A Separation Logic for Enforcing Declarative Information Flow Control Policies

David Costanzo and Zhong Shao

Yale University

Abstract. In this paper, we present a program logic for proving that a program does not release information about sensitive data in an unintended way. The most important feature of the logic is that it provides a formal security guarantee while supporting "declassification policies" that describe precise conditions under which a piece of sensitive data can be released. We leverage the power of Hoare Logic to express the policies and security guarantee in terms of state predicates. This allows our system to be far more specific regarding declassification conditions than most other information flow systems.
The logic is designed for reasoning about a C-like, imperative language with pointer manipulation and aliasing. We therefore make use of ideas from Separation Logic to reason about data in the heap.

1 Introduction

Information Flow Control (IFC) is a field of computer security concerned with tracking the propagation of information through a system. A primary goal of IFC reasoning is to formally prove that a system does not inadvertently leak high-security data to a low-security observer. A major challenge is to precisely define what "inadvertently" should mean here.

A simple solution to this challenge, taken by many IFC systems (e.g., [7, 8, 14, 19, 23]), is to define an information-release policy using a lattice of security labels. A *noninterference* property is imposed: information cannot flow down the lattice. Put another way, any data that the observer sees can only have been influenced by data with label less than or equal to the observer's label in the lattice. This property is sometimes called *pure noninterference.*

Purely-noninterfering systems are unfortunately not very useful. Almost all real-world systems need to violate noninterference sometimes. For example, consider one of the most standard security-sensitive situations: password authentication. In order for a password to be useful, there must be a way for a user to submit a guess at the password. If the guess is incorrect, then the user will be informed as such. However, the information that the guess was incorrect is dependent on the password itself; the user (who might be a malicious attacker) learns that the password is definitely *not* the one that was guessed. This represents a flow of information (albeit a minor one) from the high-security password to the low-security user, thus violating noninterference. In a purely noninterfering system, sensitive data has no way whatsoever of affecting the outcome of a computation, and so the situation is essentially equivalent to the data not being present in the system at all.

There have been numerous attempts at refining the notion of inadvertent information release beyond the rules of a strict lattice structure. IFC systems commonly allow for some method of *declassification*, a term used to describe an information leak (i.e., an information flow moving down the security lattice) that is understood to be in some way "acceptable" or "purposeful" (as opposed to "inadvertent"). These declassifications violate the pure noninterference property described above. Ideally, an IFC system should still provide some sort of security guarantee even in the presence of declassification. It is quite rare, however, for a system to have a satisfactory formal guarantee. Those that do usually must make significant concessions that limit the generality or practicality of the system.

Our primary goal is to leverage the strengths of a program logic to devise a powerful IFC system that provides formal security guarantees even in the presence of declassification. A secondary goal is to avoid relational reasoning, which is usually required for the more expressive IFC reasoning systems (e.g., [15]) due to the nature of noninterference, but can be very difficult to use in practice.

We achieve these goals by using unary state predicates to refine the pure noninterference property into one that cleanly describes exactly how a piece of high-security data could affect observable output. Instead of simply saying that an observer cannot distinguish between any values of the high-security data, we say that the observer cannot distinguish between any values among a particular set — the set described by the state predicate. This method of refining pure noninterference to express a semantic notion of declassification appears in many previous IFC reasoning systems (e.g., [3],[15],[20]), though we take a rather unique approach toward designing a system that establishes the property. Our contributions in this paper are as follows:

- We define a novel, security-aware semantics for a simple imperative language with pointer arithmetic and aliasing. The semantics instruments state with security labels, and tracks information flow through propagation of these labels. We show that this semantics is sensible and overhead-free by relating its executions back to a standard small-step operational semantics without labels.
- We present a program logic for formally verifying the safety of a program under the security-aware semantics. The logic builds upon Hoare Logic [9] and Separation Logic [16, 17], and uses a unary predicate language syntax that has the ability to refer to security labels in the program state. Note that our choice of Separation Logic is somewhat arbitrary — we need to reason about low-level pointer manipulation, but a different pointer-analysis logic may be just as suitable.
- We prove a strong, termination-insensitive security guarantee for any program that is verified using our program logic. This guarantee generalizes traditional pure noninterference to account for semantic declassification.
- All of the technical work in this paper is fully formalized and proved in the Coq proof assistant. The Coq development can be found at [6].

