COVERN: A Logic for Compositional Verification of Information Flow Control

Toby Murray*‡, Robert Sison†‡ and Kai Engelhardt†‡
*School of Computing and Information Systems, University of Melbourne, Australia
Email: toby.murray@unimelb.edu.au
†School of Computer Science and Engineering, UNSW Sydney, Australia
Email: {robs,kaie}@cse.unsw.edu.au
‡Data61 (formerly NICTA), CSIRO, Australia

***Abstract*—Shared memory concurrency is pervasive in modern programming, including in systems that must protect highly sensitive data. Recently, verification has finally emerged as a practical tool for proving interesting security properties of real programs, particularly information flow control (IFC) security. Yet there remain no general logics for verifying IFC security of shared-memory concurrent programs. In this paper we present the first such logic, COVERN (Compositional Verification of Noninterference) and its proof of soundness via a new generic framework for *general* rely-guarantee IFC reasoning. We apply COVERN to model and verify the security-critical software functionality of the Cross Domain Desktop Compositor, an embedded device that facilitates simultaneous and intuitive user interaction with multiple classified networks while preventing leakage between them. To our knowledge this is the first foundational, machine-checked proof of IFC security for a non-trivial shared-memory concurrent program in the literature.**

1. Introduction

Concurrency is ubiquitous in modern programming. Shared memory concurrency in particular remains an indispensable programming paradigm, whether in traditional programming languages like C and Java or in embedded systems, which are often built as a collection of small concurrently executing tasks or *components* that communicate via shared memory and primitives provided by a minimal operating system (OS) kernel.

For programs that handle sensitive data—a category that is becoming ever larger—formal proof of *information flow control* (IFC) security remains a gold standard of assurance that they do not leak sensitive data to attackers. Recent advances have seen entire microkernels such as seL4 [25] and mCertiKOS [11] proved IFC secure, as well as conference management systems like CoCon [18] and social media platforms like CoSMed [4] amongst others. However, none of these systems—with the exception of CoSMeDis [5], the recently-verified distributed incarnation of CoSMed—are concurrent, and none exhibit shared memory concurrency. Whereas program logics for proving ordinary functional correctness in the presence of shared memory concurrency have seen great advances in the past decade [10], [13], [20], [30], logics for proving IFC security have lagged comparatively behind. Indeed, to our knowledge, there exist no general purpose logics for proving IFC security of shared memory concurrent programs.

We remedy this shortcoming in this paper by presenting the first such logic, COVERN (Compositional Verification of Noninterference) and its proof of soundness via a generic framework for *general* rely-guarantee IFC reasoning. We apply COVERN to model and verify the security-critical functionality of the seL4-based software implementation of the Cross Domain Desktop Compositor [6] (CDDC), an embedded device that facilitates simultaneous and intuitive user interaction with multiple classified networks while preventing leakage between them. To our knowledge this is the first foundational, machine-checked proof of IFC security for a non-trivial shared-memory concurrent program in the literature.

One might argue that the lack of general logics for proving IFC security of concurrent programs is unsurprising: for interesting programs, IFC security rests on their functional correctness, and proving IFC cannot be done without also proving that such programs are functionally correct. This is especially true for programs that handle data whose sensitivity is *data-dependent* [1], [22], [26], [28], [36], [37], [39], [40]. When component B receives from component A data d whose sensitivity is dictated by some data d' (whether the current access control policy [25], scheduler state [26], mode of operation [24] or a label embedded in the data d itself [22]), B must correctly inspect d' if it is to handle d securely. Thus B's security crucially depends on its correct functioning, and so cannot be reasoned about using (only) traditional IFC reasoning techniques like security type systems [9], [23], [34], [38] that focus almost exclusively on program structure.

In this sense, proving IFC security can be at least as difficult as proving ordinary functional correctness. Carefully reasoning about concurrent programs, and doing so *compositionally* (i.e. one-component-at-a-time), is necessarily complicated by the fact that reasoning performed about one component can be invalidated by the actions (e.g. modifications to shared memory) of the other components against which it executes concurrently [31]. After executing

the assignment $x := x + 1$ shared variable x's value has increased (modulo overflow) only if no other component has decreased it in the meantime. Compositional reasoning therefore requires verifying each component under assumptions that it makes about the others (e.g. they won't decrease x's value), on whom it *relies* to abide by those assumptions. Such reasoning is sound only when each component provably *guarantees* to abide by the assumptions made about it by all others. This observation forms the essence of *rely-guarantee* reasoning [17], which (implicitly) underpins [14] many compositional concurrent program logics like concurrent separation logics [10], [30] (CSLs).

CSLs have become the dominant method for compositional reasoning about functional correctness of concurrent programs. When used to reason about *well-synchronised* locking programs, namely those in which locks are used to protect access to shared data, basic CSLs operate as follows. At all times, each lock is associated with a unique region of the shared memory that it protects, via a *shared data invariant* which can be thought of as a shared contract on the protected data. Acquiring the lock not only grants access to the data it protects, but also allows one to *rely* on the invariant holding on the data. When releasing the lock, one must *guarantee* that the invariant has been re-established (if it was violated in the meantime). Thus these data invariants serve as contracts on data that passes between concurrently executing components. For example, the functional correctness of a component that computes on shared integers a and b might rest on b being non-zero (perhaps because it computes a/b). Thus $b \neq 0$ forms a natural shared data invariant, without which we cannot prove the component functionally correct.

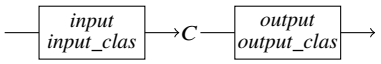
While necessary, shared data invariants on their own are insufficient for verifying IFC security of concurrent programs. When gaining access to shared data a component must rely not only on the data having a correct *value* but also having the expected *sensitivity* (which may depend on other data, as noted above). In our program logic COVERN, each shared variable (memory location) is assigned a *value-dependent classification*, which should be thought of as a shared contract specifying the sensitivity of the data held by the variable in terms of the values of other program variables. When acquiring access to a shared variable (by acquiring the lock that protects it), one can assume that its sensitivity matches its value-dependent classification. Likewise, when releasing access to the variable, one is obliged to show that the sensitivity of the data it (now) contains is consistent with the variable's classification. Thus value-dependent variable classifications form the *security* analogue of data invariants and, when paired with them, become a powerful tool for describing and enforcing inter-component security contracts on shared data.

COVERN and the general rely-guarantee framework on which it is built and proved sound are formalised in Isabelle/HOL [29].[1] The resulting metatheory alone constitutes over 10,000 lines of proof script, not including a set of custom tactics that ease its application. It operates similarly to a traditional CSL by reasoning forward over the program text. Like other CSLs [2], [3] it sometimes requires the user to supply loop invariants and in some cases to supply a common postcondition at the join point of if-statements. At present it operates over a simple imperative language without pointers and arrays, and so does not feature the traditional maps-to predicate "$\mapsto$" of CSL nor separating conjunction "$*$". Adding these should be relatively straightforward, as the underlying rely-guarantee framework is already sufficient to talk about them. However, doing so is left as future work.

1. The Isabelle theories are available at http://covern.org.

lock(ℓ);
output := *input*;
unlock(ℓ)

(a) Component code.



(b) Component architecture.

Figure 1: A component C that copies labelled data. Lock ℓ protects four variables: *input*, *output*, *input_clas*, and *output_clas*, the latter two of which are not read or written in the example component. The classifications of the former two depend respectively on the values of the latter two. This component is therefore secure only when *output_clas* = *input_clas*.

This paper is organised as follows. Section 2 provides a high-level overview of COVERN and underlying rely-guarantee framework. Section 3 then describes aspects of the rely-guarantee framework in detail, including the choice of compositional noninterference property that it establishes. Section 4 discusses the logic and its proof of soundness atop the rely-guarantee framework of Section 3, before Section 5 describes its application to the modelling and verification of the CDDC. Section 6 summarises related work before we conclude in Section 7.

2. Overview

Before introducing the logic COVERN, we first informally motivate and describe the main ideas it embodies, and how they interact to allow rich compositional verification of IFC for concurrent programs.

2.1. Value-Dependent Classifications as Contracts

Figure 1a depicts a tiny but nonetheless illustrative component in the simple imperative language of COVERN. Imagine that this component runs alongside others. Its job is to take input data from the shared variable *input* and copy it to the shared variable *output*. Other components in the system might modify these shared variables, so they are protected by a lock ℓ. Two other variables *input_clas* and *output_clas* (not shown in the code in Figure 1a) are also protected by the lock ℓ. These variables are used to *label* the sensitivity of the data in *input*, respectively, *output*. For example, when some other component in the system reads the output from *output*, it can inspect *output_clas* to learn the sensitivity of the data it has read in order to handle it appropriately. Thus the two pairs of variables (*input*,

input_clas) and (*output*, *output_clas*) should be thought of as *labelled* input, respectively, output channels to the component *C* shown in Figure 1. This architecture is depicted in Figure 1b.

We formally define the association between *input* and *input_clas* (and similarly for *output* and *output_clas*) by assigning *input* a *value-dependent classification*. A variable's *classification* defines the maximum sensitivity of the data that it is allowed to contain. A *value-dependent* classification can refer to (i.e. may depend on the values of) other program variables, allowing its valuation to change dynamically. Suppose for simplicity that there exist just two sensitivity levels Low (for non-sensitive data) and High (for sensitive data that should not be revealed to attackers). Suppose further for concreteness that *input*'s classification is defined as being Low when *input_clas* is zero and as High otherwise, and likewise for *output* and *output_clas*. Then at all times e.g. *output_clas* unambiguously indicates the (maximum) sensitivity of the data in *output*.

