and* $\mathcal{O}_p(\sigma') \neq \mathcal{O}_p(\sigma'')$.

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 $\rho$, and will revisit the relation later in this section.

**Definition 10** (Low-Level Security)**.** *Given a machine $m$ that is behavioral for principal $p$, we say that $m$ is behaviorally secure for $p$ under some indistinguishability relation $\rho$, written $\nabla m_p^\rho$, just when:*

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

## 5.4 Simulation

We next formalize our definition of simulation. It differs from standard simulations in two primary aspects:

1. As explained above, we do not require any relationships to hold between initial states.

2. As described in Section 4.1.3, we require simulation relations to preserve state indistinguishability.

Recall the indistinguishability preservation property from Section 4.1.3:

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

One option would be to directly add this property into the definition of a simulation. For reasons that will become clear later, however, we will actually take a more roundabout path that defines simulations in such a way that the above property is *implied* rather than explicitly required. Whenever we wish to show a simulation from machine $M$ to machine $m$, we require not only a simulation relation $R$ that relates states of $M$ to states $m$, but also a function $f$ that translates observations of $M$ into observations of $m$. This translation function will be useful later when we need to reason about the relationship between simulations and whole-execution behaviors. A typical example of a translation function can be seen in the mCertiKOS security proof presented in Chapter 6: an abstract state observation contains many various parts including an output buffer, while a concrete state observation contains *only* the output buffer; hence the function $f$ simply returns the output buffer from the high-level observation.

**Definition 11** (Simulation)**.** *Given two machines $M$, $m$, a principal $p$, a relation $R$ between states of $M$ and states of $m$, and a total 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 $M \sqsubseteq_{R;f;p} m$,*

*when:*

$$1.) \quad \forall \sigma, \sigma' \in \Sigma_M,\ s \in \Sigma_m\ .$$

$$\sigma \mapsto \sigma' \wedge R(\sigma, s) \quad \Longrightarrow \quad \exists s' \in \Sigma_m\ .\ s \mapsto^* s' \wedge R(\sigma', s')$$

$$2.) \quad \forall \sigma \in \Sigma_M,\ s \in \Sigma_m\ .$$

$$\sigma \in F_M \wedge R(\sigma, s) \quad \Longrightarrow \quad s \in F_m$$

$$3.) \quad \forall \sigma \in \Sigma_M,\ s \in \Sigma_m\ .$$

$$R(\sigma, s) \quad \Longrightarrow \quad f(\mathcal{O}_{M;p}(\sigma)) = \mathcal{O}_{m;p}(s)$$

The first property is the main simulation, the second relates final states, and the third connects $R$ with $f$ in such a way that the indistinguishability preservation property from above is automatically implied. For presentation purposes, we omit details regarding the well-known "infinite stuttering" problem for simulations (described, for example, in [34]). Our Coq definition of simulation includes a well-founded order that prevents infinite stuttering.

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

## 5.5 End-to-End Security

We will now describe the main theorem of our framework: end-to-end security. There are too many technical details to present the entire proof here; instead, we will only state the primary lemmas involved, and then prove how these lemmas imply the main theorem. We begin with two helpful definitions.

**Definition 12** (Bisimulation)**.** *Given two machines $M$, $m$, a principal $p$, 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 $M \equiv_{R;f;p} m$, when $M \sqsubseteq_{R;f;p} m$ and $m \sqsubseteq_{R^{-1};f^{-1};p} M$.*

**Definition 13** (Invariant-Aware Indistinguishability)**.**

$$\Theta^I_p(\sigma_1, \sigma_2) \stackrel{\triangle}{=} \sigma_1 \in I \wedge \sigma_2 \in I \wedge \mathcal{O}_p(\sigma_1) = \mathcal{O}_p(\sigma_2)$$

The following lemma turns a proof of high-level security into a bisimulation.

**Lemma 7** (High-Level Security Bisimulation)**.**

$$\forall M, I, p \,.\, \Delta M^I_p \Longrightarrow M \equiv_{\Theta^I_p;Id;p} M$$

The next two lemmas say, respectively, that executions must exhibit at least one behavior, and that deterministic executions exhibit exactly one behavior.

**Definition 14** (Determinism)**.** *We say that a machine $M$ is deterministic, written $\downarrow M$, when the following properties hold:*

$$\text{1.)} \quad \forall \sigma, \sigma', \sigma'' \,.\, \sigma \mapsto \sigma' \wedge \sigma \mapsto \sigma'' \Longrightarrow \sigma' = \sigma''$$

$$\text{2.)} \quad \forall \sigma \in F_M \,.\, \neg\, \exists \sigma' \,.\, \sigma \mapsto \sigma'$$

**Lemma 8** (Behavior Exists)**.**

$$\forall M, p, \sigma \,.\, \flat M^p(\sigma) \Longrightarrow \mathcal{B}_{M;p}(\sigma) \neq \emptyset$$

**Lemma 9** (Behavior Determinism)**.**

$$\forall M, p, \sigma \,.\, \flat M^p(\sigma) \wedge \downarrow M \Longrightarrow |\mathcal{B}_{M;p}(\sigma)| = 1$$

The remaining lemmas convert simulations and bisimulations into behavior subset and equality, respectively. There is one significant barrier to stating these lemmas, however: 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, as the types may not match. To solve this problem, we will exploit our observation translation function. This is, in fact, the reason we use a translation function to define simulations instead of directly using the indistinguishability preservation property.

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

**Definition 15** (Behavior Subset)**.**

$$\mathcal{B}_{M;p}(\sigma) \ \sqsubseteq_f \ \mathcal{B}_{m;p}(s) \stackrel{\triangle}{=}$$
$$\forall b \,.\, b \in \mathcal{B}_{M;p}(\sigma) \Longrightarrow f(b) \in \mathcal{B}_{m;p}(s)$$

Similarly, for *invertible* $f$, we can define equality of behavior sets:

**Definition 16** (Behavior Set Equality)**.**

$$\mathcal{B}_{M;p}(\sigma) \ \equiv_f \ \mathcal{B}_{m;p}(s) \stackrel{\triangle}{=}$$
$$\mathcal{B}_{M;p}(\sigma) \ \sqsubseteq_f \ \mathcal{B}_{m;p}(s) \wedge \mathcal{B}_{m;p}(s) \ \sqsubseteq_{f^{-1}} \ \mathcal{B}_{M;p}(\sigma)$$

We now have the machinery to state the two remaining lemmas:

**Lemma 10** (Simulation and Safety Imply Behavior Subset)**.**

$$\forall M, m, I, R, f, p, \sigma, s \,.$$
$$\flat M^p(\sigma) \wedge \flat m^p(s) \wedge \texttt{safe}(M, I) \wedge M \ \sqsubseteq_{R;f;p} \ m \wedge \sigma \in I \wedge R(\sigma, s)$$
$$\Longrightarrow \mathcal{B}_{M;p}(\sigma) \ \sqsubseteq_f \ \mathcal{B}_{m;p}(s)$$

**Lemma 11** (Bisimulation Implies Behavior Equality)**.**

$$\begin{aligned}&\forall M, m, R, f, p, \sigma, s\ .\\ &\quad \flat M^p(\sigma) \wedge \flat m^p(s) \wedge M \ \equiv_{R;f;p}\ m \wedge R(\sigma, s)\\ &\quad \Longrightarrow \mathcal{B}_{M;p}(\sigma) \ \equiv_f \ \mathcal{B}_{m;p}(s)\end{aligned}$$

**Technical Aside** These two lemmas are not quite true as stated. If a single step in machine $M$ produces an event (i.e., changes the state observation) and is simulated by multiple steps in $m$, it could be the case that those multiple steps produce multiple events. In our Coq proof, we resolve this problem by defining a notion of "measure" that maps observations to natural numbers, and by requiring that (1) single steps never increase measure by more than one; and that (2) the observation translation function preserves measure. For example, the measure of an output buffer is defined to be its size, and no single step is allowed to append more than one output to the buffer. This is sufficient for conducting our mCertiKOS proof, but it is unfortunately a rather ad-hoc solution; we hope that future work will yield a cleaner one.

We have now stated all the required lemmas, and can move on to our primary theorem guaranteeing that simulations preserve security. As mentioned previously, low-level security uses an indistinguishability relation derived from high-level indistinguishability and a simulation relation:

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

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

**Theorem 14** (End-to-End Security)**.** *Suppose we have two machines $M$ and $m$, a principal $p$, a high-level initialization invariant $I$, and a simulation $M \ \sqsubseteq_{R;f;p}\ m$. Further suppose*

*that $m$ is deterministic and behavioral for $p$. Let low-level indistinguishability relation $\rho$ be $\phi(M, p, I, R)$ from Definition 17. Then high-level security implies low-level security:*

$$\Delta M_p^I \Longrightarrow \nabla m_p^\rho$$

*Proof.* For the first part of the proof, we define a new machine $N$ in between $M$ and $m$, and prove 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 $\Sigma_M$
- initial states $I_M$
- final states $F_M$
- transition relation $T_M$
- observation type $\Omega_m$
- observation function $\mathcal{O}_{N;p}(\sigma) \stackrel{\triangle}{=} f(\mathcal{O}_{M;p}(\sigma))$

First, we establish the simulation $M \sqsubseteq_{Id;f;p} N$. Referring to Definition 11, 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 $\mathcal{O}_{N;p}(-)$ given above.

Next, we establish the simulation $N \sqsubseteq_{R;Id;p} m$. The first two properties of Definition 11 are exactly the same as the first two properties of the provided simulation $M \sqsubseteq_{R;f;p} m$, and thus they hold. For the third property, assuming we know $R(\sigma, s)$, we have $Id(\mathcal{O}_{N;p}(\sigma)) = \mathcal{O}_{N;p}(\sigma) = f(\mathcal{O}_{M;p}(\sigma)) = \mathcal{O}_{m;p}(s)$, where the final equality comes from the third property of the provided simulation.