The remainder of this paper is organized as follows: Section 2 informally discusses how our system works and highlights contributions; Section 3 defines our language, state model, and operational semantics; Section 4 describes the program logic and its soundness theorem relative to the operational semantics; Section 5 describes the noninterference-based security guarantee provided by the program logic; Section 6 describes related work; and Section 7 concludes.

2 Informal Discussion

In this section, we will describe our system informally in order to provide some high-level motivation. We pick a starting point of a C-like, imperative language with pointer arithmetic and aliasing, as we would like our logic to be applicable to low-level systems code. The main operations of our language are variable assignment $x := E$, heap dereference/load $x := [E]$, and heap dereference/store $[E] := E'$. The expressions E can be any standard mathematical expressions on program variables, so pointer arithmetic is allowed. Aliasing is also clearly allowed since $[x]$ and $[y]$ refer to the same heap location if x and y contain the same value.

2.1 Security Labels

Our instrumented language semantics will track information flow by attaching a *security label* to every value in the program state. For simplicity of presentation, we will assume that the only labels are `Lo` and `Hi` (a more general version of our system allows labels to be any set of elements that form a lattice structure). Unlike many static IFC reasoning systems, we attach the label to the *value* rather than the *location*. This means that a program is allowed to, for example, overwrite some `Lo` data stored in variable x with some other `Hi` data. Many other systems would instead label the location x as `Lo`, meaning that `Hi` data could never be written into it. Supporting label overwrites allows our system to verify a wider variety of programs.

Label propagation is done in a mostly obvious way. 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.

2.2 Noninterference

As discussed in Section 1, the ultimate goal of our IFC system is to prove a formal security guarantee that holds for any verified program. The standard security guarantee is 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.

We refine this noninterference property by requiring a precondition to hold on the initial state of an execution. That is, we alter the 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 of noninterference is interesting for two reasons. First, it provides a link between information flow security and Hoare Logic (a program logic that derives pre/postconditions as state predicates). Second, this property describes a certain level of dependency between high-security inputs and low-security outputs, rather than the complete independence of pure noninterference. This means that a program that satisfies this weaker noninterference may be semantically declassifying data. In this sense, we can use this property as an interesting security guarantee for a program that may declassify some data. To better understand this weaker version of noninterference, let us consider a few examples.

Public Parity Suppose we have a variable x that contains some high-security data. We wish to specify a *declassification policy* which says that only the parity of the `Hi` value can be released to the public. We will accomplish this by verifying the security of some program with a precondition P that says "x contains high data, y contains low data, and $y = x\%2$". Our security property then says that if we have an execution from some state satisfying P, then changing the value of x will not affect the output as long as the new state also satisfies P. Since y is the parity of x and is unchanged in the two executions, this means that as long as we change x to some other value *that has the same parity*, the output will be unchanged. Indeed, this is exactly the property that one would expect to have with a policy that releases only the parity of a secret value: only the secret's parity can influence the observable behavior.

Public Average Suppose we have three secrets stored in x, y, and z, and we are only willing to release their average as public (e.g., the secrets are employee salaries at a particular company). This is similar to the previous example, except that we now have multiple secrets. The precondition P will say that x, y, and z all contain `Hi` data, a contains `Lo` data, and $a = (x + y + z)/3$. In this situation, noninterference will say that we can change the value of the *set* of secrets from any triple to any other triple, and the output will be unaffected as long as the average of the three values is unchanged.

Public Zero Suppose we have a secret stored in x, and we are only willing to release it if it is zero. We could take the approach of the previous two examples and store a public boolean in another variable which is true if and only if x is 0. However, there is an even simpler way to represent the desired policy without using an extra variable. Our precondition P will say that either x is 0 and its label is `Lo`, or x is nonzero and its label is `Hi`. This is an example of a *conditional label*: a label whose value depends on some state predicate. If x is 0, then noninterference says nothing since there is no high-security data in the state. If x is nonzero, then noninterference says that changing its value (but *not* its label) will have no effect on the output as long as P still holds; in order for P to still hold, we must be changing x to some other *nonzero* value. Hence all nonzero values of x will look the same to an observer. Conditional labels are a novelty of our system; we will see in Section 4 how they can be a powerful tool for verifying the security of a program.