Indeed, because all components in the system must respect these value-dependent classifications, they form natural *security contracts* on the sensitivity of the data contained in the *input* and *output* variables. If e.g. some other component wishes to update *input*'s value with a new one that is more sensitive than *input*'s current classification, it must also update *input_clas* accordingly, to reflect *input*'s new sensitivity.

2.2. Data Invariants for Compositional Security

So what then of the example component's security? It blindly copies data from *input* to *output*, heedless of the data's sensitivity and the current classifications of these variables. In particular, if *input_clas* is nonzero (meaning that *input* might currently hold High data) but *output_clas* is zero (meaning that *output* is currently permitted to hold only Low data), the component will cause a security violation: a downstream component that subsequently reads both *output* and *output_clas* will erroneously conclude that *output* holds only non-sensitive data, and so might write it to an attacker-observable public output channel. This potential security violation is manifest here in the form of a violation of *output*'s security contract.

In fact, the security of the component in Figure 1 is predicated on the assumption that $output\_clas = input\_clas^2$. This assumption is a trivial data invariant, which the component implicitly assumes is true having acquired the lock ℓ.

COVERN allows one to associate these kinds of data invariants with locks, so that the lock is said to protect not only a set of shared variables but also some relationship between them. In the example in Figure 1, we associate the invariant $output\_clas = input\_clas$ with lock ℓ. This allows the invariant to be *relied upon* once lock ℓ is acquired, a *quid pro quo* in exchange for *guaranteeing* that the invariant holds at the point at which the lock is later released (thus allowing the next component who acquires the lock to rely on the invariant).

Thus data invariants are vital to compositionally verifying rich concurrent programs, stating basic contracts on shared data, in concert with value-dependent classifications, which serve as *security contracts* on shared data.

Verifying a concurrent program proceeds by first defining (usually static) classifications for its data sources and sinks. Sources carrying sensitive inputs to the system that should not be revealed to attackers must be classified High. Sinks that are observable to potential attackers conversely must be classified Low. This assignment defines the system's overall security policy. Verifying that its (concurrent) operation indeed adheres to (or enforces) that policy then proceeds by defining (often value-dependent) classifications for the shared variables (like *input*, *input_clas*, *output* and *output_clas*) that form the interfaces *between* the concurrently-executing components of the system, along with appropriate data invariants. If all components can be shown to respect all such contracts, the system can be said to be secure. COVERN is the first such logic to soundly capture these intuitions and allow this style of verification.

2.3. An Overview of COVERN

COVERN derives inspiration both from CSLs (especially the idea of associating shared variables and invariants with locks) as well as from recent work on dependent security type systems [27]. Like both, COVERN is geared towards forwards reasoning over each component. Like all CSLs, but unlike *any* previous rely-guarantee based security type systems [23], [27], once each component has been verified, no further work is required to conclude that the entire system (i.e. all components running in parallel) is IFC secure.

COVERN allows locks to be used to protect variables to differing degrees. For each variable v protected by a lock ℓ, the logic distinguishes between whether v is merely protected from being modified by other components while ℓ is held, or whether it is additionally protected from being read. Consider the *output* variable in the example of Figure 1. Suppose this variable represents an external data sink. In that case it might not be reasonable to assume that a potential attacker won't be able to observe values written to this variable. In this case, one would want to specify that lock ℓ protects this variable from being modified while the lock is held but not from being read. COVERN will then require that all values written to the *output* variable while the lock is held match its security contract. On the other hand, if *output* represents an internal channel between this component and another that will also be verified, then it might be reasonable to say that acquiring lock ℓ also prevents other components from reading this variable (which is not visible to attackers). In that case COVERN allows any data to be written to the variable while the lock is held, but at the time that the lock is released one must show that whatever data remains in the variable is consistent with its contract.

COVERN assumes a fixed finite set of locks *Lock* and shared variables *Var*, and a static mapping *lockvars* from

2. Technically, $output\_clas = 0 \longrightarrow input\_clas = 0$ is sufficient, but we use the stronger equality for simplicity of exposition.

locks to shared variables such that for each lock $\ell \in Lock$, $lockvars(\ell)$ defines two mutually non-overlapping sets of variables (NW, NRW) that are protected by ℓ. Those in NW ("no write") are protected from being modified but not from being read, while those in NRW ("no read-or-write") are also protected from being read. COVERN requires each shared variable to be protected by a unique lock. Invariants are associated with locks by the function *lockinv* that returns a predicate given a lock ℓ, such that $lockinv(\ell)$ mentions only variables from $lockvars(\ell)$ (and so preventing the absurdity of a lock attempting to protect an invariant over a variable that it does not protect). In the example of Figure 1, we arbitrarily choose the lock ℓ to protect both variables from being both written to and read from, so we have: $lockvars(\ell) = (\{\}, \{input, input\_clas, output, output\_clas\})$ and $lockinv(\ell) = (output\_clas = input\_clas)$.

COVERN adapts [27]'s trick for encoding value-dependent classifications. For the two-point lattice of security levels, $\{\mathsf{Low}, \mathsf{High}\}$ in which $\mathsf{Low} \sqsubseteq \mathsf{High}$ but $\mathsf{High} \not\sqsubseteq \mathsf{Low}$ (meaning that High sensitivity information should not flow to Low-classified sinks), value-dependent classifications on shared variables $v \in Var$ are defined by a function $\mathcal{L}$, such that $\mathcal{L}(v)$ gives a predicate P. When P holds, the classification is Low; otherwise it is High.[3] Variables such as *input_clas* and *output_clas* which determine the classifications of other variables must be statically classified Low, to prevent covert channels by changes to variable classifications. Thus in the example of Figure 1, $\mathcal{L}(input\_clas) = \mathcal{L}(output\_clas) = \mathsf{true}$ (the predicate that always holds), while $\mathcal{L}(input) = (input\_clas = 0)$ and similarly $\mathcal{L}(output) = (output\_clas = 0)$.

COVERN tracks three pieces of information about the local state of each component as it reasons over the component's program text. We denote these three pieces of information Γ, L, and P. Like all CSLs, P is a predicate that tracks what is currently true about the variables to which the component has acquired access (by acquiring appropriate locks). L in turn tracks the set of locks that have been acquired, while Γ tracks the sensitivity of the data contained in shared variables to which the component has acquired access, and is a partial function from *Var* to predicates (i.e. from shared variables to value-dependent classifications). Thus program judgements in COVERN take the form:

$$\Gamma, L, P \ \{c\} \ \Gamma', L', P' \tag{1}$$

where c is a fragment of program text, Γ, L, and P capture the state before c executes (a concurrency and security-aware local precondition), while their counterparts Γ', L', and P' capture the state after c has terminated (a corresponding postcondition). Intuitively Equation 1 means that: (1) if program c is executed from a state in which each shared variable x tracked in Γ contains data whose sensitivity is $\Gamma(x)$, and (2) if each of the locks in the set L are currently held by the component, and (3) if the predicate P is true of the shared memory, then (4) if and when c terminates, (5) each shared variable y tracked in Γ' will have sensitivity $\Gamma'(y)$, (6) each lock in the set L' will be held by the component and (7) predicate P' will hold.

Reasoning over each component starts knowing nothing: Γ tracks no variables, L is empty and P is true. Each of these is updated as reasoning proceeds forward over the text of the component. When a lock ℓ is acquired, L is naturally updated to become $L \cup \{\ell\}$. However the logic also augments Γ and P under the assumption that all contracts on all shared variables $x \in lockvars(\ell)$, protected by the lock, hold. For each variable $x \in lockvars(\ell)$, this means updating Γ to assume that x's sensitivity matches its value-dependent classification, and so updating Γ to become $\Gamma[x \mapsto \mathcal{L}(x)]$. In the style of CSLs it also means updating P to assume the invariant $lockinv(\ell)$ protected by the lock ℓ, updating P to $P \wedge lockinv(\ell)$.

In the example of Figure 1, after lock ℓ is acquired the sensitivity of each variable is assumed to match its classification: $\Gamma(input) = (input\_clas = 0)$ and $\Gamma(output) = (output\_clas = 0)$; lock ℓ is known to be acquired: $\ell \in L$; and the lock's invariant is assumed to hold: $(output\_clas = input\_clas)$ is a conjunct of P.

At the point at which a lock ℓ is released, COVERN requires one to prove that the values in all variables $x \in lockvars(\ell)$ protected by the lock match their contracts. This means proving that (1) the sensitivity of the data in the variable $\Gamma(x)$ is no greater than the variable's value-dependent classification $\mathcal{L}(x)$ (under the updated predicate P that holds after the lock is released), and (2) that any data invariants protected by the lock also hold (i.e. are implied by P at the point at which the lock is released).

In the example of Figure 1, when lock ℓ is released, *output* is known to hold data whose sensitivity is that of *input* (due to the preceding assignment statement), and so $\Gamma(output) = (input\_clas = 0)$. Of course the lock invariant still holds, since neither *input_clas* nor *output_clas* have been modified while the lock has been held, so $(output\_clas = input\_clas)$ is a conjunct of P. Under this equality, *output*'s sensitivity can be expressed equivalently as $\Gamma(output) = \{output\_clas = 0\}$, in which case it is equal to its value-dependent classification and so clearly consistent with it. COVERN expresses this kind of reasoning by allowing entries in Γ to be rewritten using information in P, an idea we borrow from [27].

While not shown in this example, COVERN contains rules for reasoning over if-statements, while-loops, and assignments to variables like *input_clas* and *output_clas* that define the classifications of other variables. We give a sample of these rules later in Section 4.

