Verified Secure Compilation for Mixed-Sensitivity Concurrent Programs

Robert Sison[*] [†] and Toby Murray[*]

[*]School of Computing and Information Systems, University of Melbourne, Australia
[†]CSIRO's Data61 and UNSW Sydney, Australia
(e-mail: [firstname].[lastname]@unimelb.edu.au)

Abstract

Proving only over source code that programs do not leak sensitive data leaves a gap between reasoning and reality that can only be filled by accounting for the behaviour of the compiler. Furthermore, software does not always have the luxury of limiting itself to single-threaded computation with resources statically dedicated to each user to ensure the confidentiality of their data. This results in *mixed-sensitivity concurrent programs*, which might reuse memory shared between their threads to hold data of different sensitivity levels at different times; for such programs, a compiler must preserve the *value-dependent* coordination of such *mixed-sensitivity reuse* despite the impact of *concurrency*.

Here we demonstrate, using Isabelle/HOL, that it is feasible to verify that a compiler preserves *noninterference*, the strictest kind of confidentiality property, for mixed-sensitivity concurrent programs. First, we present notions of refinement that preserve a *concurrent value-dependent* notion of noninterference that we have designed to support such programs. As proving noninterference-preserving refinement can be considerably more complex than the standard refinements typically used to verify semantics-preserving compilation, our notions include a decomposition principle that separates the semantics-preservation from security-preservation concerns. Second, we demonstrate that these refinement notions are applicable to verified secure compilation, by exercising them on a single-pass compiler for mixed-sensitivity concurrent programs that synchronise using mutex locks, from a generic imperative language to a generic RISC-style assembly language. Finally, we execute our compiler on a nontrivial mixed-sensitivity concurrent program modelling a real-world use case, thus preserving its source-level noninterference properties down to an assembly-level model automatically. All results are formalised and proved in the Isabelle/HOL interactive proof assistant.

Our work paves the way for more fully featured compilers to offer verified secure compilation support to developers of multithreaded software that must handle data of multiple sensitivity levels.

1 Introduction

Here we show how to extend secure compilation support to programs that are designed to address two fundamental problems of scale: (1) the need to divide work in computer systems that handle information, and (2) the need to share scarce resources to be able to service every customer for whom that work is done. There will always be a program for which that sharing is not abstracted; that program's responsibility is to implement that sharing in such a way that it never allows the information of one customer to flow to another. In this paper, we prove formally that a compiler does not break that program's responsibility.

It is well known that program translations of the kind carried out by compilers can easily break security properties like confidentiality (Kaufmann *et al.*, 2016; Barthe *et al.*, 2018). This is especially the case for *mixed-sensitivity concurrent programs*, which feature both:

- *Concurrency* of access to memory locations shared between different threads of execution. A compiler must preserve both (1) the synchronisation that coordinates threads' access to shared memory, and (2) the absence of any internal timing leaks, to prevent them from manifesting as storage leaks (Volpano & Smith, 1998).
- *Mixed-sensitivity reuse* of shared memory to hold information of different sensitivity levels at different times. A compiler must preserve the program functionality that coordinates this reuse; this implies support for *value-dependent classification* policies, which allow the classification of a memory location to change dynamically depending on values held in other memory locations (Murray, 2015). Furthermore, it must do so accounting for the potential impact of concurrent access by other threads.

Although existing verified compilers for dialects of mainstream programming languages, like CompCert (Leroy, 2009) and CakeML (Kumar *et al.*, 2014), have been proved to preserve program functionality (semantics) and some timing-sensitive forms of noninterference (Barthe *et al.*, 2020), none are yet verified to preserve proofs of noninterference for mixed-sensitivity concurrent programs. Ideally such a compiler, applied to the threads of a proved-secure mixed-sensitivity concurrent program, would yield assembly code that, run concurrently, also composes into a secure mixed-sensitivity concurrent program.

To this end, here we present notions of *concurrent value-dependent noninterference*-preserving refinement, which are compositional across the threads of mixed-sensitivity concurrent programs. In these notions, the usual square-shaped commuting diagram commonly used to depict (semantics-preserving) refinement (Figure 4a) has been replaced by a *cube* (Figure 3). This reflects that it preserves a *2-safety hyperproperty* (Terauchi & Aiken, 2005; Clarkson & Schneider, 2010), which compares two executions rather than examining a single one. Our earlier work (Murray *et al.*, 2016b) was the first to make this observation and to propose a general cube-shaped refinement property; however other work on verified secure compilation targeted towards noninterference preservation (Barthe *et al.*, 2018, 2020) since made the same observation. As these cube-shaped properties are significantly more complicated to prove than standard notions of semantics-preserving refinement typical in verified compilation (Leroy, 2009; Kumar *et al.*, 2014), we present a principle of decomposing the cube (Figure 3) into three separate obligations (Figure 4): the first of these is akin to semantics-preserving refinement, while the rest prevent the introduction of any termination- and timing-leaks. A simple comparison of proof effort for a refinement example (Figure 2) shows this approach can almost halve its complexity, and that it is applicable to proofs of refinement for programs with *secret-dependent control flow*—the example pads an **if** h **then** … **else** … **fi** conditional with **skip**s, so as not to introduce a timing leak of h.

We then go on to demonstrate that the decomposition principle we provide makes our notion of refinement a tractable target for verified secure compilation. Our compiler is an executable function in Isabelle/HOL that translates mixed-sensitivity concurrent programs that synchronise using mutex locks, from a generic imperative While language to a generic RISC-style assembly language. In particular, it supports the class of programs that avoid all

implicit flows, where a secret determines the choice between two control flow paths with different observable effects, by disallowing any secret-dependent control flow—for example, disallowing **if** h **then** ... **else** ... **fi** conditionals to prevent any timing leaks from h. This is a common approach against implicit flows, as it avoids any precise source-level reasoning about time. To preserve confidentiality for programs that take that approach, we instantiate the decomposition principle so that it enforces that our compiler does not introduce any new secret-dependent control flow. Furthermore, as part of satisfying the demands of our refinement notion, our compiler demonstrates a way of formalising and proving when it is safe for a compiler to perform optimisations in the presence of concurrency. To ensure that the contents of shared memory locations are preserved under compilation despite potential interference from other threads, our compiler tracks which shared memory locations are free from data races. It then makes use of this tracking to avoid redundant loads from "stable" (i.e. race-free) shared variables safely, that would otherwise be considered unsafe to omit.

Finally, to show that the compiler preserves noninterference for actual mixed-sensitivity concurrent programs, we execute it on a real-world use case: a model of the software-componentised input-handling regime for the *Cross Domain Desktop Compositor* (Beaumont *et al.*, 2016), a device that enforces information-flow control over input classified dynamically by a trusted user. We leave treatment on the design and application of per-thread proof techniques establishing CVDNI for the successive versions of this model to other works (Murray *et al.*, 2018; Sison, 2020), and here focus on its CVDNI-preserving compilation—expanding on Sison & Murray (2019), the conference version of this paper. This yields the first proofs of noninterference for an assembly-level model of a nontrivial mixed-sensitivity concurrent program, demonstrating the power of verified secure compilation to preserve security properties of compiled code.

The structure of our paper is as follows. First, we present language-independent notions of noninterference and its refinement, designed for mixed-sensitivity concurrent programs (Section 2). Our attention then turns to preliminaries for our compiler: the main properties of interest of the source `While` language it compiles (Section 3), and of the target `RISC` language it produces (Section 4). Then, after presenting the details of our compiler and its verification (Section 5), and the case study to which we apply it (Section 6), we discuss the most closely related work in the area (Section 7), before concluding (Section 8).

We expand on the conference version of this paper (Sison & Murray, 2019) as follows:

- Here we have adapted the noninterference properties to support assumptions on initial memory and extra security requirements; these will allow us to clarify exactly what our compiler is verified to preserve, and for which kinds of programs.

- We also in Section 2 present further preliminaries that will allow us to explain in greater detail the different ways to establish and use proofs about a verified compiler to obtain whole-system noninterference at the target-language level. These include:

    1. The side conditions and theorem of compositionality for the noninterference properties. In Section 3, we then for the first time present the proof, whose details were elided from Sison & Murray (2019), for a noncompositional "global" part of this side condition, which is necessary to obtain whole-system noninterference from per-thread noninterference both at source and target level.

2. A whole-system refinement theorem, adapted to support assumptions on initial memory. This theorem was alluded to in Murray *et al.* (2016b) but, until now, has never been formally presented outside of the Isabelle/HOL theories. It gives us a means to prove preservation of whole-system noninterference by the compiler, without having to re-prove the noncompositional side condition at the target-language level.

- In Section 5, we then compare alternative methods of obtaining whole-system security at RISC level, that a developer would choose depending on whether all, or only some threads are compiled with our compiler. In contrast, Sison & Murray (2019) stopped after presenting the application of refinement decomposition principle.