3 Language and Semantics

Our programming language is defined as follows:

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

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:

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{}{\langle \sigma,\ C_1; C_2,\ K\rangle \xrightarrow[l]{} \langle \sigma,\ C_1,\ C_2 :: K\rangle}\ (\text{SEQ})$$

$$\frac{}{\langle \sigma,\ \texttt{skip},\ C :: K\rangle \xrightarrow[l]{} \langle \sigma,\ C,\ K\rangle}\ (\text{SKIP}) \qquad \frac{}{\langle \sigma,\ C,\ K\rangle \xrightarrow[l]{}_0 \langle \sigma,\ C,\ K\rangle}\ (\text{ZERO})$$

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

Fig. 1. Security-Aware Operational Semantics

as they are standard and straightforward. Note that we will sometimes write $\llbracket E \rrbracket \sigma$ as shorthand for $\llbracket E \rrbracket$ applied to the store of state σ.

Figure 1 defines our 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 configurations, 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 that are made within the conditional construct. 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).

Note Two rules of our semantics are omitted here, but can be found in the Coq development [6]. These rules make sure that a low-context execution will diverge safely (rather than get stuck) when it attempts to run a high-context execution that diverges. These rules are necessary for technical reasons, but they ultimately have no significant bearing on our end-to-end noninterference guarantee, since that guarantee only ever mentions terminating executions.

Two of the rules for conditional constructs make use of a function called `mark_vars`. The function `mark_vars`(σ, C) alters σ by setting the label of each variable in `modifies`(C) to `Hi`, where `modifies`(C) is a standard 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 these 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 precondition (according to our generalized noninterference definition described in Section 2).

The operational semantics presented here is mixed-step and manipulates security labels directly. In order to make sense of such a non-standard semantics, we relate it to a standard one that erases labels. We omit the formal definition of this erasure semantics here since it is exactly the expected small-step operational semantics for a simple imperative language. The definition can be found in the technical report and Coq development [6].

The erasure semantics operates on states without labels, and it does not use continuation stacks. Given a state σ 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 τ to range over states without labels. Then the following two theorems hold:

Theorem 1. *Suppose $\langle \sigma, C, [] \rangle \xrightarrow{o}_* \langle \sigma', \texttt{skip}, [] \rangle$ in the instrumented semantics. Then, for some τ, $\langle \bar{\sigma}, C \rangle \xrightarrow{o}_* \langle \tau, \texttt{skip} \rangle$ in the standard semantics.*

Theorem 2. *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, for some σ', $\langle \sigma, C, [] \rangle \xrightarrow{o}_* \langle \sigma', \texttt{skip}, [] \rangle$ 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 instrumented semantics does not get stuck in any execution satisfying the precondition.

4 The Program Logic

In this section, we will present the logic that we use 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 2.

Note We allow assertions to contain logical variables, but we elide the details here to avoid complicating the presentation. In Figure 2, 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, and E may itself contain logical variables. The full details of logical variables can be found in the technical report and Coq development [6].

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

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

Fig. 2. Assertion Syntax and Semantics

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

1. *The operational semantics cannot get stuck when executed from initial configuration* $\langle \sigma,\, C,\, [] \rangle$ *under context* l*.*
2. *If the operational semantics executes from initial configuration* $\langle \sigma,\, C,\, [] \rangle$ *under context* l *and terminates at state* σ'*, then* $\sigma' \in \llbracket Q \rrbracket$*.*

Selected inference rules for our logic are shown in Figure 3. The rules make use of two auxiliary syntactic functions, $\texttt{vars}(P)$ and $\texttt{no\_lbls}(P, S)$ (S is a set of store variables). The first function returns the set of all store variables that appear somewhere in P, while the second checks that for each variable $x \in S$, $x.\texttt{lbl}$ does not appear anywhere in P.