For the next part of the proof, we unfold the definition of what we are trying to prove,

$\nabla m_p^{\phi(M,p,I,R)}$. This yields:

$$\forall s_1, s_2 \,.\, (\exists \sigma_1, \sigma_2 \in I \,.\, \mathcal{O}_{M;p}(\sigma_1) = \mathcal{O}_{M;p}(\sigma_2) \wedge R(\sigma_1, s_1) \wedge R(\sigma_2, s_2))$$
$$\implies \mathcal{B}_{m;p}(s_1) = \mathcal{B}_{m;p}(s_2)$$

Pick any states $s_1$, $s_2$, $\sigma_1$, $\sigma_2$ such that $\sigma_1 \in I$, $\sigma_2 \in I$, $\mathcal{O}_{M;p}(\sigma_1) = \mathcal{O}_{M;p}(\sigma_2)$, $R(\sigma_1, s_1)$, and $R(\sigma_2, s_2)$. We will prove the desired $\mathcal{B}_{m;p}(s_1) = \mathcal{B}_{m;p}(s_2)$ by relating the behaviors of $N$ with those of $m$. In order to do this, however, we first must show that $N$ has well-defined behaviors for executions starting from $\sigma_1$ or $\sigma_2$. In other words, we must prove $\flat N^p(\sigma_1)$ and $\flat N^p(\sigma_2)$. We will focus on the proof for $\sigma_1$; the other proof is analogous. We use the same partial order as provided by the assumption $\flat m^p$. Consider any execution $\sigma_1 \mapsto^* \sigma_1'$ in $N$. Since $R(\sigma_1, s_1)$, we can use the simulation $N \sqsubseteq_{R;Id;p} m$ 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 $\mathcal{O}_{N;p}(\sigma_1) = \mathcal{O}_{m;p}(s_1)$ and $\mathcal{O}_{N;p}(\sigma_1') = \mathcal{O}_{m;p}(s_1')$. Since $m$ is behavioral, we also have $\mathcal{O}_{m;p}(s_1) \preceq \mathcal{O}_{m;p}(s_1')$. Hence we conclude $\mathcal{O}_{N;p}(\sigma_1) \preceq \mathcal{O}_{N;p}(\sigma_1')$, as desired.

We now know that $\flat N^p(\sigma_1)$ and $\flat N^p(\sigma_2)$. Notice that when $f$ is $Id$, our definitions of behavior subset and equality (Definitions 15 and 16) reduce to standard subset and set equality. Therefore, applying Lemma 10 to the established simulation $N \sqsubseteq_{R;Id;p} m$ tells us that $\mathcal{B}_{N;p}(\sigma_1) \subseteq \mathcal{B}_{m;p}(s_1)$ and $\mathcal{B}_{N;p}(\sigma_2) \subseteq \mathcal{B}_{m;p}(s_2)$ (note that the safety precondition of Lemma 10 holds because $M$ and $N$ have the same state type and transition relation, and high-level security of $M$ implies safety). Furthermore, since $m$ is deterministic, Lemma 9 gives us $|\mathcal{B}_{m;p}(s_1)| = |\mathcal{B}_{m;p}(s_2)| = 1$. Since Lemma 8 guarantees that neither $\mathcal{B}_{N;p}(\sigma_1)$ nor $\mathcal{B}_{N;p}(\sigma_2)$ is empty, we conclude that $\mathcal{B}_{N;p}(\sigma_1) = \mathcal{B}_{m;p}(s_1)$ and $\mathcal{B}_{N;p}(\sigma_2) = \mathcal{B}_{m;p}(s_2)$.

To complete the proof, we now just need to show that $\mathcal{B}_{N;p}(\sigma_1) = \mathcal{B}_{N;p}(\sigma_2)$. Applying Lemma 7 to our assumption of high-level security of $M$ gives us the bisimulation $M \equiv_{\Theta_p^I;Id;p} M$. We would like to apply Lemma 11, 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 $N \sqsubseteq_{\Theta_p^I;Id;p} N$ hold. If we can establish the third property, then we will obtain the desired bisimulation $N \equiv_{\Theta_p^I;Id;p} N$ since $Id$ is obviously invertible and $\Theta_p^I$ is symmetric. The third property requires us to prove that $\Theta_p^I(\sigma_1, \sigma_2) \implies \mathcal{O}_{N;p}(\sigma_1) = \mathcal{O}_{N;p}(\sigma_2)$. By definition, $\Theta_p^I(\sigma_1, \sigma_2)$ implies that $\mathcal{O}_{M;p}(\sigma_1) = \mathcal{O}_{M;p}(\sigma_2)$. Notice that $\mathcal{O}_{N;p}(\sigma_1) = \mathcal{O}_{N;p}(\sigma_2)$ following from this fact is exactly the indistinguishability preservation property we discussed earlier. Indeed, we have $\mathcal{O}_{N;p}(\sigma_1) = f(\mathcal{O}_{M;p}(\sigma_1)) = f(\mathcal{O}_{M;p}(\sigma_2)) = \mathcal{O}_{N;p}(\sigma_2)$.

Finally, we instantiate Lemma 11 with both machines being $N$. Notice that the required precondition $\Theta_p^I(\sigma_1, \sigma_2)$ holds by assumption. Lemma 11 now gives us the conclusion $\mathcal{B}_{N;p}(\sigma_1) \equiv_{Id} \mathcal{B}_{N;p}(\sigma_2)$. As mentioned earlier, behavior equality reduces to standard set equality when $f$ is $Id$, and so we get the desired $\mathcal{B}_{N;p}(\sigma_1) = \mathcal{B}_{N;p}(\sigma_2)$. □

# Chapter 6

# Security Overview of mCertiKOS

We will now discuss how to apply the methodology of Chapters 4 and 5 to formally guarantee end-to-end isolation between user processes running on top of the mCertiKOS kernel [21]. 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.

## 6.1 mCertiKOS Overview

The starting point for our proof effort was the basic version of the mCertiKOS kernel, described in detail in Section 7 of [21]. We will give an overview of the kernel here. It is composed of 32 abstraction *layers*, which incrementally build up the concepts of physical memory management, virtual memory management, kernel-level processes, and user-level processes. Each layer $L$ consists of the following components:

- a type $\Sigma_L$ of program state, separated into machine registers, concrete memory, and abstract data of type $D_L$
- a set of initial states $I_L$ and final states $F_L$
- a set of primitives $P_L$ implemented by the layer
- for each $f \in P_L$, a deterministic specification of type $\Sigma_L \to \texttt{option}\ \Sigma_L$

- *(if $L$ is not the bottom layer)* for each $f \in P_L$, an implementation written in either LAsm($L'$) or ClightX($L'$) (defined below), where $L'$ is the layer below $L$
- two special primitives called `load` and `store` that model access to global memory; these primitives have no implementation as they are a direct model of how the x86 machine translates virtual addresses using page tables

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

The assembly language, called LAsm($L$), is an extension of CompCert's [33] model of x86 assembly that allows primitives of layer $L$ to be called atomically. When an atomic primitive call occurs, the semantics consults that primitive's specification to take a step. Note that the `load` and `store` primitives are never called explicitly (as they have no implementation), but instead are used to specify the semantics of x86 instructions that read or write memory (e.g., `movl %eax, 0(%ecx)`).

The C variant, called ClightX($L$), is an extension of CompCert's Clight language [7] (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 [21], which is an extension of CompCert that supports per-function compilation.

Each layer $L$ induces a machine $M_L$ of the kind described in Section 5.1. The state type and initial/final states of $M_L$ come directly from $L$. The transition relation of type $\mathbb{P}(\Sigma_L \times \Sigma_L)$ is precisely 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.

**Layer Simulation** Figure 6.1 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

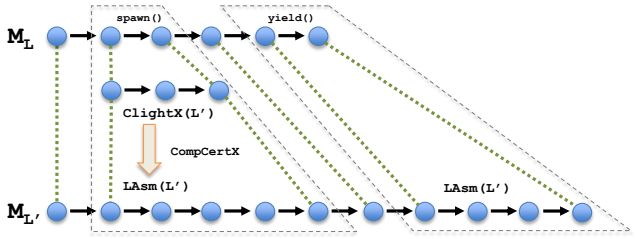



Figure 6.1: Simulation between adjacent layers. Layer $L$ contains primitives `spawn` and `yield`, with the former implemented in ClightX($L'$) and the latter implemented in LAsm($L'$).

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 directly in LAsm($L'$) (e.g., `yield` in Figure 6.1), then the simulation directly uses the small-step semantics of this implementation. If the primitive is implemented in ClightX($L'$) (e.g., `spawn` in Figure 6.1), then CompCertX's compilation is inserted into the simulation. CompCertX is verified to provide a simulation from the ClightX($L'$) execution to the corresponding LAsm($L'$) execution, so this is chained appropriately to 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 concrete memory in the lower-level machine is related to some newly-introduced portion of abstract data in the higher-level machine. In this way, as we move up the layers, concrete memory gets incrementally abstracted away in a monotonic fashion. Once we reach the top layer, TSysCall, concrete memory has been fully abstracted away from user-mode semantics; hence user processes have no mechanism for interacting with the concrete memory directly. In fact, by convention, mCertiKOS actually requires that primitive specifications at *all* layers do not interact with concrete memory. If a primitive needs to access some portion of concrete memory, then a layer must first be introduced to abstract that memory.

Once every pair of consecutive machines is connected with a simulation, they are combined transitively 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 describe the abstract data and primitives of the TSysCall layer in mCertiKOS-secure.