COVERN is defined for a language in which we use shared memory and locks as the sole means of communication between components. Remote Procedure Call (RPC) style communication can be modelled straightforwardly using locks and shared memory, as can buffers/channels. Indeed when modelling and verifying the software of the CDDC (discussed later in Section 5), we did exactly that to reason about inter-component communications implemented over seL4 Inter-Process Communication (IPC).

3. This can be extended to an arbitrary lattice by replacing these predicates with functions that return an element of the lattice.

```
lock(ℓ);
low := 0;
while high > 0 do
    high := high - 1;
low := 1;
unlock(ℓ)
```

(a) Timing insensitive secure.

```
output := low
```

(b) Timing insensitive secure.

Figure 2: Timing insensitive IFC is not compositional.

Before describing COVERN in detail in Section 4, we first discuss the security property it establishes, in the next section. This security property is a compositional noninterference property that supports general rely-guarantee reasoning. Such a general property is needed to express the kinds of reasoning that COVERN performs in which variables can carry not only security contracts (via value-dependent classifications) but also functional correctness contracts (via data invariants). While rely-guarantee frameworks for noninterference have seen some recent attention (beginning with the seminal work of [23]), all so far have supported only very limited forms of rely-guarantee reasoning insufficient to justify the preservation of data invariants across lock release and subsequent re-acquisition. Nonetheless, many of the ideas in the following section are heavily inspired by existing work on which we build, chiefly that of [23], [24].

3. General Rely-Guarantee Reasoning for IFC

3.1. Defining IFC Security

COVERN reasons separately over each component of a concurrent system in order to prove that the entire concurrent composition is secure. To do so it establishes a *compositional* security property: one that if true for each component of the system establishes that the system as a whole is also secure.

The security property that COVERN proves is a *timing sensitive* version of noninterference [16].[4] Such a property not only requires that the values written to Low-classified variables do not depend on High sensitivity data, but also that the times at which such Low variables are updated do not depend on High data. In this sense, timing sensitive security is a relatively stringent security property.

Consider the sequential program in Figure 2a. Suppose the variables *low* and *high* are classified Low and High respectively. This program satisfies timing insensitive IFC security, since the values written to the Low-classified variable *low* never depend on High data. However, it does not satisfy timing sensitive security, since *when* the second write to *low* is performed clearly depends on High data.

Now consider the trivial program in Figure 2b and suppose that *output* represents a publicly observable output channel of the system. On its own this program is obviously secure under any definition since it is guaranteed to copy only Low data to the public output channel. However suppose now that it runs concurrently alongside the code in Figure 2a under a pre-emptive scheduler. Then the value it is likely to write to the public output channel can be influenced by High data: when *high*'s initial value is small, the value 1 is more likely to be written to *output*; when *high*'s initial value is very large, 0 is more likely to be written to *output*.[5] Thus even though both programs are timing-insensitive secure on their own, their parallel composition is not.

The fact that timing-insensitive security is not compositional has been known since at least the late 1990s [38]. We recapitulate it here, however, to make it clear why we require such a strong security property.

We can phrase timing-sensitive security as follows. In this paper we give definitions in terms of concepts already introduced in Section 2, namely locks for controlling assumptions about the possible actions of other components, in order to ease exposition. In reality our rely-guarantee framework is phrased generically and could be instantiated with any other mechanism for controlling interactions between the concurrently executing components. We refer the reader to our Isabelle/HOL formalisation for the low level details.

We define the potential observations that an attacker[6] might be able to make of the shared program memory *mem* by defining an *indistinguishability relation* that relates all pairs of memories mem_1 and mem_2 that the attacker cannot distinguish. In our setting the attacker can observe all Low variables except those currently protected from being read because an appropriate lock is currently acquired. We also consider all locks as implicitly statically classified as Low since they naturally influence execution timing amongst threads that contend on them. This is captured as follows.

Definition 1 (Low Equivalence). Two memories mem_1 and mem_2 are said to be *low equivalent* (written $mem_1 \simeq mem_2$) if the same set of locks are currently acquired in both and if $mem_1(x) = mem_2(x)$ for all Low-classified variables x, except those that some component is currently assuming will not be read (because it has acquired some lock ℓ for which $x \in NRW$, where $lockvars(\ell) = (NW, NRW)$).

Timing-sensitive IFC security can then be defined thus.

Definition 2 (Timing-Sensitive Security). Program c is *timing-sensitive secure* if, for all n, when executed from any initial memory memory mem_1 for n execution steps, it results in some memory mem_1', then a corresponding execution from an initial memory mem_2 satisfying $mem_1 \simeq mem_2$ can also run for n steps to reach a memory mem_2' satisfying $mem_1' \simeq mem_2'$.

Observe that the program in Figure 2a doesn't satisfy this definition, since if it runs from two initial low equivalent memories, the time at which the second assignment to *low*

4. We discuss the issue of source level reasoning about execution time later in Section 6.

5. A probabilistic argument is not necessary; one can also consider all deterministic schedulers.

6. An attacker here could be an external adversary observing the system or an internal component that might leak secrets to an external adversary. From the perspective of compositional reasoning, these are largely the same.

will occur can differ and at this point of difference, having run for the same number n of execution steps, the resulting memories will not be low equivalent.

3.2. Proving IFC Security

How does COVERN establish this kind of property for each component? What kind of evidence must be produced to show that timing sensitive security holds? We follow the prior work of [23], [24] which we augment to support general rely guarantee reasoning. At its core, we need to show that if two executions beginning from low equivalent memories take one step each, then the resulting memories are still low equivalent, and then if they take a further step each, the resulting memories remain low equivalent, and so on. To do so, we must establish a *relational invariant* $\mathcal{B}$ that relates states between two executions of the program (beginning from low equivalent memories), that is always preserved after both make a single execution step, and always guarantees low equivalence. Then timing sensitive noninterference follows by induction on the number of execution steps n from Definition 2. The rules of COVERN, which we present in Section 4, effectively establish such an invariant: the soundness proof for the logic is a constructive proof that such a $\mathcal{B}$ exists whenever the rules are used to prove that a component is indeed secure.

Let s denote the semantic state of a component. This includes both the component's local state (e.g. program text remaining to be executed) as well as the shared memory to which all components have access, which we denote $mem(s)$. Following [23] we refer to these relational invariants $\mathcal{B}$ as *strong low bisimulations*, defined as follows.

Definition 3 (Strong Low Bisimulation). A symmetric relation $\mathcal{B}$ on pairs of component states (s_1, s_2) is called a *strong low bisimulation* when it (1) relates all low equivalent initial states, (2) is preserved on a single execution step, so that if $s_1 \mathcal{B} s_2$ and s_1 makes a single execution step then s_2 can also make a single execution step so that they respectively result in states s_1' and s_2' for which $s_1' \mathcal{B} s_2'$, and (3) guarantees low equivalence, so that for all states $s_1 \mathcal{B} s_2$ we have $mem(s_1) \simeq mem(s_2)$.

3.3. General Rely-Guarantee Reasoning

A strong low bisimulation can be thought of as the noninterference analogue of an ordinary concurrent program invariant. If we have established such an invariant for each component of our concurrent program, how do we know that when we run the components in parallel that they won't violate each other's invariants? Doing so would invalidate the reasoning about each of them, leading to potential insecurity in the concurrent program. As a trivial example, suppose one component contains code like the following:

$x := 0$; **if** $x \geq 0$ **then** $low := 0$ **else** $low := high$

This code sets shared variable x to zero, but then might behave insecurely if x has negative value when examined in the following if-statement. An invariant that states that $x \geq 0$ at the time of the test of the if-statement is certainly valid for this program in isolation, and would be sufficient to prove it secure in isolation. However, if the program is run concurrently with another that decreases the value of x in between when it is assigned and subsequently tested, then the invariant will be violated, leading to potential insecurity.

Traditional rely-guarantee reasoning [17] deals with this problem as follows. It requires verifying the component under a *rely condition* R (which may change throughout the program's execution). Such a condition is simply a (reflexive, transitive) relation on pairs of memories *mem* and *mem′*. Rather than relating pairs of memories from two program executions, however, a rely condition relates memories from a single execution and encodes assumptions on how other components in the system might alter the memory. In particular, for any such alteration from memory *mem* to produce a memory *mem′*, the assumption is that *mem R mem′*. So if a component is to be verified under the assumption that other components in the system will never decrease the value of shared variable x, this can be encoded by the rely condition R that relates all pairs of memories *mem* and *mem′* for which $mem'(x) \geq mem(x)$. To prove that some invariant established about a component won't be violated by the actions of other components, it then suffices to show that if the invariant holds for any memory *mem*, it also holds for any memory *mem′* for which *mem R mem′*.

How do we prove that a component's rely condition is sound, i.e. that it will be adhered to by the other components? Rely-guarantee reasoning does so by verifying each component not only under a rely condition R capturing its current assumptions about how the other components in the system might modify the shared memory, but also a *guarantee* condition G. Like R, G is a relation on memories *mem* and *mem′*, and describes a guarantee that the component that is being verified is making about its own behaviour, namely that its own modifications to the memory *mem* to produce memory *mem′* are related by G. For example, to guarantee that shared variable x won't be modified, a component could use the guarantee relation G that relates all memories *mem* and *mem′* for which $mem'(x) = mem(x)$. Naturally, when verifying the component we must show that each step of its execution adheres to its guarantee relation. In this paper we call this *guarantee compliance*.