The (IF)/(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

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

$$\frac{}{l \vdash \{\mathtt{emp}\}\, \mathtt{skip}\, \{\mathtt{emp}\}} \text{(SKIP)}$$

$$\frac{}{\mathtt{Lo} \vdash \{\mathtt{lbl}(E) = \mathtt{Lo} \wedge \mathtt{emp}\}\, \mathtt{output}\, E\, \{\mathtt{lbl}(E) = \mathtt{Lo} \wedge \mathtt{emp}\}} \text{(OUTPUT)}$$

$$\frac{x \notin \mathtt{vars}(E')}{l \vdash \{E = E' \wedge \mathtt{lbl}(E) = l' \wedge \mathtt{emp}\}\, x := E\, \{x = E' \wedge x.\mathtt{lbl} = l' \sqcup l \wedge \mathtt{emp}\}} \text{(ASSIGN)}$$

$$\frac{x \notin \mathtt{vars}(E_1) \cup \mathtt{vars}(E_2)}{l \vdash \{x = E_1 \wedge \mathtt{lbl}(E) = l_1 \wedge E \mapsto (E_2, l_2)\}\, x := [E]\, \{x = E_2 \wedge x.\mathtt{lbl} = l_1 \sqcup l_2 \sqcup l \wedge E[E_1/x] \mapsto (E_2, l_2)\}} \text{(READ)}$$

$$\frac{}{l \vdash \{\mathtt{lbl}(E) = l_1 \wedge \mathtt{lbl}(E') = l_2 \wedge E \mapsto \_\}\, [E] := E'\, \{E \mapsto (E', l_1 \sqcup l_2 \sqcup l)\}} \text{(WRITE)}$$

$$\frac{\begin{array}{c} P \Rightarrow B \vee \neg B \qquad B \wedge P \Rightarrow \mathtt{lbl}(B) = l_t \\ \neg B \wedge P \Rightarrow \mathtt{lbl}(B) = l_f \qquad S = \mathtt{modifies}(\mathtt{if}\, B\, \mathtt{then}\, C_1\, \mathtt{else}\, C_2) \\ l_t \sqcup l_f \not\sqsubseteq l \Rightarrow \mathtt{no\_lbls}(P,S) \qquad l_t \sqcup l \vdash \{B \wedge \mathtt{mark\_vars}(P,S,l_t,l)\}\, C_1\, \{Q\} \\ l_f \sqcup l \vdash \{\neg B \wedge \mathtt{mark\_vars}(P,S,l_f,l)\}\, C_2\, \{Q\} \end{array}}{l \vdash \{P\}\, \mathtt{if}\, B\, \mathtt{then}\, C_1\, \mathtt{else}\, C_2\, \{Q\}} \text{(IF)}$$

$$\frac{\begin{array}{c} P \Rightarrow \mathtt{lbl}(B) = l' \\ S = \mathtt{modifies}(\mathtt{while}\, B\, \mathtt{do}\, C) \qquad l' \not\sqsubseteq l \Rightarrow \mathtt{no\_lbls}(P,S) \\ l' \sqcup l \vdash \{B \wedge \mathtt{mark\_vars}(P,S,l',l)\}\, C\, \{\mathtt{mark\_vars}(P,S,l',l)\} \end{array}}{l \vdash \{P\}\, \mathtt{while}\, B\, \mathtt{do}\, C\, \{\neg B \wedge \mathtt{mark\_vars}(P,S,l',l)\}} \text{(WHILE)}$$

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

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

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

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

Fig. 3. Selected Inference Rules for the Logic

```
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
```

Fig. 4. Example: Alice's Private Calendar

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 provide an example program later in this section that illustrates this novelty.

Given our logic inference rules, we can prove the following theorem:

Theorem 3 (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.*

We will not go over the proof of this theorem here since there is not really anything novel about it in regards to security. The proof is relatively straightforward and not significantly different from soundness proofs in other Hoare/separation logics. The primary theorem in this work is the one that says that any verified program satisfies our noninterference property — this will be discussed in detail in Section 5.