- In Section 6, the case study to which we apply the compiler is significantly expanded, being a 3-component version of the CDDC input-handling program—closer to a version presented in Murray *et al.* (2018)—as opposed to the 2-component version of Sison & Murray (2019).

- Furthermore, we present substantially more details of our case study in Section 6, which were mostly elided from Sison & Murray (2019). These include formal statements of both the source-level properties preserved and the target-level properties obtained, alongside explanations of all alternative methods for obtaining the latter from the former.

- Finally, every lemma and theorem we prove is presented with a proof sketch or explanation, which were largely absent from Sison & Murray (2019). We also include here appendices with further details of our compiler (see Appendices A, B, and C).

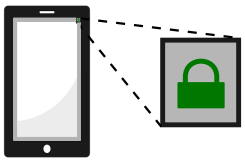
2 Noninterference and its refinement for mixed-sensitivity concurrent programs

To support mixed-sensitivity concurrent programs, we verify our compiler to preserve the *concurrent value-dependent noninterference* (CVDNI) notions of Murray *et al.* (2016b). In this section we present the definitions of CVDNI and its refinement, as we have adapted them from that work's Isabelle formalisation (Murray *et al.*, 2016c,a). In particular, the version of the theory we present here supports extra customisation of requirements beyond the prior work; we will need this to parameterise the theory with initial conditions needed for a compositionality property of our source language (Section 3), and our compiler's preservation of a ban on secret-dependent control flow (Section 5). Furthermore, it is simplified to the case where the shared memory is the same for both the original *abstract* and the refined *concrete* program—refinement adds no new shared variables. Later, we will instantiate this CVDNI theory to have our compiler's source and target languages (Sections 3, 4) respectively play the roles of the abstract and concrete programs' languages in the theory.

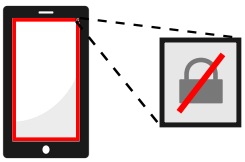
We begin by introducing with an illustrative example the challenges of verifying value-dependent noninterference in the presence of shared-variable concurrency (Section 2.1). Then we present the per-thread and whole-system noninterference properties themselves

```
while TRUE do
    lock(workspace_lock);
    while !suspended do
        lock(source_lock);
        workspace := source;
        /* ... operations on workspace ... */
        if domain = LOW then
            low_sink := workspace
        else
            high_sink := workspace;
            workspace := 0
        fi;
        unlock(source_lock)
    od;
    unlock(workspace_lock);
    while suspended do skip od
od
```

(a) Input processing worker thread program



(b) The phone providing the High personality: *domain* $\neq$ LOW, and *source* is classified High to reflect that the user might type in secrets.



(c) The phone displaying visual indicators that it is providing the Low personality: *domain* = LOW, and *source* is classified Low to reflect that we trust the user not to type in secrets.

Figure 1: Example: Touchscreen input processing for a dual-personality smartphone. Reproduced from Sison & Murray (2019).

(Section 2.2), followed by the notion of *per-thread refinement* that preserves the per-thread property between the two languages (Section 2.3). As the cube-shapedness of noninterference-preserving refinement diagrams in general makes them difficult to apply directly to compiler verification, we present a decomposition principle (Section 2.4) that we will use to prove CVDNI-preserving refinement for our compiler. We then present requirements and a theorem for *whole-system refinement* by which we have that CVDNI-preserving refinement is compositional across the threads of the program being compiled, such that it yields the whole-system property at the target language level (Section 2.5).

2.1 Illustrative example of a mixed-sensitivity concurrent program

Consider the task of verifying a multithreaded system that manages the user interface (UI) for a *dual-personality smartphone*, a phone that provides clearly distinguished user contexts (*personalities*), typically for work versus leisure. Specifically, our task is to verify that it does not leak *sensitive* information intended only for one of those personalities, which we classify High (Figure 1b), to locations belonging to the other, which we classify Low (Figure 1c).

Here and generally throughout this paper, our *attacker model* is an entity that can read from the system's *untrusted sinks*: some subset of permanently Low-classified locations not subject to synchronisation. In our example, the untrusted sinks may include WLAN device registers in a hostile environment.

The smartphone's UI system consists of a number of threads running concurrently with a shared address space; we aim to verify that, as a whole, this system of threads satisfies the security requirement. However, to avoid a state space explosion that is exponential in the

number of threads, we must do this *compositionally*: one thread at a time, then combining the results of these analyses.

We focus on a particular worker thread (Figure 1a), the one responsible for sending touchscreen input from the *source* variable to its intended destination.

The first challenge is that the destination depends on which personality the phone is currently providing, which is indicated by the value of *domain*. This is reflected by the classification of *source* being dependent on the value of *domain*: *source* is classified Low exactly when $domain = \mathsf{LOW}$ (where LOW is a designated constant), and is classified High otherwise. Due to this dependency, *domain* is known as a *control variable* of *source*.

The second challenge is the worker thread runs in a shared address space that might be accessed or modified by other threads, for various purposes. One of these threads may be responsible for maintaining that $domain = \mathsf{LOW}$ exactly when the phone indicates it is providing the Low personality (Figure 1c), so the user knows not to type in anything sensitive. Another thread may be responsible for assigning $suspended := \mathsf{TRUE}$ when the user turns the phone's screen off, to make the worker stop processing touchscreen input. We may then wish for *workspace* to be usable by some other thread—for example, processing input from a fingerprint scanner—in such a way that it can assume that *workspace* no longer contains any sensitive values.

When we analyse one thread like this worker in terms of our compositional security property (Section 2.2), all the other threads in the system are trusted to do two things:

1. They follow a *synchronisation discipline*; in particular for this example, this is a *mutual exclusion (mutex) locking discipline*: If read- or write-access to a certain variable is governed by a lock, each thread may only access the variable in that manner if they hold that lock. Mutual exclusion then follows from the semantics of the locking primitives ensuring only one thread may hold a given lock at a time.

2. They themselves do not leak values from High-classified locations (we refer to such values themselves as High) to Low-classified locations that are read-accessible to other threads. Note that, here, it is our objective to prove that the thread we are analysing can be trusted in the same way.

Even under these assumptions, the concurrency gives rise to some tricky considerations.

First, it is important that no thread in the system (including the one under analysis) modifies any control variables carelessly. For example, writing $domain := \mathsf{LOW}$ immediately after the worker reads a High value from *source*, will cause it to leak to *low_sink*. To prevent this, the worker uses *source_lock*, granting it *exclusive write-access* to *source* and *domain*.

Furthermore, as noted above, we may want to ensure that a *non-attacker-observable* location is nevertheless cleared of any sensitive values before being used by another thread. In our example, we classify *workspace* Low for the analysis to enforce this when the worker is suspended, but as the worker sometimes uses it to process High values, it is important to know *workspace* is accessible only to the worker during that time. To ensure this, the worker uses *workspace_lock*, granting it *exclusive read- and write-access* to *workspace*. It is then responsible for clearing it of any High values by the time it releases that access.

2.2 Concurrent value-dependent notions of noninterference

Having illustrated the challenges with an example, we now present the definitions of per-thread and whole-system noninterference, the theorem by which the former composes into the latter, and the compositionality side conditions demanded by that theorem.

As proved for each thread, CVDNI is defined by Murray *et al.* (2016b) in terms of:

1. A binary *strong low-bisimulation (modulo modes)* relation $\mathscr{B}$ between program configurations, which serves as witness to CVDNI. In the style of other low-bisimulation–based noninterference definitions (Focardi *et al.*, 1995; Sabelfeld & Sands, 2000; Mantel *et al.*, 2011) it requires the program configurations it relates to agree on their "low"-observable portions, and demands that lock-step execution preserves that correspondence. Furthermore, it is rely–guarantee-style concurrency aware, following Mantel *et al.* (2011), but modified to allow value-dependent classifications (Murray, 2015) for mixed-sensitivity reuse (see next point).

2. A *classification* function $\mathscr{L}$ that determines the "low"-observable portion of a program configuration, thus affecting $\mathscr{B}$'s requirements. The innovation of $\mathscr{L}$, as parameterised first by Murray (2015) and then by Murray *et al.* (2016b) as reproduced here, is that $\mathscr{L}$ can depend on values in the program configuration itself, thus expressing dynamic and not just static classifications.

The theory is parameterised over the type of values *Val*, a finite set of shared variables *Var*, and a *deterministic evaluation step semantics* $\rightsquigarrow$ between *local configurations* of a thread in a concurrent program. Each local configuration is a triple $\langle tps, mds, mem \rangle$:

- $tps :: ThreadPrivate$ is the *thread-private state*, which the theory will consider to be permanently inaccessible to the attacker and not shared with the other threads. Note that, due to this inaccessibility, we allow the user of the theory to parameterise the type *ThreadPrivate*, and we do not impose any particular structure on it.