This kind of reasoning composes soundly, of course, only when at all times, for each pair of components, one's guarantee condition G is no weaker than the other's rely condition R. In the above example, the guarantee condition stating that x won't be modified is naturally stronger than the rely condition that its value will not be decreased, and so is sufficient to establish it. In this case we say that the rely and guarantee conditions are *compatible*. Sound rely-guarantee reasoning requires that at all times each pair of components' rely and guarantee conditions are compatible with each other.

3.4. General Rely-Guarantee for IFC

Our rely guarantee framework applies these same general ideas to produce a compositional timing-sensitive nonin-

terference condition. Unlike prior rely-guarantee work for noninterference, ours is the first to incorporate arbitrary rely and guarantee relations R and G as described above. Such is needed to encode the preservation of data invariants in between when a lock is released and then subsequently acquired; we couldn't prove COVERN sound without them.

Specifically, our framework associates rely and guarantee conditions R and G with each component, that can change throughout its execution. We then require that (1) each component's strong low bisimulation $\mathcal{B}$ is preserved under the component's rely condition, that (2) its own execution adheres to its current guarantee condition G, and that (3) rely and guarantee conditions remain compatible between components when they are run in parallel. Under these requirements, timing sensitive security can be shown to be compositional.

We augment the semantic state s of each component with ghost information that includes rely and guarantee conditions, which we denote $R(s)$ and $G(s)$ respectively. We strengthen the definition of low equivalence, $s_1 \simeq s_2$, to assert that $R(s_2) = R(s_1)$ and $G(s_2) = G(s_1)$ (to avoid the paradox in which an apparently secure component is allowed to change its assumptions or guarantees about other components based on High information). We then require that the component's strong low bisimulation $\mathcal{B}$ is *closed* under its rely condition, defined as follows, which formally captures what it means for $\mathcal{B}$ to be preserved under R.

Definition 4 (Closed Under Rely Condition). A component's strong low bisimulation $\mathcal{B}$ is *closed under its rely condition* when for any states s_1 and s_2 related by $\mathcal{B}$, if their memories are modified respectively in accordance with $R(s_1)$ (which is equal to $R(s_2)$ by the definition of low equivalence and strong low bisimulation), then the resulting states are still related by $\mathcal{B}$. Formally, for any states s_1' and s_2' that are identical to s_1 and s_2 respectively except for their memories for which we require $mem(s_1)\ R\ mem(s_1')$ and $mem(s_2)\ R\ mem(s_2')$, it must be the case that $s_1'\ \mathcal{B}\ s_2'$.

Guarantee compliance, stating that each execution step of a component adheres to its current guarantee condition (condition (2) above) is straightforwardly defined by quantifying over all *locally reachable* states of each component. These are those states that can be reached by the component while allowing intermediate memory updates in between each of its execution steps in accordance with its current rely condition R. We omit the formal definition in the interest of brevity and instead refer the reader to our Isabelle theories; the intuition is however that locally reachable states are all possible states that the component might reach if run in parallel with other components that offer no more than the minimally required guarantees. In this sense it is an upper bound on all reachable states of the component when run in parallel with others that respect its assumptions.

Guarantee compliance then requires that in all such locally reachable states, if the component modifies the memory *mem* to produce memory *mem′*, then *mem G mem′*, where G denotes its current guarantee condition.

Definition 5 (Guarantee Compliance). A component satisfies *guarantee compliance* when for all locally reachable states s of the component, if the component performs a single execution step resulting in a state s', then $mem(s)\ G\ mem(s')$.

Condition (3), requiring that at all times all rely and guarantee conditions are compatible between components, we define as follows. We let σ denote states of the parallel composition of a set of components, under the usual interleaving semantics. States σ include the shared memory $mem(\sigma)$, as well as the local state of each component i, including its current rely and guarantee conditions, which we denote $R_i(\sigma)$ and $G_i(\sigma)$ respectively.

Definition 6 (Rely-Guarantee Compatibility). The rely and guarantee conditions of a collection of components running in parallel are said to be *compatible* when in all reachable states σ of the concurrent system, for all pairs of distinct components i and j, we have that $R_i(\sigma) \supseteq G_j(\sigma)$.

3.5. No-Read Assumptions and Guarantees

Standard rely-guarantee conditions R and G for verifying functional correctness of concurrent programs [17] capture assumptions and guarantees only about how memory will be *modified*. However, recall that COVERN also allows locks to protect variables from being read. Specifically, while a component holds a lock ℓ, COVERN allows it to assume that no other component will read from any variable $v \in NRW$ where $lockvars(\ell) = (NW, NRW)$ (see Section 2). Indeed, that such variables are assumed unreadable was encoded in the definition of low equivalence (Definition 1).

This kind of assumption is not part of standard rely guarantee reasoning for functional correctness, and was a novel invention of [23] in their initial rely guarantee framework for noninterference. We briefly explain its role, which we inherit from [23], [24], in our framework.

For a state s of a component, let $ANRW(s)$ ("assume no read-or-write") denote the set of variables for which the component currently has a no-read assumption. (In the context of COVERN, if the component has currently acquired all locks in the set L, this set of variables is $\bigcup\{\ NRW \mid \ell \in L \wedge lockvars(\ell) = (NW, NRW)\ \}$.) We also allow components to specify the variables they are currently guaranteeing not to read, denoted $GNRW(s)$. Then, overloading our existing notation to states σ of the parallel composition of a set of components, we augment the definition of rely-guarantee compatibility (Definition 6) to require that at all times for distinct components i and j that $GNRW_i(\sigma) \supseteq ANRW_j(\sigma)$.

To express that each component adheres to its own no-read guarantees, we strengthen the notion of guarantee compliance to require that each execution step from a locally reachable state s does not depend on any variables in $GNRW(s)$. Following [27], this is captured formally via an *erasure*-like condition that says that the operations of (a) modifying all variables in $GNRW(s)$ arbitrarily and (b) performing the execution step commute, implying that the

execution step cannot depend on the values of any such variables. As an aside, these kinds of erasure conditions have also been used to specify noninterference style security for functional programming languages, following [21], [32].

Incorporating no-read assumptions into standard rely-guarantee reasoning raises an interesting question. What does it mean if one component i makes a no-read assumption on a shared variable x, at the same time that some other component j's rely condition R_j constrains the value of x? For example, suppose R_j is such that for all memories $mem\ R_j\ mem'$, it is the case that $mem(x) = 3$. Component j thus has implicit knowledge about x's value through its rely condition, violating component i's no-read assumption.

This turns out not only to be an interesting thought experiment, but also to violate compositionality. To remedy this, we strengthen the definition of rely-guarantee compatibility further to require that at all times if one component has a no-read assumption on a shared variable, then no other component's rely condition can constrain x's value. Without this requirement, which we have motivated intuitively above, the rely-guarantee reasoning fails to compose. We leave it for future work to investigate whether there are other ways of achieving this kind of composition.

3.6. Compositionality

We can now formally state the compositionality theorem for our general rely-guarantee framework.

Theorem 1. Given a set of components running in parallel, and a strong low bisimulation $\mathcal{B}_i$ for each such component i, then the parallel composition of the components satisfies timing sensitive security (Definition 2) when: (1) each $\mathcal{B}_i$ is closed under the component's rely condition (Definition 4), (2) each component complies with its own guarantees (Definition 5) and, in the concurrent composition, all rely and guarantee conditions between components are compatible (Definition 6), where each of these definitions is strengthened as described above.

The machine checked proof of this theorem, whose presentation here is somewhat simplified for exposition, constitutes on its own about 2.5K lines of proof script, and follows the structure of prior compositionality proofs for more restrictive rely-guarantee notions of noninterference [23], [27].

3.7. Integration with COVERN

We conclude this section by describing how this rely-guarantee framework underpins and integrates with COVERN. In particular, when reasoning about each component, the rules of COVERN not only guarantee that a suitable strong low bisimulation $\mathcal{B}$ exists, but also that the side conditions of Theorem 1 are met.

Rely-Guarantee Compatibility. A component's rely and guarantee conditions naturally change throughout its execution. For instance, when acquiring a lock that protects a shared variable x, the component is allowed to assume that others won't modify x until such time as it releases the lock; likewise until it acquires the lock it should provide a corresponding guarantee to the others that it won't modify x. If rely and guarantee conditions can change dynamically, how can we ensure that at all times every component provides a sufficient guarantee condition to establish the rely condition of all others?

Just as locks provide the source of dynamically changing rely and guarantee conditions, they also provide the solution for making sure they stay compatible between components. Specifically, COVERN instantiates the rely and guarantee conditions as follows. At any time, each component guarantees not to modify any variable protected by any lock that it has not currently acquired, and likewise for variables protected from reading. The component also guarantees not to release any lock that it has not itself acquired. Rely conditions are instantiated so that each component i relies on other components not modifying any variables for which i has acquired a lock, and similarly for read-protected variables for which i has acquired an appropriate lock. Each component also relies on other components not to release any lock that it has itself acquired.

Then so long as each lock can be acquired by only one component at a time (a property guaranteed by the language semantics), then these rely and guarantee conditions can be shown to always remain compatible. All that remains for COVERN to check, to ensure compatibility, is that each component always adheres to its current guarantee relation, by never modifying (respectively modifying or reading) a variable it hasn't acquired a lock for and never releasing a lock it hasn't acquired.

Note that rely-guarantee compatibility in the presence of no-read assumptions also requires that all rely conditions are ignorant of any variables for which some other component has a no-read assumption. COVERN also ensures this by construction, which we explain shortly.

Data Invariants. COVERN's data invariants are incorporated straightforwardly via the general rely and guarantee conditions as well. We define a global invariant *global_inv* on the shared memory that asserts that for every lock ℓ that is not currently acquired by some component, its invariant $lockinv(\ell)$ must hold.