4.1 Example: Alice's Calendar

In the remainder of this section, we will show how our logic can be used to verify an interesting example. Figure 4 shows a program that we would like to prove is secure. 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 an event if she is busy. Alice decides that all free time slots in her calendar should be considered low security, while the time slots with events should be secret. This policy allows for others to schedule a meeting time with her, as they can determine when she is available. Indeed, the example program shown here prints out all free time slots.

Figure 5 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

$$P \triangleq \mathop{\ast}_{i=0}^{63} \left(A+i \mapsto (n_i,l_i) \wedge n_i = 0 \iff l_i = \texttt{Lo}\right)$$

```
   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}
```

Fig. 5. Calendar Example Verification

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 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 distinguish between any two events occurring at a particular time slot. This seems like exactly the property Alice would want to have, given that her policy specifies that all free slots are `Lo` and all events are `Hi`.

5 Noninterference

In this section, we will discuss the method for formally proving our system's security guarantee. Much of the work has already been done through careful design of the security-aware semantics and the program logic. The fundamental idea is that we can find a bisimulation relation for our `Lo`-context 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 σ_1 and σ_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:

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

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):

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

This bisimulation semantics operates on configurations consisting of a *pair* of states and a program. As two executions progress step-by-step, the bisimulation semantics makes sure that the executions continue to produce identical outputs and step to identical programs. The semantics requires a `Lo` program counter label because two executions from observably equivalent states may in fact step to different programs when the program counter label is `Hi`.

With this definition of a bisimulation semantics, we can split noninterference into the following progress and preservation properties.

Theorem 4 (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 σ_1'', σ_2'', C'', K'' such that

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

Theorem 5 (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's final value must have a `Hi` label.

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 σ_1 has the same label as the location pointed to by E in σ_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 4. In order to use this rule, the precondition requires us to show that $E \mapsto (n, l_2)$. Since both states σ_1 and σ_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 6 (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 σ_1 to obtain some $\sigma_2 \in [\![P]\!]$. Suppose, in the instrumented semantics, we have*

$$\langle \sigma_1, C, [] \rangle \xrightarrow[\texttt{Lo}]{o_1}_* \langle \sigma_1', \texttt{skip}, [] \rangle \qquad \textit{and} \qquad \langle \sigma_2, C, [] \rangle \xrightarrow[\texttt{Lo}]{o_2}_* \langle \sigma_2', \texttt{skip}, [] \rangle.$$

Then $o_1 = o_2$.

Finally, we can use the results from Section 3 along with the safety guaranteed by our logic to prove the final, end-to-end noninterference theorem:

Theorem 7 (Noninterference, Erasure Semantics). *Suppose we derive* $\mathtt{Lo} \vdash \{P\}\, C\, \{Q\}$ *using our program logic. Pick any state* $\sigma_1 \in [\![P]\!]$*, and consider changing the values of any* `Hi` *data in* σ_1 *to obtain some* $\sigma_2 \in [\![P]\!]$*. Suppose, in the erasure semantics, we have*

$$\langle \bar{\sigma}_1,\, C\rangle \xrightarrow{o_1}_{*} \langle \tau_1,\, \mathtt{skip}\rangle \qquad \text{and} \qquad \langle \bar{\sigma}_2,\, C\rangle \xrightarrow{o_2}_{*} \langle \tau_2,\, \mathtt{skip}\rangle.$$

Then $o_1 = o_2$*.*

6 Related Work

There are many different systems for reasoning about information flow. We will briefly discuss some of the more closely-related ones here.

Some IFC systems with declassification, such as HiStar [26], Flume [11], and RESIN [25], reason at the operating system or process level, rather than the language level. These systems can support complex security policies, but their formal guarantees suffer due to how coarse-grained they are.

On the language-level side of IFC [18], there are many type systems and program logics that share similarities with our logic.

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 system. 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 system, 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 check for all locations in both the store and heap that might be modified within the conditional. With the arbitrary pointer arithmetic of our system, it is not possible to syntactically bound which heap locations will be written to, so we require the additional semantic technique described in Section 5 that involves enforcing a side condition on the bisimulation semantics.