**TSysCall State** The TSysCall abstract data is a Coq record consisting of 32 separate fields. We list here those fields that will be relevant to our discussion later. In the following, whenever a field name has a subscript of $i$, the field is a finite map from process ID to some data type. Each user process running over the kernel has a unique, integer-valued process ID, and IDs are never reused. From a security standpoint, one should think of each process ID value as a unique principal or security domain.

- $\texttt{out}_i$ — The output buffer for process $i$, represented as a list of 32-bit integers. Note that output buffers exist in all layers' abstract data, including MBoot. They are never actually implemented in memory; instead, they are assumed to be a representation of some external method of output (e.g., a monitor or a network channel), and are used to define the observable events of user-process execution.

- `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 $3 \times 2^{30}$). 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 $n$ 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 tells the x86 machine how to translate virtual addresses between 0 and $2^{32} - 1$ into physical addresses.

- $\texttt{container}_i$ — Metadata for process $i$ 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 allocate.
  - `usage` — The current number of pages that process $i$ has 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 (head of the list is the next to be scheduled).

Note that process containers, which track parent/child relationships and dynamic resource usage, did not exist in the initial version of mCertiKOS taken from [21]. Prior to conducting our security proof, we realized that a lack of dynamic resource tracking would be extremely problematic for security, since a user process could easily affect others via a denial-of-service attack that repeatedly allocates pages until all of physical memory is exhausted. Therefore, inspired by the concept of containers in the HiStar security-aware operating system [62], we chose to implement a similar notion of container objects in mCertiKOS to preemptively deal with this security issue. Containers not only enforce a quota on the number of memory pages each process is allowed to dynamically allocate, but they also allow processes to distribute some of their memory quota to children. In Chapter 7, we will illustrate how containers and memory quotas are crucial to our security proof.

**TSysCall Primitives** There are 9 primitives in the TSysCall layer of mCertiKOS-secure, 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`.

- *Load/Store* — Since paging is enabled in all user-mode TSysCall states, the `load` and `store` primitives walk the two-level page table of the currently-running process to translate a virtual address 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 (see the `container` description above), 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). This means that the process will end up page faulting and yielding infinitely.

- *Get Quota* — `get_quota` returns the amount of remaining quota for the currently-executing process. This is useful to provide as a system call since it allows processes to divide their quota among children in any way they wish.

- *Spawn Process* — `proc_create` attempts to spawn a new child process. It takes a quota as a parameter, specifying the maximum number of pages the child process will be allowed to allocate. This quota allowance is taken from the current process's available quota.

- *Yield* — `sys_yield` performs the first step for yielding 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 (updating the ready queue accordingly). 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 integer parameter to the output buffer of the currently-running process.

## 6.2 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 security property that we aim to prove is exactly the high-level security defined in Section 5.2 (Definition 6), applied over the TSysCall machine using a carefully-constructed observation function that we define below. Theorem 14 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** As explained in Chapters 4 and 5, 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 actually hold over the TSysCall machine, because it models the execution of *all* user processes, not just the observer's process.

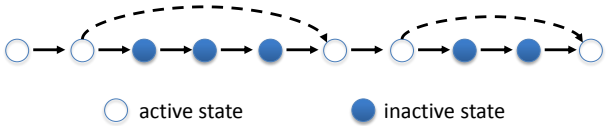



Figure 6.2: The TSysCall-local semantics, defined by taking big steps over the inactive parts of the TSysCall semantics.

To see this, consider any process ID $p$, which we call the observer process. For any TSysCall state $\sigma$, we say that $\sigma$ is "active" if $\texttt{cid}(\sigma) = p$, and "inactive" otherwise. Now consider whether the values in machine registers should be observable to $p$. Clearly, if $p$ 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 $p'$ is executing, then the registers must be unobservable to $p$ if we hope to prove that $p$ and $p'$ are isolated. We conclude that registers should be observable to $p$ 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, for example, 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 example.

The fundamental issue here is that, in order to prove that $p$ cannot be influenced by $p'$, we must show that $p$ has no knowledge that $p'$ is even executing over the kernel. We accomplish this by defining a higher-level machine above the TSysCall machine, where every state is active, meaning the semantics itself hides the executions of all processes except for the observer. We call this the TSysCall-local machine — it is parameterized by principal $p$, and it represents $p$'s local view of the TSysCall machine.

Figure 6.2 shows how the semantics of TSysCall-local is defined. The solid arrows are transitions of the TSysCall machine, white circles are active TSysCall states, and shaded circles are inactive states. The TSysCall-local semantics is then obtained by combining all

of the solid arrows connecting active states with all of the dotted arrows. Note that in the TSysCall layer, the yield primitive is the *only* way that a state can change from active to inactive, or vice-versa. Thus one can think of the TSysCall-local machine as a version of the TSysCall machine where the yield semantics takes a big step over other processes' executions, immediately returning to the observer 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 $p$. We prove simulation from TSysCall-local to TSysCall, so this strategy fits cleanly into our security verification methodology.

**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 $p$, the state observation of $\sigma$ is defined as follows:

- *Registers* — All registers are observable if $\sigma$ is active. No registers are observable if $\sigma$ is inactive.
- *Output* — The output buffer of $p$ is observable.
- *Virtual Address Space* — We can dereference any virtual address by walking through $p$'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 $p$ is observable.
- *Quota* — The remaining quota (max quota minus usage) of $p$ is observable.
- *Children* — The number of children of $p$ is observable.
- *Active* — It is observable whether $\texttt{cid}(\sigma)$ is equal to $p$.
- *Reg Ctxt* — The saved register context of $p$ is observable.

The virtual address space component of the observation function is particularly interesting, as it showcases the strength and generality of our methodology. Figure 6.3 shows pseudocode of the Coq specification used at the TSysCall level for loading data from the

```
Definition va_load p σ rs rd va :=
  match ZMap.get (PDX va) (pgmap_p σ) with
    PDEValid _ pte =>
      match ZMap.get (PTX va) pte with
        | PTEValid pg _ =>
            Next (rs # rd <-
              FlatMem.load (HP σ) (pg*PGSIZE + va%PGSIZE))
        | PTEUnPresent => exec_pagefault p σ rs va
      end
  end.
```

Figure 6.3: Pseudocode of the `load` primitive specification.

global heap (the `load` primitive). The two `match` clauses walk the two levels of page tables (`pgmap`) to convert a virtual address `va` into a physical page number `pg` and an offset (assuming a page fault does not occur). The resulting physical address, which we will refer to as `pa` in the following, is computed as $\texttt{pg} * \texttt{PGSIZE} + \texttt{va}\%\texttt{PGSIZE}$. Then `FlatMem.load` is called to obtain the value at location `pa` in the global heap `HP`. At first glance, it is not at all obvious how one might prove the security of this specification. In particular, the value of `pa` causes trouble: if the observer learns the physical address where his data is located, he could potentially learn some information about how other processes are allocating pages (e.g., the bigger the physical address, the more memory other processes have been using). In traditional label-based reasoning, `pa` would have a `Hi` security label, while the final data returned by the `FlatMem.load` lookup would have a `Lo` label. In other words, `FlatMem.load` performs a *declassification* here; yet this declassification does not actually result in an information leak. How can one prove that the declassification is acceptable?

Our verification methodology answers this question with ease. We simply define our observation function as described above, so that the physical address obtained during virtual address loading is unobservable. Only the following function is observed:

$$\mathcal{O}_p(\sigma) \quad \triangleq \quad \texttt{fun va} \Rightarrow \texttt{va\_load}\ p\ \sigma\ \texttt{va}$$

If we define the observation function in this way, and prove the high-level noninterference property (Definition 6) with respect to this observation function, then we successfully guar-

antee that end-to-end behavior of `va_load` really is independent from the specific value of `pa`. Hence our methodology cleanly and implicitly shows that the declassification performed by `va_load` is secure. Furthermore, this example demonstrates the power of allowing the observation function to express more than just a portion of program state. We define the observation to be a subtle transformation involving both the `pgmap` and `HP` portions of state.

# Chapter 7

# Proving Security of mCertiKOS

To prove end-to-end security of mCertiKOS, we must apply Theorem 14 of Chapter 5, using the simulation from TSysCall-local to MBoot, the high-level observation function described in Chapter 6, 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 (Definition 7 of Chapter 5).
3. The simulation from TSysCall-local to MBoot preserves indistinguishability.
4. TSysCall-local satisfies the high-level security property (Definition 6 of Chapter 5).

Determinism of the MBoot machine is already proved in mCertiKOS (in fact, all layers are deterministic). 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. Property 3 of Definition 11 then directly follows, using an observation translation function $f$ which simply projects the output buffer.

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

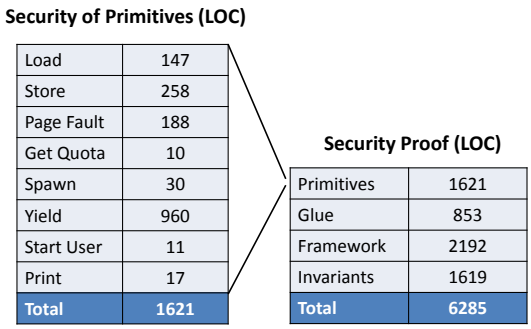



Figure 7.1: Approximate Coq LOC of proof effort.

each non-yield primitive of the TSysCall layer preserves indistinguishability. The yield primitive requires special treatment since the TSysCall-local semantics treats it differently; this will be discussed later in this section.

Figure 7.1 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 Chapter 5, and proofs of new state invariants that needed to be established. We will now discuss the most interesting aspects and difficulties of the TSysCall-local security proof.

## 7.1 Conducting 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 most important 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. A user process is always either in user mode, or is in kernel mode and the instruction pointer register points to the entry of the `start_user` primitive (meaning that the first part of yield was just executed, and user mode will be restored 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 (implying that page allocation will always be successful if the process has quota available).