Definition 7 (Global Invariant). Memory *mem* satisfies the *global invariant*, denoted $global\_inv(mem)$, when for all locks ℓ not acquired by some component in memory *mem*, $lockinv(\ell)(mem)$ holds.

We then simply augment each component's rely and guarantee conditions to include the assumption and guarantee respectively that this global invariant is always preserved, so that they relate only those memories *mem* and mem' such that if $global\_inv(mem)$ holds then $global\_inv(mem')$ must also hold. Thus at the point at which a lock ℓ is acquired (meaning that the lock is not currently held), the global invariant implies that $lockinv(\ell)$ must be true. By forcing each component to ensure that the invariant remains true when the lock is released, COVERN guarantees that the global invariant is always preserved.

No-Read Assumptions. As mentioned above, COVERN instantiates the rely and guarantee conditions in a way that ensures that rely conditions are ignorant of no-read variables by construction. In particular, since the global invariant *global_inv* asserts only those data invariants for locks not currently held, it never constrains the values of any variable that some component might be holding a lock for, and so can never constrain any variables that some component might have a no-read assumption for.

Value-Dependent Classifications. So far we have avoided mentioning value-dependent classifications in this section, which we introduced in Section 2. As noted by [24], incorporating these into rely guarantee reasoning requires extending no-read assumptions on a variable x to also cover any variable y on which x's classification $\mathcal{L}(x)$ depends.

COVERN handles this by requiring that for all locks ℓ, if $lockvars(\ell) = (NW, NRW)$, letting X denote either of the sets NW or NRW, then a variable x is present in X if and only if a variable y mentioned in $\mathcal{L}(x)$ is present in X. Thus one cannot acquire access to a variable without also acquiring the same access to those variables that control its classification, and vice versa.

4. The COVERN Logic

Section 2. Here we take the time to carefully give a flavour of some of its rules; however we defer to our Isabelle theories for the full details.

COVERN is defined over a simple imperative language, whose current grammar is as follows. A *command cmd* in the language is one of

$$\begin{aligned} cmd ::= {} & \textbf{skip} \mid cmd\,;cmd \mid \textbf{if } e \textbf{ then } cmd \textbf{ else } cmd \mid \\ & \textbf{while } e \textbf{ do } cmd \mid x := e \mid \\ & \textbf{lock}(\ell) \mid \textbf{unlock}(\ell) \end{aligned}$$

where e is an expression on shared variables, $x \in Var$ is a shared variable, and $\ell \in Lock$ is a lock. Commands include no-ops (**skip**), sequencing (;), if-statements, while-loops, assignments to shared variables (x := e) and lock acquisition (**lock**(ℓ)) and release (**unlock**(ℓ)).

Since this language supports neither pointers nor arrays, identifying which parts of the shared memory are referenced by an expression e or a variable assignment x := e is trivial. Thus as mentioned in Section 1, while it borrows many ideas from concurrent separation logic (CSL), COVERN currently eschews CSL's maps-to ("$\mapsto$") predicate and its separating conjunction ("$*$"). However extending COVERN's language with pointers and arrays and incorporating these additional CSL ingredients should be relatively straightforward since the rely-guarantee framework already explicitly encodes heap footprints and (while not used in COVERN) already has the ability to reason about language commands for which the region of shared memory that they might read or write might itself depend on values in the shared memory.

Figure 3 presents those COVERN rules that we have found most useful so far. Some of these rules, like SKIP and SEQ for reasoning about no-ops and over sequences of commands are straightforward analogues of their traditional Hoare logic counterparts. Others, like ASSIGN_C are inspired by the dependent security type system of [27] and bear close resemblance to similar rules there.

Lock Acquisition and Release. We begin by explaining the LOCKACQ and LOCKREL rules for reasoning over lock acquisition and release, where COVERN's novelty shines. The LOCKACQ rule states that after acquiring a lock ℓ, one can assume the lock invariant $lockinv(\ell)$ holds, and that lock ℓ is now known to be acquired. The notation $\Gamma \oplus \ell$ denotes updating the context Γ which we recall tracks the sensitivity of the data contained in variables to which the component has acquired access. In particular, Γ is updated by adding entries for all variables $x \in lockvars(\ell)$ that are protected by the newly-acquired lock, such that for each variable x we add an entry $x \mapsto \mathcal{L}(x)$ that records that x's sensitivity matches its classification. This captures the idea that when acquiring access to a variable, one is allowed to assume its security contract, namely that the sensitivity of the data it contains is no greater than its (value-dependent) classification.

The corresponding rule LOCKREL for releasing a lock ℓ must first check that the lock is one that has already been acquired (i.e. that $\ell \in L$). It is safe to release the lock only if its lock invariant has been re-established (if it was violated in the meantime). Therefore, the rule checks that the facts P known about the current state are sufficient to prove that the lock invariant $lockinv(\ell)$ holds, denoted by the predicate entailment judgement $P \vdash lockinv(\ell)$. The notation $\Gamma \ominus \ell$ denotes removing from Γ entries for all variables in $lockvars(\ell)$ protected by the lock, since once the lock is released these assumptions about the sensitivity of the data that they might contain could be invalidated if another component acquires access the lock ℓ. Likewise, all predicates mentioning these variables have to be removed from P, which we denote by the overloaded notation $P \ominus \ell$. Finally the rule includes an additional side condition, abbreviated *side_condition*.

Before explaining the *side_condition*, recall that under value-dependent classification, certain variables effectively control the classifications of other variables. For instance in the example in Figure 1 of Section 2, the variable *input_clas* defines the classification of the variable *input*. Recall that these value-dependent classifications are captured by the static function $\mathcal{L}$, from variables to predicates on other variables. We call variables like *input_clas control variables*, since they control the classifications of other variables. Following [27], from which COVERN largely inherits its support for value-dependent classification, we denote the set of all such control variables C. C naturally contains precisely those variables mentioned in the $\mathcal{L}$ of another variable, i.e $C = \bigcup_{x \in Var}(vars(\mathcal{L}(x)))$, where for an expression e, $vars(e)$ denotes the variables mentioned in e.

The first job of *side_condition* is to check that when we release access to a variable x protected by the lock, that its sensitivity matches its security contract, i.e. that the sensitivity of the data that x contains ($\Gamma(x)$) does not exceed x's value-dependent classification $\mathcal{L}(x)$. This check has to be made relative to the information known still to

be true about the memory once the lock is released, i.e. under the assumption that all predicates in P' hold, since it needs to be true after the lock is released. Given two value-dependent classifications t and t', we denote that t does not exceed t' under the assumption that the predicates in some set Q hold, by $t \leq_Q t'$. Therefore the main check performed by *side_condition* is that for all $x \in lockvars(\ell)$, $\Gamma(x) \leq_{P'} \mathcal{L}(x)$.

The *side_condition* must perform one additional check that we explain as follows. Consider the example from Section 2 and what would happen if Γ contained information about the sensitivity of a variable in terms of *input_class*, but we released the lock that was protecting *input_clas*'s value. Doing so would allow some other component to potentially change *input_clas*'s value, thereby invalidating the information we have in Γ. Therefore, the *side_condition* also checks that the lock ℓ being released does not protect a control variable that is mentioned by an entry in Γ.

Variable Assignments. Figure 3 contains two of COVERN's rules for variable assignments. The rule ASSIGN_1 is for reasoning about assignments to variables to which the component has acquired access (by acquiring an appropriate lock) that are not control variables. Conversely, the rule ASSIGN_C is for reasoning about changes to control variables. Such updates must be handled very carefully, since they can necessarily alter the classification of other variables. For instance, changing a variable's classification from High to Low, without first clearing out any High data that it might contain, will violate the variable's (updated) security contract.

We focus on the ASSIGN_1 rule. This rule requires that the variable x being assigned to is one to which access has been acquired by acquiring a lock $\ell \in L$ for which $x \in lockvars(\ell)$, which we denote $modifiable_L(x)$. Likewise the rule has to check that no variable is being read to which access has not been legitimately acquired (which is required to ensure guarantee compliance, from Section 3). This check applies to all variables mentioned in the expression e, which we denote $readable_L(e)$. The expression sensitivity judgement $\Gamma \vdash e : t$ states that the sensitivity of the data in expression e is at most t (a value-dependent classification). Notice that the rule updates Γ to track x's new sensitivity as t, following the assignment. Doing so makes sense, however, only if all variables mentioned by t are *stable*, by which we mean assumed not-writable (because a lock $\ell \in L$ has been acquired for them). We denote this condition $stable_L(t)$. The rule computes a weakening P' of the strongest postconditon of the assignment $x := e$ with respect to precondition P by avoiding all references to variables not guaranteed to be stable by L. Finally, the rule has to check that if x is a variable to which we do not hold a no-read assumption, by having acquired an appropriate lock $\ell \in L$ (which we denote $x \notin NRW(L)$), that the data being written to it (expression e, whose sensitivity is given by t) does not violate x's security contract ($\mathcal{L}(x)$), i.e. that t does not exceed $\mathcal{L}(x)$.

The rule ASSIGN_C for updating control variables (those $x \in C$) is similar, but performs some extra checks. It must check that the control variable is being updated only with Low data, to prevent covert channels arising from changes to variable classifications [27], and also that any information in Γ won't be invalidated by the change, by checking that Γ mentions no classifications mentioning x. Finally, the *secure_update* side condition abbreviates a check that when we change a control variable x, all variables y that x controls for which we don't have a no-read assumption on y cannot be made insecure. This means that those y currently hold Low data, and after the control variable x is updated the data they hold (whose sensitivity might have changed) cannot exceed their type classification $\mathcal{L}(y)$ (which might also have changed).