- $mds :: Mode \Rightarrow Var\ set$ is the *(assume–guarantee) mode state*, which is ghost state associating each of $Mode \triangleq \{\mathbf{AsmNoW}, \mathbf{AsmNoRW}, \mathbf{GuarNoW}, \mathbf{GuarNoRW}\}$ with a set of shared variables. Intuitively, it identifies the set of variables for which the thread currently **Assume**s it possesses (or **Guar**antees it respects) exclusive permission to **W**rite (or **R**ead and **W**rite), granted (or obligated) for those variables typically by some synchronisation scheme. This facilitates compositional, rely–guarantee-style reasoning about such access (Jones, 1981; Mantel *et al.*, 2011).

  For example, when our worker thread (of Figure 1a) holds *source_lock*, it *assumes that no other threads write* to *source* or its control variable *domain* (i.e. $\{source, domain\} \subseteq mds$ **AsmNoW**), otherwise it *guarantees it does not write* to them (**GuarNoW**). Similarly, when it holds *workspace_lock* it assumes that no other threads *read or write* to *workspace* (i.e. $workspace \in mds$ **AsmNoRW**), and at all other times it makes the corresponding guarantee (**GuarNoRW**).

- $mem :: Mem$ is *shared memory* considered potentially accessible to the attacker and other threads. To make what is accessible amenable to analysis, we impose the structure $Mem \triangleq Var \Rightarrow Val$, a total map from shared variable names to values.

The theory is then further parameterised by the value-dependent classification function $\mathscr{L} :: Mem \Rightarrow Var \Rightarrow \{\mathsf{High}, \mathsf{Low}\}$, inducing a function $\mathscr{C}\,\mathsf{vars} :: Var \Rightarrow Var\ set$ that returns all the control variables of a given variable. In our worker thread example, $\mathscr{L}\ mem\ x$ gives:

- High when x is *high_sink*, meaning *high_sink* is classified High at all times.
- when x is *source*: Low if *mem domain* $=$ LOW, and High otherwise.
- Low for all other variables x, meaning they are classified Low at all times.

The set $\mathscr{C} = \{y \mid \exists x.\ y \in \mathscr{C}\,\mathsf{vars}\ x\}$ is then defined to contain all control variables in the system. Thus in our worker thread example, $\mathscr{C}\,\mathsf{vars}\ source = \{domain\}$ and $\mathscr{C} = \{domain\}$.

With these parameters having been set, we can now define notions of *observational equivalence*—underpinning noninterference properties—that are value dependent.

The notion of observational equivalence of memories, used by the *whole-system* noninterference property to quantify over initial state pairs, is as follows: Variables that are value-dependently classified Low *according to both memories* are required to have the same value *in both memories*. Formally, as defined originally by Murray (2015):

Definition 2.1 (Low-equivalent memories)**.**

$$mem_1 =^{\mathsf{Low}} mem_2 \triangleq \forall x.\ \mathscr{L}\ mem_1\ x = \mathsf{Low} \longrightarrow mem_1\ x = mem_2\ x$$

Note that the asymmetry of Definition 2.1 (also Definition 2.3 to follow) referring only to mem_1 is resolved by requiring the classification function $\mathscr{L}$ to classify all control variables as Low *statically*—that is, Low *always*, regardless of the memory state (cf. our restriction Proposition 3.4 on the classification of state used to implement locks, later in Section 3.2).

To support compositionality for concurrent programs, however, the equivalence notion for the *per-thread* noninterference property is relaxed to be *modulo modes* in the style of Mantel *et al.* (2011): Here, Low-classified non-control variables $x \notin \mathscr{C}$ are only required to have the same value if they are assumed to be *readable* by other threads according to the mode state. (Control variables $x \in \mathscr{C}$ are excluded from that relaxation, and are *always* required to be equal.) Defined more formally, again as originally by Murray (2015):[1]

Definition 2.2 (Readability of variable x, according to mode state *mds*)**.**

$$\mathsf{readable}\ mds\ x \triangleq x \notin mds\ \textbf{AsmNoRW}$$

Definition 2.3 (Low-equivalence of memories, modulo the mode state *mds*)**.**

$$mem_1 =^{\mathsf{Low}}_{mds} mem_2 \triangleq$$
$$\forall x.\ x \in \mathscr{C} \vee \mathscr{L}\ mem_1\ x = \mathsf{Low} \wedge \mathsf{readable}\ mds\ x \longrightarrow mem_1\ x = mem_2\ x$$

Moreover, we will use notation $lc_1 =^{\mathsf{Low}}_{\mathsf{mds}} lc_2$ from Sison & Murray (2019) to lift Definition 2.3 to local program configurations, asserting also that the local configurations lc_1 and lc_2 have the same assume–guarantee mode state. Additionally, we will use notation $lc_1 =_{\mathsf{mds}} lc_2$ to denote (only) that lc_1 and lc_2 have the same assume–guarantee mode state.

[1] Logical operator precedence here is just as in Isabelle/HOL—from most tightly to least: $\wedge, \vee, \longrightarrow$.

Thus, intuitively, the user of the theory should model the permanent untrusted output sinks, of their whole concurrent program, as variables for which $\mathscr{L}$ *always returns* Low, ungoverned by any synchronisation scheme that the attacker cannot be trusted to follow. In our worker example program (of Figure 1a), *low_sink* is untrusted permanently in this way, but *workspace* is untrusted only when unlocked.

We now have almost enough definitions to state the per-thread compositional security property. This property will assert the existence of a witness *bisimulation* relation $\mathscr{B}$ for every possible observationally equivalent pair of starting configurations. Specifically, this witness relation must be a *strong low-bisimulation (modulo modes)* (denoted by $\mathsf{strong\text{-}low\text{-}bisim\text{-}mm}\ \mathscr{B}$), meaning that it must satisfy the following three conditions:

1. It must maintain observational indistinguishability by requiring that all configuration pairs it relates (i.e. $(lc_1, lc_2) \in \mathscr{B}$) that have the same mode state ($lc_1 =_{\mathsf{mds}} lc_2$), are low-equivalent modulo modes ($lc_1 =^{\mathsf{Low}}_{\mathsf{mds}} lc_2$).

2. Furthermore, it must be a *bisimulation* by being symmetric (denoted by $\mathsf{sym}\ \mathscr{B}$) and *progressing to itself*: Any step taken by one of the configurations ($lc_1 \rightsquigarrow lc_1'$) must be matched by some step taken by the configuration related to it ($lc_2 \rightsquigarrow lc_2'$), so the destinations remain related (i.e. $(lc_1', lc_2') \in \mathscr{B}$) and modes-equal ($lc_1' =_{\mathsf{mds}} lc_2'$).

3. Finally, it must be *closed under globally consistent changes* made to memory by other threads (denoted by $\mathsf{cg\text{-}consistent}\ \mathscr{B}$)—that is, changes that preserve low-equivalence and are permitted by the current mode state *mds*. Specifically, other threads are permitted to change either of variable x's value or its classification only when x is considered *writable* by the current mode state (denoted by $\mathsf{writable}\ mds\ x$, Definition 2.5). This is the most crucial element of the per-thread CVDNI property itself that ensures its compositionality for concurrent programs.

These requirements are formalised by Definition 2.4, using Definitions 2.5 and 2.6:

Definition 2.4 (Strong low bisimulation, modulo modes)**.**

$$\begin{aligned}&\mathsf{strong\text{-}low\text{-}bisim\text{-}mm}\ \mathscr{B} \triangleq \mathsf{cg\text{-}consistent}\ \mathscr{B} \wedge \mathsf{sym}\ \mathscr{B} \wedge \\ &\quad (\forall lc_1\ lc_2.\ (lc_1, lc_2) \in \mathscr{B} \wedge lc_1 =_{\mathsf{mds}} lc_2 \longrightarrow \\ &\qquad lc_1 =^{\mathsf{Low}}_{\mathsf{mds}} lc_2 \wedge \\ &\qquad (\forall lc_1'.\ lc_1 \rightsquigarrow lc_1' \longrightarrow\ (\exists lc_2'.\ lc_2 \rightsquigarrow lc_2' \wedge lc_1' =_{\mathsf{mds}} lc_2' \wedge (lc_1', lc_2') \in \mathscr{B})))\end{aligned}$$

Definition 2.5 (Writability of variable x, according to mode state *mds*)**.**

$$\mathsf{writable}\ mds\ x \triangleq x \notin mds\ \mathbf{AsmNoW} \wedge x \notin mds\ \mathbf{AsmNoRW}$$

Definition 2.6 (Closedness under globally consistent changes)**.**

$$\begin{aligned}&\mathsf{cg\text{-}consistent}\ \mathscr{B} \triangleq \forall tps_1\ mem_1\ tps_2\ mem_2\ mds.\\ &\quad (\langle tps_1, mds, mem_1\rangle, \langle tps_2, mds, mem_2\rangle) \in \mathscr{B} \longrightarrow\\ &\quad (\forall mem_1'\ mem_2'.\ (\forall x.\ (mem_1\ x \neq mem_1'\ x \vee mem_2\ x \neq mem_2'\ x \vee\\ &\qquad \mathscr{L}\ mem_1\ x \neq \mathscr{L}\ mem_1'\ x) \longrightarrow \mathsf{writable}\ mds\ x) \wedge mem_1' =^{\mathsf{Low}}_{mds} mem_2' \longrightarrow\\ &\quad (\langle tps_1, mds, mem_1'\rangle, \langle tps_2, mds, mem_2'\rangle) \in \mathscr{B})\end{aligned}$$

Note that, to prevent unnecessary proof effort, strong-low-bisim-mm assumes instead of asserting the initial modes-equality ($lc_1 =_{\mathsf{mds}} lc_2$), as the security property that will use strong-low-bisim-mm will take responsibility for asserting it (to follow, in Definition 2.7).