Additionally, for a given observer principal $p$, we assume the invariant that process $p$ has been spawned. Anything occurring before the spawning of $p$ is considered part of the initialization/configuration phase; we are not interested in reasoning about the security of process $p$ 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 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, thus ruling out the counterexample.

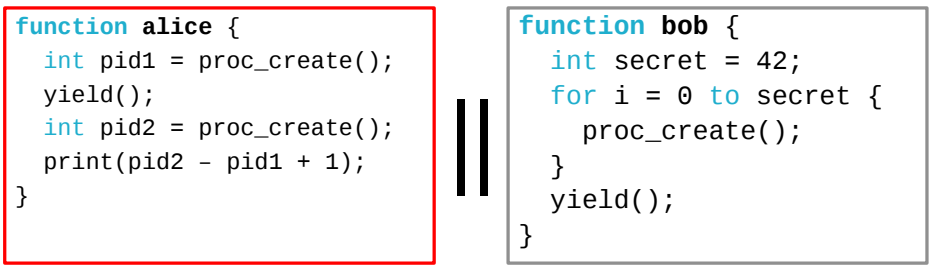



Figure 7.2: Using child process IDs to as a side channel.

**Security of Process Spawning** During our verification effort, we discovered that the `proc_create` primitive had a major security flaw. Figure 7.2 shows two user programs that exploit `proc_create` as a side channel for communication. When the insecure version of mCertiKOS creates a new child process, it chooses the lowest process ID not currently in use, and returns the ID to the user. In the figure, Alice spawns a child process, stores its ID into variable $x$, and then yields to Bob. Bob spawns a number of children equal to some secret value, and then yields back to Alice. Finally, Alices spawns another child, stores its ID into $y$, and observes the value $y - x - 1$. This observed value is exactly Bob's secret!

To close this side channel, we had to revamp the way child process IDs are chosen in mCertiKOS-secure. The new ID system works as follows. We define a global parameter $m$ limiting the number of children any process is allowed to spawn. Suppose a process with ID $i$ and $c$ children ($c < m$) spawns a new child. Then the child's ID will always be $i * m + c + 1$. 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 $j$. It is easy to see how this scheme closes the leak of Figure 7.2. $x$ will always the contain the ID of Alice's first child, while $y$ will contain the ID of Alice's second child; these two values are determined solely by Alice's own ID, so Bob is no longer capable of influencing them.

**Security of Page Fault** Since page faults dynamically allocate new pages, security of page faulting requires some reasoning about the resource quota implemented by containers (recall the container discussion in Chapter 6). More specifically, notice that the global

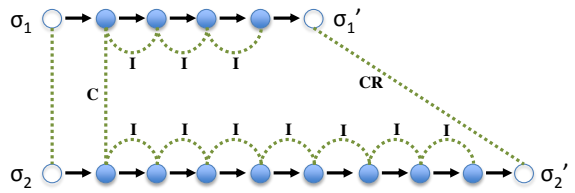



Figure 7.3: Applying the three lemmas to prove the security property of TSysCall-local yielding.

allocation table `AT` must be unobservable to user processes since all processes can affect it via page allocation. 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 deal with this difficulty, 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.

**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 7.3, where active states $\sigma_1$ and $\sigma_2$ are indistinguishable, and they both call yield. The TSysCall-local semantics takes a big step over the executions of all non-observer processes; these big steps are unfolded in Figure 7.3, so the solid arrows are all of the individual steps of the TSysCall semantics. We must establish that a big-step yield of the TSysCall-local machine preserves indistinguishability, meaning that states $\sigma_1'$ and $\sigma_2'$ in Figure 7.3 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 7.3. 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 $\sigma_1$ 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 $j$ is able to immediately call the page fault handler without first triggering a page fault, then it may very well learn some information about process $i$. We resolve this potential leak 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.

## 7.2 End-to-End Process Isolation

We conclude this chapter by taking a step back and revisiting the top-level end-to-end security theorem. The final statement is the conclusion of Theorem 14, which is low-level security of the MBoot machine for all states related by $\phi(M, p, I, R)$. In other words, the theorem says that if we consider any MBoot state $s_1$ related to a properly-initialized TSysCall state $\sigma_1$, and we consider related modified states $s_2$ and $\sigma_2$ with $\sigma_2$ initialized and $\mathcal{O}_p(\sigma_1) = \mathcal{O}_p(\sigma_2)$, then the whole-execution behaviors of MBoot on $s_1$ and $s_2$ are identical. Notice that this statement requires a solid understanding of the particular mCertiKOS

observation function. While it may feel intuitively clear that the observation function defined in Section 6.2 is reasonable, how can we be certain? Poorly-defined observation functions are certainly possible; for example, if we define an identity observation function where all principals observe all program state, then the unwinding condition degenerates into a simple statement of determinism. Thus applying our methodology with such an observation function would not actually give us a result that is related to security.

In general, we consider the observation function definition to be a trusted part of the methodology, in the same way that high-level specifications must be trusted to actually specify our intuitive desires for the software's behavior. However, depending on the context, we may be able to do better than this. While we have not yet completed a formal Coq development, we hope to prove a higher-level end-to-end security theorem in mCertiKOS that truly guarantees user-process isolation by being completely independent from the choice of observation function. Our vision for how this would work involves the following two steps:

1. Define a new state predicate $\texttt{Spawned}(p)$, saying that process $p$ was *just spawned* by the kernel. Prove that this predicate is stronger than the initialization predicate $I$ (i.e., $\forall p \,.\, \texttt{Spawned}(p) \Rightarrow I$).

2. Prove that *all* states satisfying $\texttt{Spawned}(p)$ are indistinguishable from one another according to $p$ (i.e., $\forall \sigma_1, \sigma_2 \in \texttt{Spawned}(p) \,.\, \mathcal{O}_p(\sigma_1) = \mathcal{O}_p(\sigma_2)$). We imagine that such a property should be provable since all of $p$'s data should be in a deterministic initial state (e.g., all zeros) immediately after $p$ is spawned.

Assuming we can successfully implement these steps, the end-to-end security guarantee can clearly be reformulated as follows: if we consider some MBoot state $s_1$ that is related to a TSysCall state $\sigma_1$ which occurs immediately after $p$ is spawned (i.e., $\sigma_1 \in \texttt{Spawned}(p)$), along with related modified states $s_2$ and $\sigma_2$ (with $\sigma_2 \in \texttt{Spawned}(p)$), then the whole-execution behaviors of MBoot on $s_1$ and $s_2$ are identical. With this theorem formulation, we no longer need to understand and trust the choice of observation function. While we still have some details to work out, we are hopeful that such a theorem can be established in the near future.

# Chapter 8

# New Feature: Virtualized Time

To demonstrate the extensibility of our methodology, we decided to add a new, useful feature to mCertiKOS-secure: the ability for user processes to time their own executions. Timing flows pose a notoriously difficult challenge for security reasoning. Nevertheless, we were able to successfully implement and verify the security of this new feature with only about two person-weeks of effort. The generality of the observation function is extremely helpful for clearly specifying how user processes view time.

## 8.1 Specification and Implementation of Timing

Figure 8.1 shows a motivating example for the new feature that we would like to support. We wish to provide a system call `gettime` that user processes can invoke to help time their executions. However, we do not want to allow processes to communicate with each other by exploiting this timestamp. Hence we must create some notion of *virtualized* time, whereby each process has its own isolated timeline. It is important to note that the feature we are implementing here is completely orthogonal to a wall-clock time. While a user can observe the time of an event within the context of his own timeline, our threat model still assumes that he has no mechanism for associating that event with a global wall-clock time.

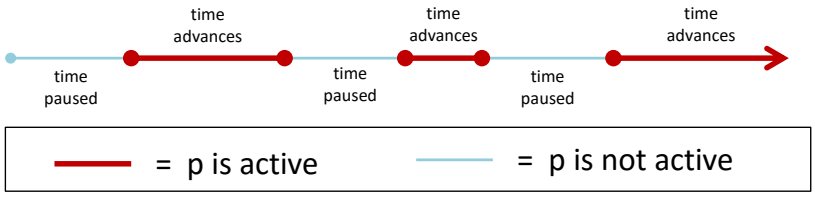
```
function alice {
  int t0 = gettime();
  while (true) {
    for i = 0 to 10^6 {
       // do some work...
    }
    int t = gettime();
    print(t - t0);
    yield();
  }
}
```

Figure 8.1: A sample usage of the `gettime` feature.

**TSC Oracle** As a basis for implementing the timing feature, we make use of x86's Time Stamp Counter (TSC), which keeps track of the total number of clock cycles since the last reset. At the machine model level (MBoot), we add a new primitive `rd_tsc`, and axiomatize its specification. The primitive's implementation is unverified but extremely simple: it looks up the current TSC value using x86's RDTSC instruction, and returns the value. Note that the TSC value clearly can be used as a potential information-flow side channel; thus the `rd_tsc` primitive is not exposed to users at the TSysCall level, and users are not allowed to directly invoke the RDTSC instruction (x86 has a global TSD flag which, when set, will cause a user-mode RDTSC invocation to trigger an exception).

Defining the specification of `rd_tsc` is somewhat subtle. We certainly do not want to model the actual number of cycles on the machine, since this is highly nondeterministic. A single assembly instruction could take any number of cycles to execute depending on various unpredictable conditions. Rather than try to accurately specify the value of the TSC, we will instead assume the existence of an *oracle* for a given execution that will accurately answer the question, "for any $n$, what is the TSC value returned by the $n$th invocation of the RDTSC instruction?" This strategy is reasonable because of the following two facts: (1) our theorems will apply to *any* choice of oracle; and (2) for every actual execution, there exists some oracle consistent with that execution.