Reasoning about Compound Statements. The rule WHILE for reasoning over while loops is the IFC analogue of the traditional while loop rule from Hoare logic. The rule IFLOW for reasoning over if-statements is similar, in that it reasons over each of the branches under the assumption that the condition e is true and false respectively. However, because COVERN performs forward reasoning, like existing separation logics [2], [3], it requires the user to supply a common postcondition at the join point of the if-statement, that is compatible with the postconditions derived on each of the two branches. This compatibility includes not only equivalence of value-dependent variable sensitivities tracked by Γ, and entailment between predicate sets P, but also preservation of internal well-formedness constraints between the three contexts Γ, L and P, which we denote using the $\leq$ symbol for brevity. In practice, one can derive more specialised forms of the IFLOW rule that can automatically derive suitable common contexts and discharge the compatibility checks; we refer the reader to our Isabelle formalisation.

4.1. Soundness

The soundness theorem for COVERN and its proof comprise about 6.5K lines of Isabelle/HOL, at most a few hundred of which define the logic itself. The theorem states that if one uses the rules of the logic to prove a set of components secure, then the parallel composition of those components satisfies timing sensitive noninterference. The proof proceeds by establishing a bisimulation $\mathcal{B}$ for each component and showing that all such meet the side conditions of Theorem 1.

Proving a component secure using COVERN starts with the empty contexts Γ_0, L_0, and P_0, each of which contain no information, so $dom(\Gamma_0) = \{\}$, $L_0 = \{\}$ and $P_0 = \text{true}$. Then the soundness theorem is:

Theorem 2 (Soundness of* COVERN*). Given components c_i and contexts Γ_i, L_i, and P_i, for $i = 1, \ldots, k$, such that $\Gamma_0, L_0, P_0 \ \{c_i\} \ \Gamma_i, L_i, P_i$ the components running in parallel satisfy timing sensitive noninterference.

4.2. Discussion

Observe that each of the rules of Figure 3 essentially performs two kinds of checks: those that preserve security (often categorised as comparisons between variable/expression

$$\frac{\mathit{modifiable}_L(x) \quad \mathit{readable}_L(e) \quad \Gamma \vdash e : t \quad \mathit{stable}_L(t) \quad P' = \mathit{post}_L(x := e, P) \quad x \notin \mathit{NRW}(L) \longrightarrow t \leq_{P'} \mathcal{L}(x)}{\Gamma, L, P\ \{x := e\}\ \Gamma[x \mapsto t], L, P'}\ \textsc{Assign}_1$$

$$\frac{x \in C \quad \mathit{modifiable}_L(x) \quad \mathit{readable}_L(e) \quad \Gamma \vdash e : t \quad t \leq_P \mathsf{Low} \quad (\forall v \in \mathit{dom}(\Gamma).\ x \notin \mathit{vars}(\Gamma(v))) \quad P' = \mathit{post}_L(x := e, P) \quad \mathit{secure\_update}}{\Gamma, L, P\ \{x := e\}\ \Gamma', L, P'}\ \textsc{Assign}_C$$

$$\frac{\mathit{readable}_L(e) \quad \Gamma \vdash e : t \quad t \leq_P \mathsf{Low} \quad \Gamma, L, (P \wedge e)\ \{c_1\}\ \Gamma_1, L', P_1 \quad \Gamma, L, (P \wedge \neg e)\ \{c_2\}\ \Gamma_2, L', P_2 \quad \Gamma_1 =_{P_1} \Gamma' \quad \Gamma_2 =_{P_2} \Gamma' \quad P_1 \vdash P' \quad P_2 \vdash P' \quad (\Gamma_1, L', P_1) \leq (\Gamma', L', P') \quad (\Gamma_2, L', P_2) \leq (\Gamma', L', P')}{\Gamma, L, P\ \{\textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2\}\ \Gamma', L', P'}\ \textsc{IfLow}$$

$$\frac{\mathit{readable}_L(e) \quad \Gamma \vdash e : t \quad t \leq_P \mathsf{Low} \quad \Gamma, L, (P \wedge e)\ \{c\}\ \Gamma, L, P}{\Gamma, L, P\ \{\textbf{while}\ e\ \textbf{do}\ c\}\ \Gamma, L, P}\ \textsc{While}$$

$$\frac{\Gamma, L, P\ \{c_1\}\ \Gamma', L', P' \quad \Gamma', L', P'\ \{c_2\}\ \Gamma'', L'', P''}{\Gamma, L, P\ \{c_1\ ;\ c_2\}\ \Gamma'', L'', P''}\ \textsc{Seq} \qquad \frac{}{\Gamma, L, P\ \{\textbf{skip}\}\ \Gamma, L, P}\ \textsc{Skip}$$

$$\frac{\Gamma' = \Gamma \oplus \ell}{\Gamma, L, P\ \{\textbf{lock}(\ell)\}\ \Gamma', L \cup \{\ell\}, (P \wedge \mathit{lockinv}(\ell))}\ \textsc{LockAcq}$$

$$\frac{\ell \in L \quad P \vdash \mathit{lockinv}(\ell) \quad \Gamma' = \Gamma \ominus \ell \quad P' = P \ominus \ell \quad \mathit{side\_condition}}{\Gamma, L, P\ \{\textbf{unlock}(\ell)\}\ \Gamma', L - \{\ell\}, P'}\ \textsc{LockRel}$$

Figure 3: Commonly used rules of Covern.

sensitivities and classifications), and those that ensure that the component is complying with its own guarantees, as captured via the set of locks L currently acquired.

Unlike prior security type systems built on top of rely-guarantee frameworks for noninterference, Covern is the first to not only establish suitable bisimulations for each verified component, but also to simultaneously establish the other side conditions of the associated compositionality theorem (which we note in our case are even more involved than their predecessors because we support general rely guarantee reasoning for the first time).

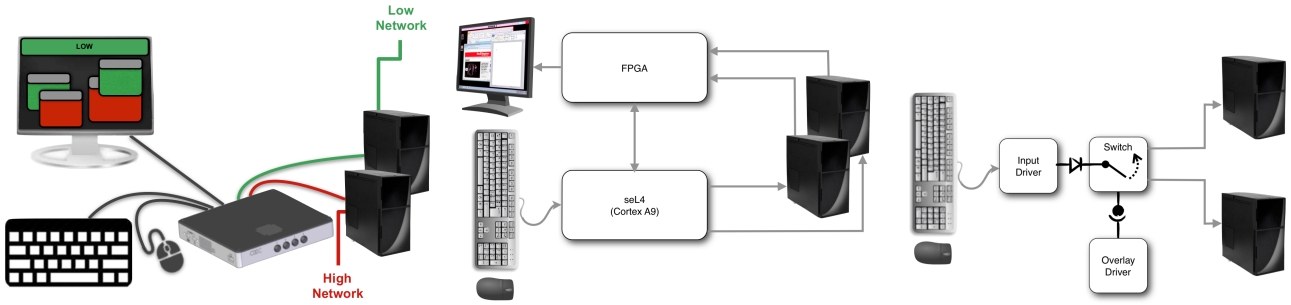
While the rules in Figure 3 are currently restricted to reasoning only about programs that do not branch on High data, we note that the set of such programs has increased dramatically in recent years as the danger of secret dependent memory accesses has become apparent through ever more sophisticated cache side channel attacks [15] on cryptographic implementations, via the rise of "constant time" crypto libraries like NaCl [7], [8], [12]. Indeed, the Cross Domain Desktop Compositor (CDDC) software whose modelling and verification we discuss in Section 5 also avoids secret dependent memory accesses. For these reasons Covern remains useful even without additional rules to reason over if-conditions that branch on High data.

Nonetheless, adding such rules should be straightforward. In particular, one can imagine a rule that when encountering a potentially High conditional **if** e **then** c_1 **else** c_2 first checks that the lengths of the two branches c_1 and c_2 are identical, ensuring (at least to the level of the source language semantics) that the two branches take the same number of steps to execute. The rule would then fire additional ones to check over each branch separately to make sure that any variables being assigned to that are not protected from being read by an appropriate lock are High classified. We leave adding these kinds of additional rules for future work.

5. Modelling and Verifying the CDDC

The Cross Domain Desktop Compositor [6] (CDDC) is a security-critical device that allows a single trusted user to interact with multiple desktop machines, each of which might be connected to a network of differing security classification, via a single keyboard, video display (monitor) and mouse. It is designed to provide the separation guarantees of physical isolation between the networks while at the same time affording a seamless user experience.

The CDDC receives video input from each of the desktop machines (see Figure 4a) and composites the various display elements onto a single screen to provide an integrated user experience. Window geometry information is communicated in-band in the video signal from each of the desktop machines, which is necessary because the *only* output channel from the desktop machines to the CDDC is via the video display output. Likewise the CDDC receives input from the



(a) CDDC hardware use-case setup. (b) CDDC hardware architecture. (c) CDDC seL4 component architecture.

Figure 4: Cross-domain desktop compositor functional schematics.

user's keyboard and mouse, which it forwards to whichever desktop machine is currently *active*, namely the one whose application the user is currently interacting with. We refer to each of the desktop machines as a separate *domain*, and sometimes call the active domain the *current domain*.

When the user clicks a different part of the display with her mouse pointer, the CDDC determines which domain has the topmost content on screen at that location. That domain then becomes the new active domain, causing keyboard and mouse input to be directed to it. In this way, domain switching is facilitated via ordinary mouse gestures enhancing the quality of the user experience.