We now present definitions of the CVDNI security properties that differ from those published in Murray *et al.* (2016b) and our conference paper Sison & Murray (2019), in that they allow two additional forms of customisation as parameters to the theory, necessary for a fuller written presentation of the formal verification of our compiler:

1. Initialisation requirements for the system, in the form of a predicate over shared memory called *INIT*.

   The per-thread and whole-system security properties are *relaxed* such that they only quantify over initial shared memories that obey this predicate.

2. Extra requirements to be imposed on top of strong low-bisimulation modulo modes, in the form of a predicate over bisimulation relations called *EXTRA*.

   The per-thread security property is *strengthened* to impose these additional requirements on any candidate security witness.

When dropped from each of the names of the properties "com-secure" and "sys-secure" soon to be introduced, *INIT* and *EXTRA* default to $(\lambda\_\,.\ \mathsf{True})$; in that case, the definitions of those properties will then simplify to their original versions as presented in Murray *et al.* (2016b); Sison & Murray (2019).

The per-thread security property is then as follows:

Definition 2.7 (Per-thread compositional security, with *INIT*, *EXTRA* requirements)**.**

$$\begin{aligned}&\mathsf{com\text{-}secure}^{EXTRA}_{INIT}\ (tps, mds)\ \triangleq\ \forall mem_1\ mem_2.\\ &\quad mem_1 =^{\mathsf{Low}}_{mds} mem_2\ \wedge\ INIT\ mem_1\ \wedge\ INIT\ mem_2\ \longrightarrow\\ &\quad\quad (\exists \mathscr{B}.\ \mathsf{strong\text{-}low\text{-}bisim\text{-}mm}\ \mathscr{B}\ \wedge\ EXTRA\ \mathscr{B}\ \wedge\\ &\quad\quad\quad (\langle tps, mds, mem_1\rangle, \langle tps, mds, mem_2\rangle) \in \mathscr{B})\end{aligned}$$

We have proved in Isabelle/HOL that the compositionality theorem of Murray *et al.* (2016b) holds regardless of the *INIT*, *EXTRA* chosen—in short, the *INIT* condition relaxes the goal sufficiently to relax each of its assumptions the same way, and the *EXTRA* requirement only strengthens its assumptions. Subject to some "sound mode use" side conditions (to be discussed soon), it gives us that the parallel composition *cms* :: $(ThreadPrivate \times (Mode \Rightarrow Var\ set))\ list$ of com-secure program threads will itself be a concurrent program that enforces "sys-secure", a system-wide value-dependent noninterference property. Here, the set operator returns the set of all the elements in a given list:

Theorem 2.8 (Compositionality of $\mathsf{com\text{-}secure}^{EXTRA}_{INIT}$)**.**

$$\frac{\begin{array}{c}\forall (tps, mds) \in \mathsf{set}\ cms.\ \mathsf{com\text{-}secure}^{EXTRA}_{INIT}\ (tps, mds)\\ \forall mem.\ INIT\ mem\ \longrightarrow\ \mathsf{sound\text{-}mode\text{-}use}\ (cms, mem)\end{array}}{\mathsf{sys\text{-}secure}_{INIT}\ cms}$$

We first introduce the elements of this whole-system property "sys-secure", before defining it formally (to follow, in Definition 2.9).

From all low-equivalent pairs of initial memories that both satisfy the *INIT* conditions, this whole-system property "sys-secure" asserts a form of low-equality between all global configuration pairs that are reachable via evaluation $\dashrightarrow_{sched}$ to *the same* fixed schedule *sched*, for all such finite lists *sched* giving an order of steps of execution from each thread:

$$gc \dashrightarrow_{[]} gc' \triangleq (gc = gc')$$
$$gc \dashrightarrow_{n.ns} gc' \triangleq (\exists gc''.\ gc \rightsquigarrow_n gc'' \wedge gc'' \dashrightarrow_{ns} gc')$$

Here [] is an empty list, $\_.\_$ is the cons operator, and $\rightsquigarrow_n$ means the nth thread in the global configuration takes one step.

In always comparing pairs of runs executing against the same schedule, the property models the class of schedulers whose decisions never depend on any secrets. Consequently, this excludes schedulers that are specialised, in the manner of Barthe *et al.* (2007a), to actively monitor the sensitivity level of each thread's control flow, so as to intervene and avoid interleaving it with others when it has become dependent on secrets. Instead, the CVDNI theory puts the onus on the developer of the program to prove that any branching on secret conditionals does not lead to timing-sensitive flows of the secret as discernible via low-classified sinks accessible to other threads in the system. Note that, as CVDNI prohibits mode state from ever becoming secret dependent, it will implicitly prohibit any leaks into parts of memory with which the mode state is directly associated—in Section 3, we will need to prohibit leaks into the memory we use to implement mutex locks, for this reason.

The special form of low-equality applied by the whole-system property is one that is modified from Definition 2.1, so that it only requires each Low-classified non-control variable $x \notin \mathscr{C}$ to be of equal value in both global configurations if the mode states of *all threads* consider x to be readable (Definition 2.2). Furthermore, the property ensures that paired global configurations continue to agree on the number of threads in the system, and on the mode states for all threads, written $cms'_1 =_{\mathsf{all\text{-}mds}} cms'_2 \triangleq (\mathsf{map\ mds}\ cms'_1 = \mathsf{map\ mds}\ cms'_2)$, where the syntax "map mds *cms*" denotes the mapped projection that extracts a list of mode states from a list *cms* of $ThreadPrivate \times (Mode \Rightarrow Var\ set)$ pairs. Finally, we will use syntax $cms[i]$ to denote the ith element in list *cms*.

This whole-system noninterference property, written formally, is then as follows:

Definition 2.9 (Whole-system value-dependent security, with *INIT* requirements)**.**

$$\begin{aligned}
&\mathsf{sys\text{-}secure}_{INIT}\ cms \triangleq \forall mem_1\ mem_2.\\
&INIT\ mem_1 \wedge INIT\ mem_2 \wedge mem_1 =^{\mathsf{Low}} mem_2 \longrightarrow\\
&\quad (\forall sched\ cms'_1\ mem'_1.\ (cms, mem_1) \dashrightarrow_{sched} (cms'_1, mem'_1) \longrightarrow\\
&\qquad (\exists cms'_2\ mem'_2.\ (cms, mem_2) \dashrightarrow_{sched} (cms'_2, mem'_2)) \wedge\\
&\qquad (\forall cms'_2\ mem'_2.\ (cms, mem_2) \dashrightarrow_{sched} (cms'_2, mem'_2) \longrightarrow\\
&\qquad\quad \mathsf{length}\ cms'_1 = \mathsf{length}\ cms'_2 \wedge cms'_1 =_{\mathsf{all\text{-}mds}} cms'_2 \wedge\\
&\qquad\quad (\forall x.\ x \in \mathscr{C} \vee \mathscr{L}\ mem'_1\ x = \mathsf{Low} \wedge\\
&\qquad\qquad (\forall i < \mathsf{length}\ cms'_1.\ \mathsf{readable}\ cms'_1[i]\ x) \longrightarrow mem'_1\ x = mem'_2\ x)))
\end{aligned}$$

Finally, we must note that the use of assume–guarantee reasoning to obtain the compositionality of the per-thread property in the style of Mantel *et al.* (2011) gives rise to requirements justifying the soundness of that reasoning; requirements that we will prove our compiler to preserve. For CVDNI, these are summed up by the "sound-mode-use" side condition of Theorem 2.8, which consists of a "local" and a "global" part:

Definition 2.10 (Sound mode use side-condition)**.**

$$\begin{aligned}&\mathsf{sound\text{-}mode\text{-}use}\ (cms, mem) \triangleq\\ &\quad (\forall cm \in \mathsf{set}\ cms.\ \mathsf{local\text{-}mode\text{-}compliance}\ (cm, mem)) \wedge\\ &\quad \mathsf{global\text{-}modes\text{-}compatibility}\ (cms, mem)\end{aligned}$$

First, all threads must each obey a *local mode compliance* requirement. This says that for all reachable local configurations of the program, at no point will the thread violate any of its own guarantees not to access a particular location in the shared state, which implies also not accessing any of its control variables. We leave precise definitions for "reachable-lcs" and "doesnt-read-(or-modify)" to this paper's Isabelle/HOL supplement material,[2] but mention here that it is the doesnt-read-(or-modify) assertions that enforce that any guarantees not to access some variable x will effectively apply also to all of x's control variables:

Definition 2.11 (Local mode compliance)**.**

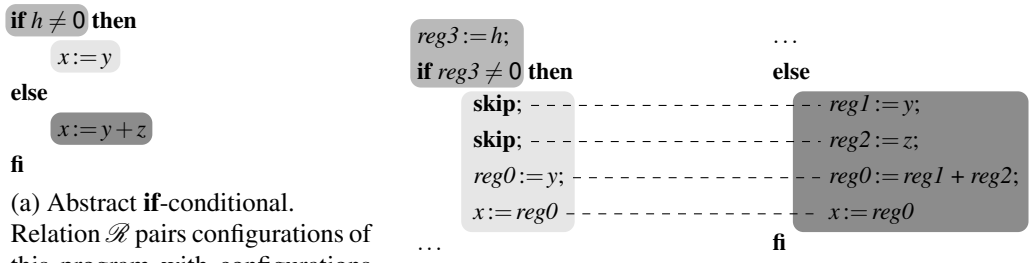
$$\begin{aligned}&\mathsf{local\text{-}mode\text{-}compliance}\ lc \triangleq\\ &\forall c\ mds\ mem.\ \langle c, mds, mem\rangle \in \mathsf{reachable\text{-}lcs}\ lc \longrightarrow \mathsf{respects\text{-}own\text{-}guarantees}\ (c, mds)\\ &\text{where}\\ &\quad \mathsf{respects\text{-}own\text{-}guarantees}\ (c, mds) \triangleq\\ &\qquad (\forall x.\ (x \in mds\ \mathbf{GuarNoRW} \longrightarrow \mathsf{doesnt\text{-}read\text{-}or\text{-}modify}\ c\ x) \wedge\\ &\qquad\quad (x \in mds\ \mathbf{GuarNoW} \longrightarrow \mathsf{doesnt\text{-}modify}\ c\ x))\end{aligned}$$

Then, all threads must together obey a *global modes compatibility* requirement. This requirement says that the threads' mode states in all reachable global configurations of the concurrent program (the "reachable-mds-lists") are *compatible*—that is, if any one thread assumes a particular location will not be accessed for writing or reading, then all other threads must be guaranteeing not to access that location for the same purpose:

Definition 2.12 (Global modes compatibility)**.**

$$\begin{aligned}&\mathsf{global\text{-}modes\text{-}compatibility}\ gc \triangleq \forall mdss \in \mathsf{reachable\text{-}mds\text{-}lists}\ gc.\ \mathsf{compatible\text{-}modes}\ mdss\\ &\text{where}\\ &\quad \mathsf{reachable\text{-}mds\text{-}lists}\ gc \triangleq\\ &\qquad \{mdss \mid \exists cms'\ mem'\ sched.\ gc \dashrightarrow_{sched} (cms', mem') \wedge \mathsf{map}\ \mathsf{mds}\ cms' = mdss\}\\ &\quad \mathsf{compatible\text{-}modes}\ mdss \triangleq \forall i\ x.\ i < \mathsf{length}\ mdss \longrightarrow\\ &\qquad (x \in mdss[i]\ \mathbf{AsmNoRW} \longrightarrow\\ &\qquad\quad (\forall j < \mathsf{length}\ mdss.\ j \neq i \longrightarrow x \in mdss[j]\ \mathbf{GuarNoRW})) \wedge\\ &\qquad (x \in mdss[i]\ \mathbf{AsmNoW} \longrightarrow\\ &\qquad\quad (\forall j < \mathsf{length}\ mdss.\ j \neq i \longrightarrow x \in mdss[j]\ \mathbf{GuarNoW}))\end{aligned}$$

[2]The Isabelle/HOL theories are available at `http://covern.org/jfpsc.html`.



(a) Abstract **if**-conditional. Relation $\mathscr{R}$ pairs configurations of this program with configurations of the program in Figure 2b that are of the same-shaded region.

(b) Concrete **if**-conditional. Relation $\mathscr{I}$ pairs configurations of this program as shown by the dashed lines.

Figure 2: Excerpts from a CVDNI-preserving refinement example with secret-dependent control flow: h contains a secret, y and z contain zero, and x is an untrusted sink. Reproduced from Sison & Murray (2019)—the example is originally from Murray *et al.* (2016b).

Note that this global modes compatibility requirement is *not compositional*; consequently, instead of obliging the program developer to prove it for the source programs to be fed to our compiler, we will prove it as an invariant maintained by the execution semantics of our source language—particularly, by its synchronisation primitives (see Section 3).

For more details and precise presentations of all the definitions we have adapted from Murray *et al.* (2016b,c) to enable the compiler verification work described in this paper, please refer to the Isabelle/HOL formalisation in our supplement material.

2.3 Cube-shaped refinement for preserving noninterference

Proof of *CVDNI-preserving refinement* (also *security-preserving* or *secure refinement*), for a single-threaded program that will be run as a thread of a concurrent program, requires the user of the theory to nominate two binary relations (both illustrated by Figure 2):

1. A *refinement relation* $\mathscr{R}$ relating local configurations of the abstract program to local configurations of the concrete program: Abstract must simulate concrete, in a sense typical of much other work on program refinement, including compiler verification.

2. A *concrete coupling invariant* $\mathscr{I}$ that allows us to use $\mathscr{B}$ and $\mathscr{R}$ to build a new strong low-bisimulation (modulo modes) for the concrete program, by discarding pairs of local configurations *after the refinement* that should not be reached in the same number of evaluation steps. It thereby witnesses that any changes a refinement (or compiler) might make to the execution time do not introduce any timing channels.

The essence of the proof technique is to require that a number of conditions—analogous to those for strong-low-bisim-mm (Definition 2.4)—be imposed on the nominated $\mathscr{R}$ and $\mathscr{I}$, in relation to a given witness relation $\mathscr{B}$ establishing com-secure (Definition 2.7) for the abstract program. The definitions to follow are adapted from Murray *et al.* (2016b) Section V, as we presented in Sison & Murray (2019)—for better readability, a simplified version in which no new shared variables are added by the refinement. Consequently, we use the notation $=^{\mathsf{mem}}_{\mathsf{mds}}$ to denote that two local configurations have equal mode state and memory, regardless of whether relating configurations of the same or differing languages.

$$
\begin{aligned}
&\mathsf{coupling\text{-}inv\text{-}pres}\ \mathscr{B}\ \mathscr{R}\ \mathscr{I} \triangleq \\
&\forall \mathbf{a}_1\ \mathbf{c}_1.\ (\mathbf{a}_1,\mathbf{c}_1) \in \mathscr{R} \longrightarrow \\
&\quad (\forall \mathbf{c}_1'.\ \mathbf{c}_1 \rightsquigarrow_{\mathsf{C}} \mathbf{c}_1' \longrightarrow \\
&\qquad (\exists n\ \mathbf{a}_1'.\ \mathbf{a}_1 \rightsquigarrow_{\mathsf{A}}^{n} \mathbf{a}_1' \wedge (\mathbf{a}_1',\mathbf{c}_1') \in \mathscr{R} \wedge \\
&\qquad\quad (\forall \mathbf{a}_2\ \mathbf{c}_2\ \mathbf{a}_2'.\ (\mathbf{a}_1,\mathbf{a}_2) \in \mathscr{B} \wedge \mathbf{a}_1 =_{\mathsf{mds}} \mathbf{a}_2 \wedge \\
&\qquad\qquad (\mathbf{a}_2,\mathbf{c}_2) \in \mathscr{R} \wedge (\mathbf{c}_1,\mathbf{c}_2) \in \mathscr{I} \wedge \mathbf{c}_1 =_{\mathsf{mds}} \mathbf{c}_2 \wedge \\
&\qquad\qquad \mathbf{a}_2 \rightsquigarrow_{\mathsf{A}}^{n} \mathbf{a}_2' \wedge \mathbf{a}_1' =_{\mathsf{mds}} \mathbf{a}_2' \longrightarrow \\
&\qquad\qquad\quad (\exists \mathbf{c}_2'.\ \mathbf{c}_2 \rightsquigarrow_{\mathsf{C}} \mathbf{c}_2' \wedge \mathbf{c}_1' =_{\mathsf{mds}} \mathbf{c}_2' \wedge \\
&\qquad\qquad\qquad (\mathbf{a}_2',\mathbf{c}_2') \in \mathscr{R} \wedge (\mathbf{c}_1',\mathbf{c}_2') \in \mathscr{I}))))
\end{aligned}
$$

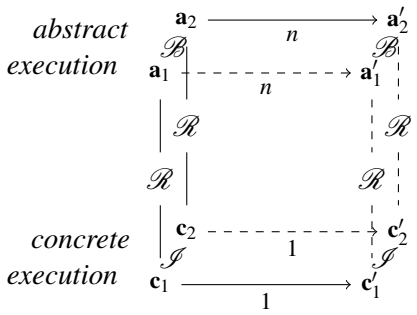


Figure 3: Definition and graphical depiction of refinement preservation obligation for secure-refinement (Definition 2.15). Reproduced from Sison & Murray (2019)—the definition is a simplified restatement of its original formalisation in Murray *et al.* (2016b).