**Local Timeline** Given our specification of the physical TSC in terms of an oracle, we virtualize the TSC value for each user process by implementing isolated timelines. Fig-




```
int gettime() {
  int p = get_cid();
  int t = rd_tsc();
  return (sum_p + (t - cur));
}

void stoptime() {
  int p = get_cid();
  int t = rd_tsc();
  sum_p += t - cur;
}

void starttime() {
  cur = rd_tsc();
}
```

Figure 8.2: Illustration and implementation of local timelines.

ure 8.2 illustrates a timeline that is local to process $p$, and shows the code that implements all timelines. It is useful to think of our implementation in terms of stopwatches. Each process has its own stopwatch: `gettime` returns the value of the active (i.e., currently-running) process's stopwatch, `stoptime` pauses the active process's stopwatch, and `starttime` resumes the active process's stopwatch. Whenever a process yields, `stoptime` is called prior to context switching, the active process ID is changed to be the head of the ready queue, and finally `starttime` is called after context switching. This guarantees that, at any given moment during execution, all process's stopwatches are paused except for the active one.

The code of Figure 8.2 shows a straightforward way to implement these stopwatches. In the following, we will use the term *epoch* to refer to a portion of execution in between two consecutive yields. We say that an epoch belongs to process $p$ if $p$ is the active process during that epoch. At any moment during an execution, we use the term *current epoch* to refer to the portion of execution since the most recent yield. Given this terminology, the stopwatch implementation uses the following two fields of program state:

- $\texttt{sum}_i$ — For each process $i$, this integer value gives the total amount of time spent in previous completed epochs belonging to $i$.
- `cur` — This integer value records the TSC value of the beginning of the current epoch.

Given these fields, the current virtualized time at any moment can be computed as $\texttt{sum}_p + (t - \texttt{cur})$, where $p$ is the active process and $t$ is the current TSC value returned by `rd_tsc`. When-

ever a yield occurs, the appropriate `sum` is increased by the length of the just-completed epoch (`stoptime`), and then `cur` is reset to the current TSC value (`starttime`). In this way, each process has its own virtualized timeline.

**Aside on TSC Overflow** Technically, we need to be concerned with errors in our code arising from the TSC value overflowing. However, the TSC is a 64-bit value in x86 architecture, so a machine would have to be running for well over a century on modern architecture in order for overflow to occur. Therefore, we choose to embed an assumption within our verification that the return value of `rd_tsc` is always less than $2^{64}$. Notice that this assumption implies that the virtualized time $\texttt{sum}_p + (t - \texttt{cur})$ also does not overflow, since this value is a subset of the timeline interval from 0 to $t$.

## 8.2 Security of Virtualized Time

The generality of the observation function allows for a clean and simple security verification of virtualized time. For convenience, we first add a field `tsc` to abstract state that represents the physical TSC. It gets updated to the return value of `rd_tsc` whenever that primitive is called. In other words, `tsc` simply remembers the value that `rd_tsc` returned the last time it was called. Given this, we define three timing-related observations made by process $p$:

- *Previous Epochs* — The value of $\texttt{sum}_p$ is observable.
- *Current Epoch* — The value of $(\texttt{tsc} - \texttt{cur})$ is observable if $p$ is the active process.
- *Oracle* — The $p$-filtered subset of the TSC oracle is observable (discussion below).

The first two observations clearly imply that the virtualized time $\texttt{sum}_p + (t - \texttt{cur})$ is always observable when $p$ is the active process and $t$ is the value just returned by `rd_tsc`; this fact yields a simple security proof of the `gettime` system call. Notice how we exploit generality of the observation function here by making the difference between `tsc` and `cur` observable, even though each of the two values individually must be unobservable since they pinpoint timestamps on a *global* timeline rather than local.

The third observation is quite subtle. Consider the following two insights:

1. *The oracle cannot be entirely unobservable.* Recall that the specification of `gettime` first sets $t$ to be the next value given by the oracle, and then returns $\texttt{sum}_p + (t - \texttt{cur})$. In order to prove that this specification is noninterfering, we must know that the oracle produces the same $t$ value in two indistinguishable states. Therefore the oracle cannot be entirely hidden from observation.

2. *The oracle cannot be entirely observable.* It is possible that in two executions from indistinguishable states, a process $p'$ which is not the observer $p$ may invoke `gettime` a different number of times, implying an information flow from $p'$ to $p$. In other words, the integrity lemma of Chapter 7 will fail to hold if the entire oracle is observable to $p$, since other processes can clearly change the oracle by invoking `gettime`.

Together, these insights imply that only some portion of the oracle can be observable to $p$. In particular, we need to somehow filter the oracle, observing only those entries which correspond to `rd_tsc` invocations made by $p$. One potential solution would be to divide the single global oracle into many per-process oracles. Unfortunately, this solution makes it difficult to define the semantics of `rd_tsc` at the MBoot level. At the TSysCall level, we would choose which oracle to query based on the currently-running process, stored in the abstract data field `cid`; at the MBoot level, however, `cid` has not yet been abstracted. It might be possible to specify the MBoot-level semantics to inspect the concrete memory corresponding to `cid`, but this would cause all kinds of trouble since it would violate the mCertiKOS convention that primitive specifications never depend on concrete memory.

Instead, we use a simple solution that cleverly exploits our assumption of safety of the high-level machine (the first property of Definition 6 in Chapter 5). For each natural number $n$, we extend the oracle to return not only the current TSC value, but also an integer $p$ representing a process ID. At the TSysCall level, $p$ is used as a prerequisite for safe execution: the specification of `gettime` checks whether $p$ is equal to `cid`, and returns `None` if this check fails. At the MBoot level, on the other hand, $p$ is completely ignored by the `rd_tsc` specification, allowing us to define the specification without a need to refer to `cid`.

This means that there are some oracles for which a TSysCall-level execution gets stuck, but the corresponding MBoot-level execution is safe. Thankfully, this odd situation is irrelevant due to the fact that we assume safe execution at the TSysCall level as a prerequisite for all of our major theorems. Notice that this solution is still justifiable in the following sense: even though many choices of oracle will now yield faulty high-level execution, there still always exists some valid oracle for any actual, safe execution.

Given this extended definition of oracle, we can now $p$-filter an oracle by choosing only those entries with process ID $p$. Then, as mentioned above, we make only this $p$-filtered oracle subset observable to $p$. While the intuitive concept should be clear at this point, the technical details of defining $p$-filtering are actually still a bit tricky. Because we represent the oracle as a function from natural numbers to pairs, we also keep track of the current natural-number position in abstract state. Whenever `rd_tsc` is called, this position is incremented by one. When defining the observation, we must be careful how we treat this oracle position, since it could be a source of information leak. To make observability independent from the precise value of the oracle position, we define a function $\texttt{filter}(o, c, p, n)$ which takes oracle $o$, oracle position $c$, process ID $p$, and natural number $n$ as parameters, and returns the $n$th occurrence of pairs in $o$ with process ID equal to $p$, starting from position $c$. Then oracle observation is defined as:

$$\mathcal{O}_p(\sigma) \triangleq \lambda\ n\ .\ \texttt{filter}(\texttt{oracle}(\sigma), \texttt{oracle\_pos}(\sigma), p, n)$$

In other words, rather than make the oracle position $c$ observable, we make the infinite stream of $p$-related oracle entries starting from $c$ observable.

With the help of a few simple lemmas characterizing this `filter` function, the entire security verification of virtualized time goes through quite easily. The generality of the observation function is key in allowing us to prove security of this new kernel feature. This clearly demonstrates the extensibility of our novel methodology: even though we need to introduce the concept of an oracle for reasoning about time within mCertiKOS, the timing-sensitive security verification is still completely straightforward.

# Chapter 9

# Assumptions, Limitations, and Future Work

We have demonstrated that our new methodology is extremely general, and is effective in guaranteeing security of realistic, low-level systems code. Nevertheless, like any framework, it has its fair share of assumptions and limitations. In this chapter, we discuss the most important limitations in order to help contextualize the situations in which our methodology should or should not be applicable.

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

- *Completeness Gap* — Our model is certainly not complete for all user-level assembly programs, so it may be possible to violate security on actual hardware by exploiting unmodeled assembly instructions. One example of such an instruction is RDTSC, which reads the x86 timestamp counter, as described in Chapter 8. The TSC can be

used as a communication channel between processes, leaking information about how much time certain processes have spent executing. We do not model the RDTSC instruction — a user program that uses the instruction would not even be considered valid syntax in our model, so there is no way that any verified properties could apply to such a program. Note that even in the extension described in Chapter 8, we choose not to model the RDTSC instruction; instead, we axiomatize a `rd_tsc` primitive and call the RDTSC instruction in its unverified implementation.

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

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

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

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

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

We hope that future work can gradually model more and more hardware features and instructions like RDTSC that do not satisfy this assumption. Each new feature could potentially violate security, and thus will require some additional verification effort. For the RDTSC example, we would close the timestamp counter information channel by setting the timestamp disable flag (TSD), which causes the hardware to treat RDTSC as a privileged instruction. Then, if a user process attempts to execute the instruction, the hardware will generate an exception and trap into the kernel. The kernel will then handle the exception in a way that is verified to be secure (e.g., it could kill the process, yield to a different process, or return a virtualized timestamp as in Chapter 8).

**High-Level Policy Specification** As with any formal verification effort, we must trust that the top-level specification of our system actually expresses our intentions for the system, including the security policy specified as an observation function. Because observation functions can have any type, our notion of security is far more expressive than classical pure noninterference. This does mean, however, that it can potentially be difficult to comprehend the security ramifications of a complex or poorly-constructed observation function. We place the onus on the system verifier to make the observation function as clear and concise as possible. This view is shared by a number of previous security frameworks with highly-expressive policy specification, such as the PER model [50] and Relational Hoare Type Theory [42]. In our mCertiKOS security specification, the virtual address space observation provides a good example of a nontrivial but clear policy specification — hiding physical addresses is, after all, the primary reason to use virtual address spaces. Note, however, as we discussed in Section 7.2, it may actually be possible to remove this trust requirement in certain contexts by proving a higher-level theorem that is independent from the choice of observation function.