The CDDC's security naturally rests on the user remaining at all times aware of which domain is currently active, and to which domain each of the on-screen elements belongs. Therefore, the CDDC decorates each of the composited display elements with a separate coloured border to unambiguously identify which domain it belongs to. The CDDC additionally indicates at all times which is the active domain by rendering a permanent and prominent UI element, called the *overlay*, at the top of the screen. The overlay is a large, vividly coloured banner that names the currently active domain and is coloured according to a unique colour assigned to that domain (also used to decorate the borders of all of the domain's other on-screen content).

The CDDC was originally prototyped using just an FPGA [6], but has since been re-implemented using a FPGA-software combination, in which keyboard and mouse input processing is implemented in software on top of seL4 running on an ARM Cortex A9 processor (see Figure 4b). The software implementation provides greater flexibility with regard to runtime configuration and user preferences.

In the FPGA-software implementation, the FPGA remains responsible for rendering the composited display, including the overlay indicating the active domain and the decorations on each of the display elements. Thus the FPGA holds the authoritative information about which domain occupies each region of the composited display, and which domain is currently active. However, the software is responsible for directing keyboard and mouse input at all times to the active domain. When a mouse click is received, the software queries the FPGA to find out which domain was clicked on, but then it is the *software* that is responsible for directing the FPGA that a domain switch has taken place, and which domain was switched to (allowing for future customisation of domain-switching gestures). Therefore the software maintains its own internal state about which domain is currently active. A critical security invariant, then, is that the software's idea of which is the active domain agrees with the FPGA's.

Indeed, a basic assumption is that at all times, the user inputs data whose sensitivity agrees with the active domain, as rendered on the on-screen overlay. If the software's idea of which domain is currently active disagrees, the input may go to the wrong domain: a clear security violation.

5.1. Verified seL4-Based Software Architecture

The user input handling software of the CDDC is implemented as a concurrent program consisting of multiple application components running on top of the verified seL4 microkernel [19]. The software component architecture we verified is depicted in Figure 4c.

In the remainder of this section we discuss how we applied COVERN to model and verify the behaviour of this concurrent software system for input handling.

The software comprises three components, which we abbreviate SWITCH, INPUT, and OVERLAY (see Figure 4c).

INPUT is a driver component that sits in an infinite loop querying the mouse and keyboard hardware interfaces for input, and forwarding it to SWITCH via a shared-memory buffer. In COVERN, for simplicity we model the shared buffer as a single-place buffer; doing so does not affect the essence of the security argument, as we explain later.

Importantly, to verify this concurrent software, we have to capture the assumption that the user always provides input that agrees with the current domain indicated on the on-screen overlay. Therefore INPUT includes a model of the decision of a trusted user who faithfully chooses (via a single if-statement) whether to provide High or Low input, making this choice in accordance with the value of a single shared variable, `indicated_domain`, which in our model represents the state of the on-screen overlay (i.e. represents the FPGA's idea of the currently active domain).

OVERLAY is a driver component for interacting with the FPGA that sits in an infinite loop and provides an RPC interface so that SWITCH can query the FPGA about which

domain occupies a particular location on-screen (e.g. when a mouse click is received, as described above). We model this RPC interface in COVERN using shared variables and locks, whose details are relatively straightforward and are omitted here for brevity.

SWITCH sits in an infinite loop that on each iteration reads an input event from the buffer that it shares with INPUT, and then directs that input to whichever domain SWITCH believes is currently active. Mouse clicks are handled by SWITCH by first querying the OVERLAY to find out which domain was clicked on, and then updating the FPGA state if SWITCH determines that the user has performed a domain-switch. We discuss below our decisions regarding value-dependent classification of the different input event types, in light of which input events are allowed to cause a domain switch.

To implement these concerns, each iteration of SWITCH's server loop polls the shared buffer to take a local copy of the current input event, and then branches on whether the input event is a mouse event or a keyboard event. Keyboard input events cannot change which is the active domain, and so are handled relatively simply: SWITCH merely sends the keyboard event to the currently active domain according to the software's idea of which domain is active, which is stored in a variable `active_domain`. On the other hand, mouse inputs can change which is the active domain, hence this case is more complicated, and includes not only querying OVERLAY as mentioned above, but also discerning whether a mouse click has occurred by comparing the current mouse event against past ones, which SWITCH tracks in a variable named `switch_state_mouse_down`.

If the current mouse event is deemed to cause a domain switch (because it clicks on a domain other than `active_domain`), SWITCH will then update both the software's `active_domain` variable alongside issuing a command to the compositor hardware to update its idea of which is the active domain. In our model we capture this final update with a write to the shared `indicated_domain` variable. Finally, SWITCH sends the mouse event itself to whichever domain it is currently hovering over, and updates the aforementioned `switch_state_mouse_down` variable.

5.2. Value-Dependent Classification of Input

Recall that a domain switch may occur only via mouse clicks. Mouse input must therefore be classified Low, for otherwise the user's decisions about when to switch domains would itself constitute a covert channel.

Keyboard input on the other hand is drawn from two different sources, one for High input, which is classified High, and the other for Low input, naturally classified Low. Once it enters the SWITCH component, however, its classification is value-dependent. Since SWITCH chooses the input source based on the FPGA's `indicated_domain`, the variables that hold keyboard input received by SWITCH are classified according to the current value of `indicated_domain`.

Thus, the classification of any input event stored in the single shared buffer between INPUT and SWITCH is dependent on two *control variables*: the field of the input event indicating whether it is a keyboard or a mouse event, which we model with a variable `input_event_type`, and the current `indicated_domain`. As required by COVERN, control variables must be statically classified Low. We thus split off the data field of the input event into its own variable `input_event_data`. Now it can be dynamically classified High or Low without affecting the classification of its associated control variable. Its value-dependent classification is as follows:

if `input_event_type = KEYBOARD` $\wedge$ `indicated_domain = DOM_HIGH` then High else Low

Observe that even in the relatively simple language of COVERN, this model makes good use of fine-grained value-dependent classifications for specifying security contracts, in this case for the interface between INPUT and SWITCH.

5.3. Critical Sections and Data Invariants

COVERN's support for data invariants is vital for encoding the contract that the FPGA's idea of which domain is currently active (`indicated_domain`), agrees with the software's (`active_domain`). Therefore, we associate these variables with a common lock used to protect access to them, and associate with this lock the data invariant that

$$\texttt{indicated\_domain} = \texttt{active\_domain}.$$

This invariant *must* be encoded as one attached to a lock in this way because e.g. `indicated_domain` is accessed by both INPUT (to choose the appropriate input source, to model a trustworthy user), and SWITCH (which updates it when it determines that a domain switch has occurred). Therefore each of these components must rely on the other to preserve the data invariant while that component is accessing the shared variables. Specifying it as a data invariant captures this requirement precisely. Naturally COVERN then obliges us to prove that each of these components does indeed maintain the invariant, as required for security.

Since this lock protects a control variable for `input_event_data`, it must naturally protect that variable as well, along with its other control variables.

Thus INPUT acquires this lock while it consumes the user's input from whichever input source is indicated by `indicated_domain`. Doing so naturally captures the requirement that the `indicated_domain` cannot change while the user is inputting data (but only between each inputted value, ensuring our model of the trustworthy user is not subject to Time-Of-Check-Time-Of-Use (TOCTOU) vulnerabilities.)

The SWITCH must also acquire this lock when reading the input from the shared buffer. Since the input is spread across multiple variables (as it is in the real implementation), it cannot be read atomically without this lock. Besides ensuring atomic access to the compound input data, this lock also prevents INPUT trying to place new input in the buffer while a domain switch might be occurring.

COVERN naturally allows SWITCH to assume the data invariant `active_domain = indicated_domain` when it

acquires the lock, and obliges it to show that the invariant is maintained when the lock is released.

Thus updates to these variables must be performed while holding the lock, and the data invariant prevents SWITCH from releasing the lock having updated e.g. its own internal idea of which domain is active without also having updated the FPGA state. Changing `indicated_domain` (the FPGA state) alters the classification of the input buffer, which depends on `indicated_domain` as described above. Thus when changing the active domain, SWITCH is required to flush the shared buffer. Doing so prevents vulnerabilities in which stale input data, input by the user while a previous domain was still active (when SWITCH hadn't yet processed a pending domain switch gesture), is subsequently sent to the *new* active domain.

5.4. Discussion

The CDDC model comprises about 200 lines of "code" in COVERN's language. While we have described only a part of it, we see that both data invariants and value-dependent classifications were critical to specify and verify the dynamic security requirements of this concurrent program. The entire model and its proofs are just over 1,000 lines of Isabelle.

As mentioned above, we model here only a single-place shared buffer for input events between INPUT and SWITCH. In the implementation, this buffer is a shared-memory array with head and tail indices that implement a producer-consumer ring buffer. While these details do not change the essence of the security argument, proving the ring buffer implementation secure would benefit from augmenting COVERN with support for arrays. We leave doing so for future work.

We note that our model captures an early prototype of the CDDC software, which has since been extended to support features like initiating domain switches via keyboard hotkey sequences and via hardware buttons on the front of the device. Modelling the latter functionality would be relatively straightforward (as such inputs would be classified as statically Low); however COVERN is not equipped to model the former (in which the classification of a keystroke might depend on the ones that *follow* it).

We also note that our model applies to an instance of the CDDC connected to just two networks of High and Low relative classifications. Extending COVERN to arbitrary lattices of classifications should be relatively straightforward.