Regarding the maintenance of modes equivalence and observational equivalence across the relation, the restrictions on refinement are tighter than those that were applied to strong-low-bisim-mm, in that $\mathscr{R}$ is required to preserve the shared memory in its entirety:

Definition 2.13 (Preservation of modes and memory)**.**

$$\mathsf{preserves\text{-}modes\text{-}mem}\ \mathscr{R} \triangleq \forall lc_A\ lc_C.\ (lc_A,lc_C) \in \mathscr{R} \longrightarrow lc_A =_{\mathsf{mds}}^{\mathsf{mem}} lc_C$$

Regarding the closedness under changes by other threads that ensures compositionality for concurrency, on $\mathscr{I}$ we again impose cg-consistent (Definition 2.6) from Section 2.2. However, in the case of $\mathscr{R}$, we instead impose "closed-others", a simplification of cg-consistent that considers only environmental actions that affect the memories on both sides of the relation identically. Furthermore, closed-others ensures equality of *all* shared variables, not just those judged observable. Defined formally:

Definition 2.14 (Closedness of refinements under changes by others)**.**

$$
\begin{aligned}
&\mathsf{closed\text{-}others}\ \mathscr{R} \triangleq \forall tps_A\ tps_C\ mds\ mem\ mem'. \\
&\quad (\langle tps_A, mds, mem\rangle_{\mathsf{A}}, \langle tps_C, mds, mem\rangle_{\mathsf{C}}) \in \mathscr{R}) \wedge \\
&\quad (\forall x.\ (mem\ x \neq mem'\ x \vee \mathscr{L}\ mem\ x \neq \mathscr{L}\ mem'\ x) \longrightarrow \mathsf{writable}\ mds\ x) \longrightarrow \\
&\quad (\langle tps_A, mds, mem'\rangle_{\mathsf{A}}, \langle tps_C, mds, mem'\rangle_{\mathsf{C}}) \in \mathscr{R})
\end{aligned}
$$

The final major—and hardest—requirement for confidentiality preservation is to prove $\mathscr{R}$ and $\mathscr{I}$ closed simultaneously under the pairwise executions of the concrete and abstract programs, using a cube-shaped "refinement and coupling invariant preservation" diagram (coupling-inv-pres, depicted in Figure 3), whose edges are configuration pairs in $\mathscr{B}$, $\mathscr{R}$, and $\mathscr{I}$. (Reducing its difficulty is the focus of the decomposition principle in Section 2.4.)

All that then remains is for the nominated concrete coupling invariant $\mathscr{I}$ to be symmetric (sym $\mathscr{I}$), and the predicate secure-refinement puts together all the requirements:

Definition 2.15 (Requirements for confidentiality-preserving secure refinement)**.**

$$
\begin{aligned}
\mathsf{secure\text{-}refinement}\ \mathscr{B}\ \mathscr{R}\ \mathscr{I} \triangleq\ & \mathsf{preserves\text{-}modes\text{-}mem}\ \mathscr{R} \wedge \mathsf{closed\text{-}others}\ \mathscr{R} \wedge \\
& \mathsf{cg\text{-}consistent}\ \mathscr{I} \wedge \mathsf{sym}\ \mathscr{I} \wedge \mathsf{coupling\text{-}inv\text{-}pres}\ \mathscr{B}\ \mathscr{R}\ \mathscr{I}
\end{aligned}
$$

The soundness theorem for confidentiality-preserving refinement by Murray *et al.* (2016b) then gives us that, under these conditions, the concrete relation "$\mathscr{B}_{\mathsf{C}}\mathsf{of}\ \mathscr{B}\ \mathscr{R}\ \mathscr{I}$", derived from a witness strong-low-bisim-mm relation $\mathscr{B}$, refinement relation $\mathscr{R}$, and coupling invariant $\mathscr{I}$, is itself a witness strong-low-bisim-mm for the concrete program. For readability, from here onwards we will use $\mathbf{a}_1, \mathbf{c}_1, \ldots$ instead of lc_{1A}, $lc_{1C}, \ldots$ for local configuration variables when comparing abstract and concrete executions simultaneously:

Definition 2.16 (Concrete bisimulation relation derived from $\mathscr{B}$,$\mathscr{R}$ and $\mathscr{I}$)**.**

$$\mathscr{B}_{\mathsf{C}}\mathsf{of}\ \mathscr{B}\ \mathscr{R}\ \mathscr{I} \triangleq \{(\mathbf{c}_1,\mathbf{c}_2) \mid \exists \mathbf{a}_1\ \mathbf{a}_2.\ (\mathbf{a}_1,\mathbf{c}_1) \in \mathscr{R} \wedge (\mathbf{a}_2,\mathbf{c}_2) \in \mathscr{R} \wedge (\mathbf{a}_1,\mathbf{a}_2) \in \mathscr{B} \wedge \mathbf{c}_1 =^{\mathsf{Low}}_{\mathsf{mds}} \mathbf{c}_2 \wedge (\mathbf{c}_1,\mathbf{c}_2) \in \mathscr{I}\}$$

Theorem 2.17 (Preservation of strong-low-bisim-mm by secure-refinement)**.**

$$\frac{\mathsf{strong\text{-}low\text{-}bisim\text{-}mm}\ \mathscr{B} \qquad \mathsf{secure\text{-}refinement}\ \mathscr{B}\ \mathscr{R}\ \mathscr{I}}{\mathsf{strong\text{-}low\text{-}bisim\text{-}mm}\ (\mathscr{B}_{\mathsf{C}}\mathsf{of}\ \mathscr{B}\ \mathscr{R}\ \mathscr{I})}$$

2.4 Decomposition principle and its impact on refinement proofs

We now present, as we first did in Sison & Murray (2019), an alternative way to prove secure-refinement (Definition 2.15) that obviates the need to use the cube-shaped, two-sided refinement obligation (depicted by Figure 3), by decomposing its concerns into:

1. Proving $\mathscr{R}$ closed using a square-shaped simulation diagram (depicted by Figure 4a) akin to the *backward simulations* commonly used to prove semantics-preserving refinement by compilers (e.g. for CompCert (Leroy, 2009)), and

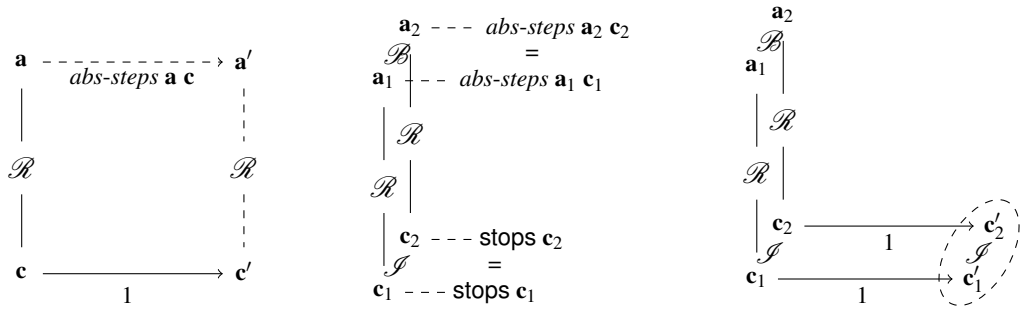
2. Security-focused proof obligations (depicted by Figures 4b, 4c), separable from the square-shaped simulation, that prevent the introduction of *timing leaks*, *termination leaks*, and secret-dependent differences in assume–guarantee mode state.