**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 6) 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 specification of 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 4.1.3. That example failed to preserve security across simulation because the local variable `z` was being considered observable. A caller of the `swap` primitive should obviously 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.

**User Process Safety** There are a number of assumptions required specifically for the mCertiKOS security guarantee (as opposed to the general theory). Most of these are directly inherited from the mCertiKOS soundness theorem, such as correctness of the bootloader, device drivers, and the CompCert assembler; see [21] for more details on these assumptions. There is, however, one new assumption that requires discussion here, related to the safety of user processes.

Notice that the theory presented in Chapter 5 requires a proof of safety of the top-level semantics, with respect to some initialization invariant $I$ (Definition 6). This means we must prove that $I$ is preserved by each individual step of the semantics, and that the semantics can always take a step from any state satisfying $I$ (i.e., standard preservation and progress properties). We have a proof of preservation for mCertiKOS, but not progress. The current version of mCertiKOS is non-preemptive and trusts user processes to have well-defined semantics. The TSysCall-local semantics can thus get stuck in either of the following ways:

- The semantics does not currently specify what should happen when a user process attempts to execute an assembly instruction that has undefined semantics, such as a division by zero. Ideally, an operating system should provide a sandbox environment for user processes, where any undefined instruction causes a trap into the kernel, and is handled by either killing the offending process or by yielding to a different process. mCertiKOS does not yet do this, but we hope this could be done in future work.

- The big steps of the TSysCall-local semantics (Figure 6.2) could get stuck if a process yields but is never scheduled again. Even if we proved that the kernel scheduler is fair (which would not be difficult as it currently only does round-robin scheduling), we would still need to assume that user processes always eventually call yield. This is a fundamental limitation of a non-preemptive kernel. There are plans to make mCertiKOS preemptive in the future, but this requires a significant amount of effort.

Because of this potential for the top-level semantics to get stuck, we assume a significant hypothesis in our Coq proof, which essentially says that neither of the two situations above ever happens. While this hypothesis is necessary at the moment, it can be completely discharged if mCertiKOS is upgraded with a sandbox feature and preemption.

**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.

# Chapter 10

# Related Work and Conclusions

## 10.1 Locality in Separation Logic

The definition of locality (or local action), which enables the frame rule, plays a critical role in Separation Logic [28, 46, 58]. Almost all versions of Separation Logic — including their concurrent [8, 9, 43], higher-order [6], and relational [57] variants, as well as mechanized implementation (e.g., [2]) — have always used the same locality definition that matches the well-known Safety and Termination Monotonicity properties and the Frame Property [58].

In Chapter 2, we argued a case for strengthening the definition of locality to enforce *behavior preservation*. This means that the behavior of a program when executed on a small state is identical to the behavior when executed on a larger state — put another way, excess, unused state cannot have any effect on program behavior. We showed that this change can be made to have no effect on the usage of Separation Logic, and we gave multiple examples of how it simplifies reasoning about metatheoretical properties.

**Determinism Constancy** One related work that calls for comparison is the property of "Determinism Constancy" presented by Raza and Gardner [45], which is also a strengthening of locality. While they use a slightly different notion of action than we do, it can be shown that Determinism Constancy, when translated into our context (and ignoring

divergence behaviors), is logically equivalent to:

$$\sigma_0[\![C]\!]\sigma_0' \wedge \sigma_0' \# \sigma_1 \Longrightarrow \sigma_0 \# \sigma_1 \wedge (\sigma_0 \bullet \sigma_1)[\![C]\!](\sigma_0' \bullet \sigma_1)$$

For comparison, we repeat our Forwards Frame Property here:

$$\sigma_0[\![C]\!]\sigma_0' \wedge \sigma_0 \# \sigma_1 \Longrightarrow \sigma_0' \# \sigma_1 \wedge (\sigma_0 \bullet \sigma_1)[\![C]\!](\sigma_0' \bullet \sigma_1)$$

While our strengthening of locality prevents programs from increasing state during execution, Determinism Constancy prevents programs from *decreasing* state. The authors use Determinism Constancy to prove the same property regarding footprints that we proved in Section 2.4.1. Note that, while behavior preservation does not imply Determinism Constancy, our concrete logic of Section 2.2 does have the property since it never decreases state (we chose to have the `free` command put the deallocated cell back onto the free list, rather than get rid of it entirely).

While Determinism Constancy is strong enough to prove the footprint property, it does not provide behavior preservation — an execution on a small state can still become invalid on a larger state. Thus it will not, for example, help in resolving the dilemma of growing relations in the data refinement theory of [18]. Due to the lack of behavior preservation, we do not expect the property to have a significant impact on the metatheory as a whole. Note, however, that there does not seem to be any harm in using *both* behavior preservation and Determinism Constancy. The two properties together enforce that the area of memory accessible to a program be constant throughout execution.

**Module Reasoning** Besides our discussion of data refinement in Section 2.4.2, there has been some previous work on reasoning about modules and their implementations within the context of Separation Logic. In [44], a "Hypothetical Frame Rule" is used to allow modular reasoning when a module's implementation is hidden from the rest of the code. In [6], a higher-order frame rule is used to allow reasoning in a higher-order language with hidden module or function code. However, neither of these works discuss relational reasoning

between different modules. We are not aware of any relational logic for reasoning about modules.

Section 2.4.5 discussed how behavior preservation is fundamentally important for combining local reasoning with security verification; one interesting area for future work would be to formalize this relationship in the context of mCertiKOS. Currently, mCertiKOS uses a single monolithic datatype for abstract state. In the future, the framework could be altered to instead divide abstract state into minimal composable pieces. This would yield clea primitive specifications that only operate over the portion of abstract state needed by the primitive (e.g., the `get_quota` specification would only take the `container` portion of abstract state as input). Behavior preservation would then need to be explicitly enforced in order to *soundly* and *securely* combine all of these "small" specifications into a single, system-wide guarantee.

## 10.2 Security-Aware Program Logic

In the area of language-based IFC reasoning [48], there are many type systems and program logics that share similarities with our logic presented in Chapter 3.

Amtoft et al. [1] develop a program logic for proving noninterference of a program written in a simple object-oriented language. They use relational assertions of the form "$x$ is independent from high-security data." Such an assertion is equivalent to saying that $x$ contains `Lo` data in our assertion language. Thus their logic can be used to prove that the final values of low-security data are independent from initial values of high-security data — this is pure noninterference. Note that, unlike our logic, theirs does not attempt to reason about declassification. Some other differences between these IFC systems are:

- We allow pointer arithmetic, while they disallow it by using an object-oriented language. Pointer arithmetic adds significant complexity to information flow reasoning. In particular, their system uses a technique similar to our `mark_vars` function for reasoning about conditional constructs, except that they syntactically search for all locations in both the store and heap that might be modified within the conditional.

With the arbitrary pointer arithmetic of our C-like language, it is not possible to syntactically bound all possible heap-writes, so we require the additional semantic technique described in Section 3.5 that involves enforcing a side condition on the bisimulation semantics.

- Our model of observable behavior provides some extra leniency in verification. Our logic allows some leaks to happen within the program state, so long as these leaks are not made observable via an output command. In their logic (and many other IFC systems), the enforcement mechanism must prevent those leaks within program state from happening in the first place. Of course, we take this idea to the extreme when we move away from a specific program logic in Chapters 4 and 5.

Banerjee et al. [5] develop an IFC system that specifies declassification policies through state predicates in basically the same way that we do. For example, they might have a (relational) precondition of "$A(x \geq y)$," saying that two states agree on the truth value of $x \geq y$. This corresponds directly to a precondition of "$x \geq y$" in our system, and security guarantees for the two systems are both stated relative to the precondition. The two systems have very similar goals, but there are a number of significant differences in the basic setup that make the systems quite distinct:

- Their system does not attempt to reason about the program heap at all. They have some high-level discussions about how one might support pointers in their setup, but there is nothing formal.

- Their system enforces noninterference primarily through a type system (rather than a program logic). The declassification policies, specified by something similar to a Hoare triple, are only used at specific points in the program where explicit "declassify" commands are executed. A type system enforces pure noninterference for the rest of the program besides the declassify commands. Their end-to-end security guarantee then talks about how the knowledge of an observer can only increase at those points where a declassify command is executed (a property called "gradual release", defined by Askarov and Sabelfeld [3]). Thus their security guarantee for individual

declassification commands looks very similar to our version of noninterference, but their end-to-end security guarantee looks quite different. We do not believe that there is any comparable notion of gradual release in our system, as we do not have explicit program points where declassification occurs.

- Because they use a type system, their system must statically pick security labels for each program variable. This means that there is no notion of dynamically propagating labels during execution, nor is there any way to express our novel concept of conditional labels. As a result, the calendar example program of Section 3.3 would not be verifiable in their system.

Jif [41] is a practical IFC language built on top of Java. It employs the Decentralized Label Model [40] to enforce a static type system that controls security and integrity of data in a decentralized environment. A decentralized label describes each user's access control policy for the data, and thus can viewed as an instance of our principal-parameterized observation function of Chapter 4. Because label checks occur throughout the various typing rules, there is a close relationship between Jif and the static, instrumented semantics of our program logic. Declassifications in Jif are performed through an explicit declassify command in the language, however, and no attempts are made to provide any formal security guarantees in the presence of such declassifications.