6. Related Work

COVERN and its underlying general rely guarantee theory is heavily inspired by previous work on rely guarantee reasoning for noninterference [23], [27]. It can be seen as extending such prior work by adding support for general rely guarantee relations R and G. Prior frameworks supported only coarse grained assumptions on individual shared variables, such as "no write" or "no read-write". Such assumptions are insufficient to justify the reasoning necessary to preserve data invariants in between when a component releases a lock ℓ and later re-acquires it. Thus COVERN couldn't exist without the general rely guarantee framework of Section 3.

COVERN also borrows from concurrent separation logic [10], [30]. In particular, by associating data invariants and assumptions and guarantees on locks, it automatically enforces rely-guarantee compatibility. No prior rely-guarantee-based noninterference reasoning system achieved this feat.

COVERN implicitly requires all locks to be classified as Low, preventing locks from being acquired when the program's execution timing is affected by High data (e.g. within a High loop or a High conditional). This kind of restriction is natural in the context of timing sensitive noninterference to prevent timing leaks between threads [33].

One might reasonably argue that source level reasoning about timing sensitive security is necessarily fraught, since timing variations due to cache interference will cause far more serious deviations in execution time than e.g. differing numbers of execution steps in a High conditional. We agree and note that these issues do not arise in the context of code that doesn't branch on secrets or perform secret dependent memory look-ups, a category that we mentioned already is becoming ever larger, and to which COVERN is currently geared. On the other hand, [12] recently present a type system for verifying (non-value-dependent) timing-sensitive noninterference for a micro-architecture in which caches are absent, namely AVR. We note that transferring COVERN's reasoning down to assembly code, where reasoning about timing becomes more precise, can be achieved by crafting special purpose refinement theories, even in the presence of concurrency [27]. Finally, the assumption of execution steps taking constant time can also be approximated, at least internally between components, by adopting *instruction based scheduling* [35] (IBS).

Beaumont et al. [6] present a formal IFC proof for an initial prototype hardware design of the CDDC. Unlike ours presented here in Section 5, theirs deals with a far simpler logical model of the hardware's functionality, whereas ours deals in its more recent seL4-based concurrent software implementation. Their proof is not compositional, resting on the manual definition and proofs of maintenance of a single relational invariant across all of the state in the model. Ours exploits COVERN's natural support for modular reasoning.

7. Conclusion

We presented COVERN, the first general logic for compositional reasoning about IFC for concurrent shared memory programs, underpinned by a general framework for rely guarantee reasoning for IFC, demonstrating its utility by describing its successful application to verifying the software design of the Cross Domain Desktop Compositor.

Acknowledgements

We thank the anonymous reviewers for their valuable feedback and for convincing us not to name the logic COVFEFE. This research was supported by the Commonwealth

of Australia Defence Science and Technology Group and the Defence Science Institute, an initiative of the State Government of Victoria.

References

[1] T. Amtoft, S. Bandhakavi, and A. Banerjee, "A logic for information flow in object-oriented programs," in *POPL*, 2006, pp. 91–102.

[2] A. W. Appel, R. Dockins, A. Hobor, L. Beringer, J. Dodds, G. Stewart, S. Blazy, and X. Leroy, *Program Logics for Certified Compilers*, New York, NY, USA, 2014.

[3] A. W. Appel and others, "The Verified Software Toolchain," https://github.com/PrincetonUniversity/VST, 2017.

[4] T. Bauereiß, A. P. Gritti, A. Popescu, and F. Raimondi, "CoSMed: A confidentiality-verified social media platform," in *ITP*, 2016, pp. 87–106.

[5] ——, "CoSMeDis: a distributed social media platform with formally verified confidentiality guarantees," in *S&P*. IEEE, 2017, pp. 729–748.

[6] M. Beaumont, J. McCarthy, and T. Murray, "The cross domain desktop compositor: Using hardware-based video compositing for a multi-level secure user interface," in *ACSAC*, 2016, pp. 533–545.

[7] D. J. Bernstein, T. Lange, and P. Schwabe, "Nacl: Networking and cryptography library," Available at: http://nacl.cr.yp.to/.

[8] ——, "The security impact of a new cryptographic library," in *2nd LATINCRYPT*, Santiago, CL, Oct 2012, pp. 159–176.

[9] G. Boudol and I. Castellani, "Noninterference for concurrent programs and thread systems," *Theoretical Computer Science*, vol. 281, no. 1-2, pp. 109–130, 2002.

[10] S. Brookes, "A semantics for concurrent separation logic," *Theoretical Computer Science*, vol. 375, pp. 227–270, 2007.

[11] D. Costanzo, Z. Shao, and R. Gu, "End-to-end verification of information-flow security for C and assembly programs," in *PLDI*, 2016, pp. 648–664.

[12] F. Dewald, H. Mantel, and A. Weber, "AVR processors as a platform for language-based security," in *ESORICS*, ser. LNCS, vol. 10492, 2017, pp. 427–445.

[13] T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang, "Views: compositional reasoning for concurrent programs," *SIGPLAN Notices*, vol. 48, no. 1, pp. 287–300, 2013.

[14] X. Feng, R. Ferreira, and Z. Shao, "On the relationship between concurrent separation logic and assume-guarantee reasoning," in *ESOP*, ser. LNCS, vol. 4421, 2007, pp. 173–188.

[15] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *Journal of Cryptographic Engineering*, 2018, (to appear).

[16] J. Goguen and J. Meseguer, "Security policies and security models," in *S&P*, Oakland, California, USA, Apr 1982, pp. 11–20.

[17] C. B. Jones, "Development methods for computer programs including a notion of interference," D.Phil. thesis, University of Oxford, Jun 1981.

[18] S. Kanav, P. Lammich, and A. Popescu, "A conference management system with verified document confidentiality," in *CAV*, ser. LNCS, vol. 8559, 2014, pp. 167–183.

[19] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive formal verification of an OS microkernel," *ACM Transactions on Computer Systems*, vol. 32, no. 1, pp. 2:1–2:70, Feb 2014.

[20] R. Krebbers, A. B. Ralf Jung, J.-H. Jourdan, D. Dreyer, and L. Birkedal, "The essence of higher-order concurrent separation logic ("Iris 3.0")," in *ESOP*, ser. LNCS, vol. 10201, 2017, pp. 696–723.

[21] P. Li and S. Zdancewic, "Arrows for secure information flow," *Theoretical Computer Science*, vol. 411, no. 19, pp. 1974–1994, 2010.

[22] L. Lourenço and L. Caires, "Dependent information flow types," in *POPL*, Mumbai, India, Jan 2015, pp. 317–328.

[23] H. Mantel, D. Sands, and H. Sudbrock, "Assumptions and guarantees for compositional noninterference," in *CSF*, Cernay-la-Ville, France, Jun 2011, pp. 218–232.

[24] T. Murray, "On high-assurance information-flow-secure programming languages," in *PLAS*, Prague, Czech Republic, Jul 2015, pp. 43–48.

[25] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein, "seL4: from general purpose to a proof of information flow enforcement," in *S&P*, San Francisco, CA, May 2013, pp. 415–429.

[26] T. Murray, D. Matichuk, M. Brassil, P. Gammie, and G. Klein, "Noninterference for operating system kernels," in *CPP*, Kyoto, Japan, Dec 2012, pp. 126–142.

[27] T. Murray, R. Sison, E. Pierzchalski, and C. Rizkallah, "Compositional verification and refinement of concurrent value-dependent noninterference," in *CSF*, Lisbon, Portugal, Jun 2016, pp. 417–431.

[28] A. Nanevski, A. Banerjee, and D. Garg, "Verification of information flow and access control policies with dependent types," in *S&P*, May 2011, pp. 165–179.

[29] T. Nipkow, L. Paulson, and M. Wenzel, *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, ser. LNCS, 2002, vol. 2283.

[30] P. W. O'Hearn, "Concurrency, and local reasoning," ser. LNCS, vol. 3170, 2004.

[31] S. Owicki and D. Gries, "An axiomatic proof technique for parallel programs," *Acta Informatica*, vol. 6, pp. 319–340, 1976.

[32] A. Russo, K. Claessen, and J. Hughes, "A library for light-weight information-flow security in Haskell," in *1st ACM SIGPLAN Haskell Symposium*, 2008, pp. 13–24.

[33] A. Sabelfeld, "The impact of synchronisation on secure information flow in concurrent programs," in *Ershov Memorial Conference 2001*, 2001, pp. 225–239.

[34] A. Sabelfeld and D. Sands, "Probabilistic noninterference for multi-threaded programs," in *IEEE Computer Security Foundations Workshop (CSFW)*, 2000, pp. 200–215.

[35] D. Stefan, P. Buiras, E. Z. Yang, A. Levy, D. Terei, A. Russo, and D. Mazières, "Eliminating cache-based timing attacks with instruction-based scheduling," in *ESORICS*, Egham, UK, Sep 2013, pp. 718–735.

[36] N. Swamy, J. Chen, and R. Chugh, "Enforcing stateful authorization and information flow policies in Fine," in *ESOP*, March 2010.

[37] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang, "Secure distributed programming with value-dependent types," in *ICFP*, 2011, pp. 266–278.

[38] D. Volpano and G. Smith, "Probabilistic noninterference in a concurrent language," *Journal of Computer Security*, vol. 7, no. 2,3, pp. 231–253, 1999.

[39] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," in *ASPLOS*, 2015.

[40] L. Zheng and A. C. Myers, "Dynamic security labels and static information flow control," *International Journal of Information Security*, vol. 6, no. 2–3, Mar 2007.