The decomposition requires the verifier to nominate a new parameter, called *abs-steps* or the *pacing function*. Its role is to dictate the pace of the square-shaped simulation by specifying the number of abstract steps that ought to be taken for one concrete step, as depicted by Figure 4a. Deferring the security-focused side conditions ("decomp-refinement-safe") to afterwards, the decomposition principle is then defined formally as follows:

Definition 2.18 (Decomposition principle for secure-refinement)**.**

$$\begin{aligned}&\mathsf{secure\text{-}refinement\text{-}decomp}\ \mathscr{B}\ \mathscr{R}\ \mathscr{I}\ \textit{abs-steps} \triangleq\\ &\quad \mathsf{preserves\text{-}modes\text{-}mem}\ \mathscr{R} \wedge \mathsf{closed\text{-}others}\ \mathscr{R} \wedge \mathsf{cg\text{-}consistent}\ \mathscr{I} \wedge \mathsf{sym}\ \mathscr{I} \wedge\\ &\quad \mathsf{decomp\text{-}refinement\text{-}safe}\ \mathscr{B}\ \mathscr{R}\ \mathscr{I}\ \textit{abs-steps} \wedge (\forall \mathbf{a}\ \mathbf{c}.\ (\mathbf{a},\mathbf{c}) \in \mathscr{R} \longrightarrow\\ &\qquad (\forall \mathbf{c}'.\ \mathbf{c} \rightsquigarrow_{\mathsf{C}} \mathbf{c}' \longrightarrow (\exists \mathbf{a}'.\ \mathbf{a} \rightsquigarrow_{\mathsf{A}}^{(\textit{abs-steps}\ \mathbf{a}\ \mathbf{c})} \mathbf{a}' \wedge (\mathbf{a}',\mathbf{c}') \in \mathscr{R})))\end{aligned}$$

The aforementioned side conditions on all refinement parameters, depicted by Figures 4b, 4c, are then defined formally under the predicate decomp-refinement-safe as follows:



(a) Refinement preservation for relation $\mathscr{R}$ under program execution paced by *abs-steps*. (Part of Definition 2.18.)

(b) Consistency of pacing and stopping behaviour, to prevent timing and termination leaks. (Part of Definition 2.19.)

(c) Closedness of the coupling invariant relation $\mathscr{I}$ under lockstep program execution. (Part of Definition 2.19.)

Figure 4: Graphical depictions of decomposed refinement preservation obligations. Reproduced from Sison & Murray (2019).

Definition 2.19 (Security-focused side conditions for decomposition principle)**.**

$$\begin{aligned}&\mathsf{decomp\text{-}refinement\text{-}safe}\ \mathscr{B}\ \mathscr{R}\ \mathscr{I}\ \mathit{abs\text{-}steps} \triangleq \forall \mathbf{a}_1\ \mathbf{a}_2\ \mathbf{c}_1\ \mathbf{c}_2.\ (\mathbf{a}_1,\mathbf{a}_2) \in \mathscr{B} \wedge \\ &\mathbf{a}_1 =_{\mathsf{mds}} \mathbf{a}_2 \wedge (\mathbf{a}_1,\mathbf{c}_1) \in \mathscr{R} \wedge (\mathbf{a}_2,\mathbf{c}_2) \in \mathscr{R} \wedge (\mathbf{c}_1,\mathbf{c}_2) \in \mathscr{I} \wedge \mathbf{c}_1 =_{\mathsf{mds}} \mathbf{c}_2 \\ &\longrightarrow \mathsf{stops}\ \mathbf{c}_1 = \mathsf{stops}\ \mathbf{c}_2 \wedge \mathit{abs\text{-}steps}\ \mathbf{a}_1\ \mathbf{c}_1 = \mathit{abs\text{-}steps}\ \mathbf{a}_2\ \mathbf{c}_2 \wedge \\ &\quad (\forall \mathbf{c}'_1\ \mathbf{c}'_2.\ \mathbf{c}_1 \rightsquigarrow_{\mathsf{C}} \mathbf{c}'_1 \wedge \mathbf{c}_2 \rightsquigarrow_{\mathsf{C}} \mathbf{c}'_2 \longrightarrow (\mathbf{c}'_1,\mathbf{c}'_2) \in \mathscr{I} \wedge \mathbf{c}'_1 =_{\mathsf{mds}} \mathbf{c}'_2)\end{aligned}$$

The intuitive meanings of the side conditions in Definition 2.19 are:

- $\mathsf{stops}\ \mathbf{c}_1 = \mathsf{stops}\ \mathbf{c}_2$ ensures that the refinement has not introduced any termination leaks, by asserting *consistent stopping behaviour* for $\mathscr{I}$-related concrete program configurations, which we know to be observationally indistinguishable.
- $\mathit{abs\text{-}steps}\ \mathbf{a}_1\ \mathbf{c}_1 = \mathit{abs\text{-}steps}\ \mathbf{a}_2\ \mathbf{c}_2$ ensures that the refinement has not introduced any timing leaks, by asserting *consistency of the pace of the refinement* for $\mathscr{R}$-related program configurations, which we again know to be observationally indistinguishable.
- The final $\forall$-quantified clause asserts $\mathscr{I}$'s suitability as a coupling invariant, in that it must remain *closed under lockstep evaluation* of the concrete program configurations it relates. Furthermore it must *maintain mode state equality* with each lockstep evaluation, which ensures that the refinement has not introduced any inconsistencies in the memory access assumptions and guarantees needed for the concurrent compositionality of the property.

Note that the $\mathscr{B}$- and $\mathscr{R}$-edges in Figure 4c may capture useful facts about a particular program verification technique and compiler (respectively), so their availability as assumptions is intended to reduce greatly the effort needed to specify a coupling invariant $\mathscr{I}$ and prove it satisfies the condition.

Assuming the fulfilment of all the decomposed requirements, we obtain that they are a sound method for establishing secure refinement of the per-thread confidentiality property, as desired:

Theorem 2.20 (Soundness of the decomposition principle)**.**

$$\textsf{secure-refinement-decomp}\ \mathscr{B}\ \mathscr{R}\ \mathscr{I}\ \textit{abs-steps} \implies \textsf{secure-refinement}\ \mathscr{B}\ \mathscr{R}\ \mathscr{I}$$

Proof. The only obligation for secure-refinement (Definition 2.15) not obtained immediately from secure-refinement-decomp (Definition 2.18) is the cube-shaped coupling-inv-pres (Figure 3). We discharge this as follows:

The front face of the cube is just ordinary square-shaped refinement preservation (depicted in Figure 4a), given to us by secure-refinement-decomp: that a single concrete step from $\mathbf{c}_1$ is simulated by n abstract steps from $\mathbf{a}_1$, where n is given by *abs-steps*.

We are then obliged to prove a simulation in the other direction (the back face of the cube), that n abstract steps from all configurations $\mathbf{a}_2$ related by $\mathscr{B}$ to $\mathbf{a}_1$ are simulated by some concrete step from $\mathbf{c}_2$ related by $\mathscr{R}$ to $\mathbf{a}_2$ and by $\mathscr{I}$ to $\mathbf{c}_1$.

Here, we lean on the determinism of the abstract program's evaluation semantics (required by the theory) to flip the direction of simulation, knowing that n abstract steps from $\mathbf{a}_2$, simulating a single concrete step from $\mathbf{c}_2$, could only be the very same n abstract steps from $\mathbf{a}_2$ that we were required to consider. This allows us to obtain that simulation by using, once again, the square-shaped refinement preservation (Figure 4a) given to us by secure-refinement-decomp.

Consistency of refinement pacing and stopping behaviour (depicted in Figure 4b) given by decomp-refinement-safe (Definition 2.19) then respectively ensure that n (via *abs-steps*) is the correct number of abstract steps to consider, and that there will indeed be a concrete step from $\mathbf{c}_2$ to drive the matching simulation step.

Finally, the remainder of decomp-refinement-safe (depicted in Figure 4c) discharges the requirement of closedness and modes-equality maintenance of $\mathscr{I}$ under lockstep execution, demanded by the bottom face of the cube. □

To demonstrate how the decomposition principle reduces proof complexity and effort, we returned to the example program refinement discussed in Section V-E of Murray *et al.* (2016b) and proved in its Isabelle formalisation (Murray *et al.*, 2016a), an excerpt of which is shown in Figure 2. The abstract program (9 imperative commands) branches on a sensitive value, and executes a single atomic expression assignment in each branch. Its refinement (to 16 commands) models expansion of the expressions into multiple steps, resolving a timing disparity between the two branches by padding with **skip**.