– Our model of observable behavior provides some extra leniency in verification. Our system allows bad leaks to happen within the program state, so long as these leaks are not made observable via an output command. In their system (and most other IFC systems), the enforcement mechanism must prevent those leaks within program state from happening in the first place.

Banerjee et al. [3] 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 known in the literature as "gradual release"). 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 4 would not be verifiable in their system.

Delimited Release [19] is an IFC system that allows certain prespecified expressions (called *escape hatches*) to be declassified. For example, a declassification policy for high-security variable h might say that the expression $h\%2$ should be considered low security. Relaxed Noninterference [12] uses a similar idea, but builds a lattice of semantic declassification policies, rather than syntactic escape hatches — e.g., h would have a policy of $\lambda x \, . \, x\%2$. Our system can easily express any policy from these systems, using a precondition saying that some low-security data is equal to the escape hatch function applied to the secret data. Our strong security guarantee is identical to the formal guarantees of both of these systems, saying that the high-security value will not affect the observable behavior as long as the escape hatch valuation is unchanged.

Relational Hoare Type Theory (RHTT) [15] is a logic framework for verifying information-flow properties. It is based on a highly general relational logic. The system can be used to reason about a wide variety of security-related notions, including declassification, information erasure, and state-dependent access control. While RHTT can be extremely expressive, it seems to achieve different goals than our system. We began with a desire to formally reason about the propagation of security labels through a system, and to specify declassification policies in terms of these labels. Thus the natural choice for us was to use a syntactic representation for labels and explicitly add them into program state. Ideally, we could use these labels to represent different principals, and thus be able to specify interesting policies in a decentralized setting. RHTT, on the other hand, is built around a semantic notion of security labels. Instead of saying that x is `Lo`, one says that in two corresponding states, x has the same value. This allows policies to be highly expressive, but also can make it quite difficult to understand what a given policy is saying. It is unclear how one should go about representing a decentralized setting in RHTT, where there is interaction between the data of various principals.

Intransitive noninterference [13] is a declassification mechanism whereby certain specific downward flows are allowed in the label lattice. The system formally verifies that a program obeys the explicitly-allowed flows. These special flows are intransitive — e.g., we might allow Alice to declassify data to Bob and Bob to declassify to Charlie, but that does not imply that Alice is allowed to declassify to Charlie. The intransitive noninterference system is used to verify simple imperative programs; their language is basically the same as ours, except without the heap-related commands. One idea for future work is to generalize our state predicate P into an action G that precisely describes the transformation that a program is allowed to make on the state. If we implemented this idea, it would be easy to embed the intransitive noninterference system. The action G would specify exactly which special flows are allowed (e.g., the data's label can be changed from Alice to Bob or from Bob to Charlie, but not from Alice to Charlie directly). Ideally, we would have a formal noninterference theorem in terms of G that would give the same result as the formal guarantee in [13].

Another related system is Chong and Myers' lambda calculus with downgrading policies [5]. This system shares a similar goal to ours: to provide an end-to-end security guarantee relative to declassification policies. In their system, the language contains explicit declassify commands, and certain conditions (specified via the policy syntax) must be verified at the point of declassification. The actual method for verifying these conditions is left as a parameter of the system. While both of our systems prove an end-to-end guarantee, these guarantees seem to be rather different. Theirs provides a road map describing how a `Hi` piece of data may end up affecting observable behavior, while ours specifies which values that piece of data could have in order to affect observable behavior. It is unclear how either guarantee would be described in the other system.

Self-composition [4] is an approach to noninterference reasoning that essentially converts relational predicates into unary ones. The fundamental idea is that we can prove a program C is noninterfering by making a copy of it, C', giving all variables in C' a fresh name, and then executing the composed program $C; C'$. A pre/postcondition for $C; C'$ of $x = x'$, for example, will then effectively say that x is a low-security location. Our system, just like the self-composition approach, is based on the desire to deal with unary predicates rather than relational ones. Unlike the self-composition approach, however, we use a syntactic notion of labels, and we do not perform any syntactic translations on the program. Additionally, it is unclear whether the self-composition approach can be used for programs that do odd things with pointer arithmetic and aliasing. Indeed, in [4], there is an explicitly-stated assumption that the address values of pointers do not affect the control flow of programs.