The language-based IFC systems mentioned above, as well as our own program logic, use static reasoning. There are also many dynamic IFC systems (e.g., [4, 25, 54, 59]) that attempt to enforce security of a program during execution. Because dynamic systems are analyzing information flow at runtime, they will incur some overhead cost in execution time. Static IFC systems need not necessarily incur extra costs. Indeed, in our setup we have a "true machine" that executes on states with all labels erased (Figure 3.2). The security-aware machine is for reasoning purposes only; it will never be physically executed.

## 10.3 Security Verification over Specifications

**Noninterference and Relational Program Logics** There have been numerous relational program logics in the literature that naturally help with verification of noninterference properties, as noninterference is a relational property comparing two executions. In a relational program logic such as Yang's Relational Separation Logic [57], logical inference rules are used to verify a relational pre/post-condition pair for two programs. If the following Hoare triple is derived,

$$\{R\} \begin{array}{c} C \\ \\ C' \end{array} \{S\},$$

where $R$ and $S$ are relational predicates, then we are guaranteed that: if (1) two initial states $\sigma_1$ and $\sigma_2$ satisfy $R$, (2) $C$ takes $\sigma_1$ to final state $\sigma_1'$, and (3) $C'$ takes $\sigma_2$ to final state $\sigma_2'$, then $\sigma_1'$ and $\sigma_2'$ must satisfy $S$. This kind of program logic can easily support noninterference: we just make $C$ and $C'$ be the same program, and $R$ and $S$ both say that states are related if they are *indistinguishable* to an observing principal or security domain. Then the soundness property just described becomes the standard unwinding condition of noninterference. As mentioned in Section 10.2, Amtoft et al. [1] and Banerjee et al. [5] present two systems that employ this view of relational program logics for verifying noninterference. Our own security-aware program logic presented in Chapter 3 (and in [14]) also does something similar, although we directly model logical security labels in program state to allow for unary predicates rather than relational predicates; the unary predicates are easier to work with and cleaner for describing intricate security policies.

All of these program logics unfortunately suffer from the issues mentioned in Section 3.6. Program logics are inherently connected to a specific programming language; if one has a system that links together code written in different languages (e.g., C and assembly), then a program logic would need to be designed for *each* language being used. Program logics also assume full access to the entire system's codebase, which may be an unrealistic assumption under some circumstances. Additionally, security-aware program logics necessarily suffer from some level of incompleteness, since they reason about a program's security on line-by-

line basis, and therefore may not be able to infer that some seemingly-insecure operation within a function is actually completely hidden by the function's overall, end-to-end behavior. The novel security verification methodology presented in the dissertation gets around all of these difficulties by first abstracting all code within a system into precise and abstract functional specifications; all security verification is then performed over the specifications, and our special security-preserving simulations automatically propagate security from the specifications back down to the implementations.

**Observations and Indistinguishability** Our flexible notion of observation presented in Chapters 4 and 5 is similarly powerful to purely semantic and relational views of state indistinguishability, such as the ones used in Sabelfeld et al.'s PER model [50] and Nanevski et al.'s Relational Hoare Type Theory [42]. 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 approach is a generalization of Delimited Release [49] and Relaxed Noninterference [35]. 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.

Sabelfeld and Sands [51] define a road map for analyzing declassification policies in terms of four dimensions: *who* can declassify, *what* can be declassified, *when* can declassification

occur, and *where* can it occur. Our implicit notion of declassification can easily represent any of these dimensions due to the extreme generality of our methodology. The *who* dimension is handled directly via the explicit parameterization of the observation function based on principals. The *what* dimension is directly handled since the observation function is parameterized by program state, and can therefore specify exactly what data within the state is observable. The *when* dimension can be handled by representing time within program state (note that this piece of state could be either physical or logical). Similarly, we can handle the *where* dimension by including an explicit program counter within the state.

**Preserving Security across Simulation/Refinement** As explained in Chapter 4, refinements and simulations may fail to preserve security. There have been a number of solutions proposed for dealing with this so-called refinement paradox [29, 36, 37]. The one that is most closely related to our setup is Murray et al.'s seL4 security proof [38, 39], where the main security properties are shown to be preserved across refinement. As we mentioned in Chapter 4, 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.

## 10.4 Security Verification of mCertiKOS

**Comparison with mCertiKOS-base** Our verified secure kernel builds directly over the "base" version of mCertiKOS presented in [21]. 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 Chapters 4 and 5 (over 2000 lines of Coq code, as shown in Figure 7.1), and integrating these ideas into mCertiKOS. The previous mCertiKOS soundness theorem in [21] 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. This soundness theorem over whole-execution behaviors is then used to obtain the end-to-end noninterference property for the kernel.

**Security of seL4** An important work in the area of formal operating system security is the seL4 verified kernel [30, 38, 39, 52]. 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 Chapter 7, 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. Chapter 7 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 Chapter 6, but our methodology allows for all kinds of policy models depending on context. Thus, while the particular security property that we proved over mCertiKOS is not an advance over the seL4 security property, our new methodology involved in stating and proving the property, and for propagating security proofs through verified compilation and abstraction layers, is a significant advance.

**seL4 and Inter-Process Communication** As just mentioned, we verify pure isolation between processes when IPC is disabled, while seL4 uses intransitive noninterference [47] to specify a policy allowing for processes to communicate with each other. While the seL4 security property is certainly more general, it is also far more complex, and we do not feel the property gives a particularly useful security guarantee beyond its specialization to pure isolation (which happens when no processes use IPC). Intransitive noninterference allows one to specify an information flow relation between principals that is *intransitive* — e.g., a policy might say that Alice can flow to Bob and Bob can flow to Charlie, but Alice cannot flow to Charlie. Therefore, there is some inherent difficulty built into the property: Alice can clearly flow to Charlie by using Bob as a middleman. The final seL4 security theorem deals with this difficulty by using the intransitive flow relation to specify the minimum number of execution steps required for one principal to influence another. For example, we might say that Alice can influence Bob in $a$ execution steps and Bob can influence Charlie in $b$ steps, but Alice requires $a + b$ execution steps to influence Charlie.

For the mCertiKOS security verification, we choose to stick to the clearer property of pure isolation. We could certainly handle IPC in a similar way to seL4. For example: in

$a$ or more execution steps, Alice's observation is added to Bob's; in $b$ or more steps, Bob's observation is added to Charlie's; in $a + b$ or more steps, Alice's observation is added to Charlie's. We have not found much value, however, in such a property since it only provides guarantees for partial executions, up to a certain number of execution steps. We are much more interested in guaranteeing a clean, end-to-end, *whole-execution* security property.

**Security of Other OS Kernels** Dam et al. [15] aim to prove isolation of separate kernel components that are allowed to communicate across authorized channels. They do not formulate security as standard noninterference, since some communication is allowed. Instead, they prove a property saying that the machine execution is trace-equivalent to execution over an idealized model where the communicating components are running on physically-separated machines. Their setup is fairly different from ours, as we disallow communication between processes and hence prove noninterference. Furthermore, they conduct all verification at the assembly level, whereas our methodology supports verification and linking at both the C and assembly levels.

The Ironclad [23] system aims for full correctness and security verification of a system stack, which shares a similar goal to ours: provide guarantees that apply to the low-level assembly execution of the machine. The overall approaches are quite different, however. Ironclad uses Dafny [32] and Z3 [16] for verification, whereas our approach uses Coq; this means that Ironclad relies on SMT solving, which allows for more automation, but does not produce machine-checkable proofs as Coq does. Another difference is in the treatment of high-level specifications. While Ironclad allows some verification to be done in Dafny using high-level specifications, a trusted translator converts them into low-level specifications expressed in terms of assembly execution. The final security guarantee applies only to the assembly level; one must trust that the guarantee corresponds to the high-level intended specifications. Contrast this to our approach, where we verify that low-level execution conforms to the high-level policy.

**Asbestos, HiStar, and Flume** Asbestos [17] is a security-aware operating system that attempts to enforce security policies by monitoring label propagation between com-

municating processes. Each process $p$ has a send label $S_p$ representing the security level of information that has tainted $p$, and a receive label $R_p$ representing the maximum security level that $p$ is ever allowed to be tainted with. Process $p$ is allowed to send a message to process $q$ only if $S_p \sqsubseteq R_q$ (ordering defined by a lattice of labels), and $q$'s send label will be tainted by the message, increasing from $S_q$ to $S_q \sqcup S_p$. In this way, the operating system can prevent untrusted processes from maliciously or accidentally leaking users' secret data. Declassification is supported as well: a process may have declassification privileges for user Alice, implying that Alice trusts that process to only release her secret data in ways that she deems appropriate.

HiStar [62] is another security-aware operating system that was directly inspired by Asbestos. It expands upon the Asbestos label model to design a low-level kernel interface that tracks security label propagation between various kernel objects. While Asbestos only tracks labels between processes communicating via IPC mechanisms, HiStar tracks labels on *all* relevant resources, such as shared memory.

Both Asbestos and HiStar are helpful in providing users with some some amount of protection for their secret data. However, neither operating system provides any formal guarantees. Both the code and the label model are too complex to reasonably allow for formal reasoning. Flume [31] is an IFC system that provides security between user processes, and is built purely in user space. Flume borrows the label model of Asbestos/HiStar, and improves upon it by separating out mechanisms for privacy, integrity, authentication, declassification, and port send rights. Because the system operates purely at user level, and the label model is cleaner, some formal reasoning about Flume is possible. The notions of safety and security are formally defined within the model, and there is a formal argument that security is enforced. However, because Flume only models user space, all of the guarantees are predicated upon the assumption that the underlying operating system is behaving appropriately. This results in a potentially enormous trusted computing base.

The work presented in this dissertation has the potential to greatly improve the trustworthiness of IFC systems like Asbestos, HiStar, and Flume. By verifying the security of the entire operating system kernel API, we can remove Flume's reliance on trusting the entire

kernel codebase. In theory, one could imagine implementing Flume over the mCertiKOS API; however, Asbestos, HiStar, and Flume all use models that support explicit declassification via specially-marked trusted processes. As described previously, our methodology handles declassification differently: we require that declassifications are implicitly encoded within security policies through careful construction of an observation function. In other words, instead of allowing a certain process to be trusted by Alice to declassify her data, we require the process's functional specification to say precisely under what circumstances it will release Alice's data. Therefore, depending on the specific application, some additional specification work is required to directly support a Flume-like system within our methodology; however, this additional work yields a far more trustworthy guarantee since we do not need to trust either user processes with declassification privileges or the OS kernel code.

## 10.5 Conclusions

This dissertation presents a lengthy journey, starting from a novel, stronger notion of local reasoning that is compatible with security verification (Chapter 2), then moving on to a new program logic making use of this strong locality to formally guarantee security of C-like programs (Chapter 3), and finally learning from the problematic aspects of this program logic to devise a general methodology for security verification that is completely free from a specific programming language and logic (Chapters 4 and 5). The beauty of this final destination is then demonstrated with the formal security verification of a real, executable operating system kernel (Chapters 6, 7, and 8).

Ultimately, we consider the following abstraction principle to be the most fundamental and encompassing conclusion of this journey: whenever one desires to prove some property $P$ over a complex system implemented in low-level code, one should first verify a precise and descriptive *specification* of the system's behavior at an extremely high level of abstraction. Then property $P$ should be proved by looking *only* at the specification; all implementation code should be completely irrelevant. Of course, it is crucial that $P$ can then be soundly propagated from the specification to any correct implementation. This principle of abstrac-

tion is advocated by Gu et al. in the original presentation of mCertiKOS [21], with the desired precise high-level specifications being deemed "deep specifications". In that work, however, it is only an optimistic hope that the principle can be applied to any desired property $P$. In this dissertation, we demonstrate some solid evidence by showing that a property as complex as noninterference fits cleanly. Indeed, noninterference is famous in the literature for not being preserved across program refinement. Nevertheless, our novel contribution shows that this problem can be resolved in a clean manner with only a minor strengthening of the requirements for refinement. In an ideal future, all software would come with a highly-abstracted deep specification, and all properties of interest would be derivable by utilizing *only* this deep specification.

# Bibliography


[1] Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. A logic for information flow in object-oriented programs. In *POPL*, pages 91–102, 2006.

[2] Andrew W. Appel and Sandrine Blazy. Separation logic for small-step cminor. In *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings*, pages 5–21, 2007.

[3] Aslan Askarov and Andrei Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *2007 IEEE Symposium on Security and Privacy (S&P 2007), 20-23 May 2007, Oakland, California, USA*, pages 207–221, 2007.

[4] Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In *PLAS*, pages 113–124, 2009.

[5] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Expressive declassification policies and modular static enforcement. In *IEEE Symposium on Security and Privacy*, pages 339–353, 2008.

[6] Lars Birkedal, Noah Torp-Smith, and Hongseok Yang. Semantics of separation-logic typing and higher-order frame rules. In *Proc. 20th IEEE Symposium on Logic in Computer Science*, pages 260–269, 2005.

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

[8] Stephen Brookes. A semantics for concurrent separation logic. In *Proc. 15th International Conference on Concurrency Theory (CONCUR'04)*, volume 3170 of *LNCS*, 2004.

[9] C. Calcagno, P.W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *Logic in Computer Science, 2007. LICS 2007. 22nd Annual IEEE Symposium on*, pages 366–378, July 2007.

[10] Silviu Chiricescu, André DeHon, Delphine Demange, Suraj Iyer, Aleksey Kliger, Greg Morrisett, Benjamin C. Pierce, Howard Reubenstein, Jonathan M. Smith, Gregory T. Sullivan, Arun Thomas, Jesse Tov, Christopher M. White, and David Wittenberg. Safe: A clean-slate architecture for secure systems. In *Proceedings of the IEEE International Conference on Technologies for Homeland Security*, November 2013.

[11] David Costanzo. Dissertation companion website. `http://www.cs.yale.edu/homes/dsc5/thesis.html`. Accessed: 2016-08-02.

[12] David Costanzo and Zhong Shao. A case for behavior-preserving actions in separation logic. In *Programming Languages and Systems - 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13, 2012. Proceedings*, pages 332–349, 2012.

[13] David Costanzo and Zhong Shao. A case for behavior-preserving actions in separation logic. Technical report, Dept. of Computer Science, Yale University, New Haven, CT, June 2012. `http://flint.cs.yale.edu/publications/bpsl.html`.

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

[15] Mads Dam, Roberto Guanciale, Narges Khakpour, Hamed Nemati, and Oliver Schwarz. Formal verification of information flow security for a simple ARM-based separation kernel. In *2013 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 223–234, 2013.

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

[17] Petros Efstathopoulos, Maxwell N. Krohn, Steve Vandebogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, M. Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005*, pages 17–30, 2005.

[18] Ivana Filipovic, Peter W. O'Hearn, Noah Torp-Smith, and Hongseok Yang. Blaming the client: on data refinement in the presence of pointers. *Formal Asp. Comput.*, 22(5):547–583, 2010.

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

[20] Joseph A. Goguen and José 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.

[21] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In *Proc. 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), Mumbai, India*, pages 595–608, 2015.

[22] Gurvan Le Guernic. Automaton-based confidentiality monitoring of concurrent programs. In *20th IEEE Computer Security Foundations Symposium, CSF 2007, 6-8 July 2007, Venice, Italy*, pages 218–232, 2007.

[23] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated full-system

verification. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Broomfield, CO, USA*, pages 165–181, 2014.

[24] Nevin Heintze and Jon G. Riecke. The slam calculus: Programming with secrecy and integrity. In *POPL*, pages 365–377, 1998.

[25] Catalin Hritcu, Michael Greenberg, Ben Karel, Benjamin C. Pierce, and Greg Morrisett. All your ifcexception are belong to us. In *IEEE Symposium on Security and Privacy*, pages 3–17, 2013.

[26] Catalin Hritcu, John Hughes, Benjamin C. Pierce, Antal Spector-Zabusky, Dimitrios Vytiniotis, Arthur Azevedo de Amorim, and Leonidas Lampropoulos. Testing noninterference, quickly. In *ICFP*, pages 455–468, 2013.

[27] Sebastian Hunt and David Sands. On flow-sensitive security types. In *POPL*, pages 79–90, 2006.

[28] Samin Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Proc. 28th ACM Symposium on Principles of Programming Languages*, pages 14–26, January 2001.

[29] Jan 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.

[30] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1), February 2014.

[31] Maxwell N. Krohn, Alexander Yip, Micah Z. Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard os abstractions. In *SOSP*, pages 321–334, 2007.

[32] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR) - 16th International Conference, Dakar, Senegal*, pages 348–370, 2010.

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

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

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

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

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

[38] Toby C. Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. sel4: From general purpose to a proof of information flow enforcement. In *IEEE Symposium on Security and Privacy*, pages 415–429, 2013.

[39] Toby C. Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, and Gerwin Klein. Noninterference for operating system kernels. In *Certified Programs and Proofs (CPP) - Second International Conference, Kyoto, Japan, Proceedings*, pages 126–142, 2012.

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

[41] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4):410–442, 2000.

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

[43] Peter W. O'Hearn. Resources, concurrency and local reasoning. In *CONCUR'04*, pages 49–67, 2004.

[44] Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. *ACM Trans. Program. Lang. Syst.*, 31(3):1–50, 2009.

[45] Mohammad Raza and Philippa Gardner. Footprints in local reasoning. *Journal of Logical Methods in Computer Science*, 5(2), 2009.

[46] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS'02*, pages 55–74, 2002.

[47] John Rushby. Noninterference, transitivity, and channel-control security policies. Technical report, dec 1992.

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

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

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

[51] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.

[52] Thomas Sewell, Simon Winwood, Peter Gammie, Toby C. Murray, June Andronick, and Gerwin Klein. seL4 enforces integrity. In *Interactive Theorem Proving (ITP) -*

*Second International Conference, Berg en Dal, The Netherlands, Proceedings*, pages 325–340, 2011.

[53] Paritosh Shroff, Scott F. Smith, and Mark Thober. Dynamic dependency monitoring to secure information flow. In *20th IEEE Computer Security Foundations Symposium, CSF 2007, 6-8 July 2007, Venice, Italy*, pages 203–217, 2007.

[54] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible dynamic information flow control in haskell. In *Haskell*, pages 95–106, 2011.

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

[56] V. N. Venkatakrishnan, Wei Xu, Daniel C. DuVarney, and R. Sekar. Provably correct runtime enforcement of non-interference properties. In *Information and Communications Security, 8th International Conference, ICICS 2006, Raleigh, NC, USA, December 4-7, 2006, Proceedings*, pages 332–351, 2006.

[57] Hongseok Yang. Relational separation logic. *Theor. Comput. Sci.*, 375(1-3):308–334, 2007.

[58] Hongseok Yang and Peter W. O'Hearn. A semantic basis for local reasoning. In *Proc. 5th Int'l Conf. on Foundations of Software Science and Computation Structures (FOSSACS'02)*, volume 2303 of *LNCS*, pages 402–416. Springer, 2002.

[59] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. In *POPL*, pages 85–96, 2012.

[60] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Improving application security with data flow assertions. In *SOSP*, pages 291–304, 2009.

[61] Stephan Arthur Zdancewic. *Programming Languages for Information Security.* PhD thesis, Ithaca, NY, USA, 2002. AAI3063751.

[62] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. In *OSDI*, pages 263–278, 2006.