All of the language-based IFC systems mentioned so far, including our own system, use static reasoning. There are also many dynamic IFC systems (e.g., [2, 10, 22, 24]) 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, our final noninterference theorem uses the erasure semantics, meaning that there is no overhead whatsoever.

Finally, Sabelfeld and Sands [21] 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 notion of declassification can talk about any of these dimensions if we construct the precondition in the right way. The *who* dimension is most naturally handled via the label lattice, but one could also imagine representing principals explicitly in the program state and reasoning about them in the logic. The *what* dimension is handled by default, as the program state contains all of the data to be declassified. The *when* dimension can easily be reasoned about by including a time field in the state. Similarly, the *where* dimension can be reasoned about by including an explicit program counter in the state.

7 Conclusion

In this paper, we described a novel program logic for reasoning about information flow in a low-level language with pointer arithmetic. Our system uses an instrumented operational semantics to statically reason about the propagation of syntactic labels. Our logic can reason about labels conditioned on state predicates — as far as we are aware, the example program of Section 4 cannot be verified in any other IFC system that uses syntactic labels.

In the future, we hope to extend our work to handle termination-sensitivity, dynamic memory allocation, nondeterminism, and concurrency. We also plan to develop some automation and apply our logic to actual operating system code.

8 Acknowledgments

This research is based on work supported in part by NSF grant 1065451, and DARPA grants FA8750-10-2-0254 and FA8750- 12-2-0293. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

1. T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In *POPL*, pages 91–102, 2006.
2. T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *PLAS*, pages 113–124, 2009.
3. A. Banerjee, D. A. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. In *IEEE Symposium on Security and Privacy*, pages 339–353, 2008.
4. G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *CSFW*, pages 100–114, 2004.
5. S. Chong and A. C. Myers. Security policies for downgrading. In *ACM Conference on Computer and Communications Security*, pages 198–209, 2004.
6. D. Costanzo and Z. Shao. A separation logic for enforcing declarative information flow control policies. Technical report, Dept. of Computer Science, Yale University, New Haven, CT, Jan. 2014. `http://flint.cs.yale.edu/publications/ddifc.html`.
7. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.
8. N. Heintze and J. G. Riecke. The slam calculus: Programming with secrecy and integrity. In *POPL*, pages 365–377, 1998.
9. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
10. C. Hritcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett. All your ifcexception are belong to us. In *IEEE Symposium on Security and Privacy*, pages 3–17, 2013.
11. M. N. Krohn, A. Yip, M. Z. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard os abstractions. In *SOSP*, pages 321–334, 2007.
12. P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *POPL*, pages 158–170, 2005.
13. H. Mantel and D. Sands. Controlled declassification based on intransitive noninterference. In *APLAS*, pages 129–145, 2004.
14. A. C. Myers and B. Liskov. A decentralized model for information flow control. In *SOSP*, pages 129–142, 1997.
15. A. Nanevski, A. Banerjee, and D. Garg. Verification of information flow and access control policies with dependent types. In *IEEE Symposium on Security and Privacy*, pages 165–179, 2011.
16. P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL*, pages 1–19, 2001.
17. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.
18. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
19. A. Sabelfeld and A. C. Myers. A model for delimited information release. In *ISSS*, pages 174–191, 2003.
20. A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. In *ESOP*, pages 40–58, 1999.
21. A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.

22. D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in haskell. In *Haskell*, pages 95–106, 2011.
23. D. M. Volpano and G. Smith. A type-based approach to program security. In *TAPSOFT*, pages 607–621, 1997.
24. J. Yang, K. Yessenov, and A. Solar-Lezama. A language for automatically enforcing privacy policies. In *POPL*, pages 85–96, 2012.
25. A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *SOSP*, pages 291–304, 2009.
26. N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. In *OSDI*, pages 263–278, 2006.