We use proof size as a proxy for proof effort, since the former is known to be strongly linearly correlated with the latter (Staples *et al.*, 2014). Formalised in Isabelle/HOL as `EgHighBranchRevC.thy` (Murray *et al.*, 2016a), the proof line count for that theory stood at about 4.6K lines of definitions and proof, of which approx. 3.6K line were proofs. Adapting the proof instead to use the decomposition principle (secure-refinement-decomp, Definition 2.18), the proof line count drops from 3.6K to approx. 2K, a 44% reduction. Regarding definition changes, the new proof makes less than 10 lines of adaptations to a coupling invariant and pacing function used by the old proof, and adds about 30 lines worth of new helper definitions, for use with the decomposition principle. The rest of the theory and its external dependencies remain in common between the two versions.

As would be expected, the bulk of the deletions are from the full cube-shaped refinement diagram proof (Figure 3) of secure-refinement (Definition 2.15) for the refinement relation. The surviving parts of that proof just become the square-shaped refinement diagram proof

(Figure 4a) of the decomposition principle (Definition 2.18), without much modification. The deletions are replaced by newly added proofs of the decomposition principle's more security-focused side conditions (Definition 2.19, depicted by Figures 4b, 4c).

2.5 Compositional whole-system secure refinement

We now present the whole-system refinement theorem from Murray *et al.* (2016b,a), which we adapt here to support the specification of *INIT* requirements (as in $\mathsf{com\text{-}secure}^{EXTRA}_{INIT}$, Definition 2.7), and simplify to the case of refinements that add no shared variables.

The main usefulness of this theorem is that, beyond demanding secure-refinement (Definition 2.15), which dealt with the preservation of per-thread security as witnessed by a strong-low-bisim-mm (Definition 2.4), it deals additionally with the preservation of the sound-mode-use side conditions (Definition 2.10) that will be demanded by the compositionality theorem for CVDNI (Theorem 2.8) at the target language level.

Notably, although it imposes the requirement for the refinement to preserve the "local" part of sound-mode-use (Definition 2.11), it *automatically* preserves the non-compositional "global" part of this side condition (Definition 2.12) as a consequence of the requirements imposed by the per-thread secure refinements. Thus, our source-level proof of the global condition (see Section 3) will be sufficient, and there will be no need for us to prove anything extra about our compiler for it to preserve that to the target-language level.

We now present the requirements and theorem for whole-system refinement formally.

First, in addition to the per-thread refinement notion secure-refinement (Definition 2.15) that we addressed in Sections 2.3 and 2.4, our whole-system refinement theorem will require that the refinement relation $\mathscr{R}$ established by the compiler additionally preserves the compositional local mode compliance property for each thread. Here, "respects-own-guarantees" is from Definition 2.11:

Definition 2.21 (Refinement $\mathscr{R}$ preserves local mode compliance)**.**

$$\begin{aligned}&\mathsf{preserves\text{-}local\text{-}compliance}\ \mathscr{R} \triangleq \forall tps_A\ mds_A\ mem_A\ tps_C\ mds_C\ mem_C.\\ &\quad \mathsf{respects\text{-}own\text{-}guarantees}\ (tps_A, mds_A) \wedge\\ &\quad (\langle tps_A, mds_A, mem_A\rangle_{\mathsf{A}}, \langle tps_C, mds_C, mem_C\rangle_{\mathsf{C}}) \in \mathscr{R} \longrightarrow\\ &\qquad \mathsf{respects\text{-}own\text{-}guarantees}\ (tps_C, mds_C))\end{aligned}$$

We define a new "compositional refinement" predicate to capture all per-thread requirements that will be demanded by our compositional whole-system refinement theorem. This bundles together preserves-local-compliance and secure-refinement so as to preserve the strong-low-bisim-mm relations (Definition 2.4) that witness noninterference for each thread of the abstract program. Alongside all these requirements just described, it also requires the concrete coupling invariant $\mathscr{I}$ to cover all possible initial memory pairs that are low-equal modulo modes (Definition 2.3) and satisfy the $INIT_C$ conditions that will parameterise the target language-level CVDNI property:

Definition 2.22 (Requirements for compositional whole-system refinement)**.**

$$\begin{aligned}
&\mathsf{compositional\text{-}refinement}\ \mathscr{B}\ \mathscr{R}\ \mathscr{I}\ \triangleq \\
&\quad \mathsf{secure\text{-}refinement}\ \mathscr{B}\ \mathscr{R}\ \mathscr{I}\ \wedge\ \mathsf{strong\text{-}low\text{-}bisim\text{-}mm}\ \mathscr{B}\ \wedge \\
&\quad \mathsf{preserves\text{-}local\text{-}compliance}\ \mathscr{R}\ \wedge \\
&\quad (\forall tps_C\ mds\ mem_1\ mem_2.\ mem_1 =^{\mathsf{Low}}_{mds} mem_2\ \wedge\ INIT_C\ mem_1\ \wedge\ INIT_C\ mem_2\ \longrightarrow \\
&\qquad (\langle tps_C, mds, mem_1\rangle_{\mathsf{C}}, \langle tps_C, mds, mem_2\rangle_{\mathsf{C}}) \in \mathscr{I})
\end{aligned}$$

With these requirements, we prove using Isabelle/HOL that a whole-system refinement theorem, proved originally by Murray *et al.* (2016b,a), can be adapted to support the specification of *INIT* requirements on initial memory at both abstract- and concrete-level. (As with Theorem 2.8, the relaxation of the goal by $INIT_C$ is enough to permit the relaxations of its assumptions by $INIT_C, INIT_A$.) First we will state the theorem, then we will explain it, line-by-line. This theorem proves that abstract-level sound-mode-use (including its global part) by a system of secure mixed-sensitivity concurrent program threads (i.e. list cms_A, as witnessed by bisimulations $\mathscr{B}\mathsf{s}$ for each thread) is sufficient for a set of per-thread secure refinements (in terms of the lists $\mathscr{B}\mathsf{s}, \mathscr{R}\mathsf{s}, \mathscr{I}\mathsf{s}$ of bisimulation, refinement, and concrete coupling invariant relations for each thread, respectively) to yield a concrete-level secure concurrent program (i.e. list cms_C that satisfies sys-secure, Definition 2.9):

Theorem 2.23 (Whole-system compositionality of per-thread secure refinement)**.**

$$\begin{aligned}
&(\forall mem.\ INIT_C\ mem\ \longrightarrow\ INIT_A\ mem)\ \wedge \\
&(\forall mem.\ INIT_A\ mem\ \longrightarrow\ \mathsf{sound\text{-}mode\text{-}use}\ (cms_A, mem))\ \wedge \\
&\mathsf{length}\ cms_A = \mathsf{length}\ \mathscr{B}\mathsf{s} = \mathsf{length}\ \mathscr{R}\mathsf{s} = \mathsf{length}\ \mathscr{I}\mathsf{s} = \mathsf{length}\ cms_C\ \wedge \\
&(\forall i < \mathsf{length}\ cms_C. \\
&\quad \mathsf{compositional\text{-}refinement}\ \mathscr{B}\mathsf{s}[i]\ \mathscr{R}\mathsf{s}[i]\ \mathscr{I}\mathsf{s}[i]\ \wedge \\
&\quad (\forall mem.\ INIT_C\ mem\ \longrightarrow ((cms_A[i], mem), (cms_C[i], mem)) \in \mathscr{R}\mathsf{s}[i])\ \wedge \\
&\quad (\forall mem_1\ mem_2.\ INIT_A\ mem_1\ \wedge\ INIT_A\ mem_2\ \wedge\ mem_1 =^{\mathsf{Low}}_{(\mathsf{snd}\ cms_C[i])} mem_2\ \longrightarrow \\
&\qquad ((cms_A[i], mem_1), (cms_A[i], mem_2)) \in \mathscr{B}\mathsf{s}[i]))
\end{aligned}$$

$$\mathsf{sys\text{-}secure}_{INIT_C}\ cms_C$$

The premises of this theorem can be understood as follows:

- $(\forall mem.\ INIT_C\ mem\ \longrightarrow\ INIT_A\ mem)$:
  The concrete-level "$INIT_C$" initial condition must be no weaker than the abstract-level "$INIT_A$" one.
- $(\forall mem.\ INIT_A\ mem\ \longrightarrow\ \mathsf{sound\text{-}mode\text{-}use}\ (cms_A, mem))$:
  For the abstract program, sound-mode-use (Definition 2.10) must hold for all possible initial memories.
- The lists of initial thread-private and mode states at abstract and concrete level (resp. cms_A, cms_C), and lists of bisimulation, refinement, and concrete coupling invariant relations (resp. $\mathscr{B}\mathsf{s}, \mathscr{R}\mathsf{s}, \mathscr{I}\mathsf{s}$) must all be for the same number of threads.

- Then, for all threads i in the system:
  - The relations $\mathscr{B}, \mathscr{R}, \mathscr{I}$ for thread i must meet the requirements for "compositional whole-system refinement" (Definition 2.22).
  - The refinement relation $\mathscr{R}$ for thread i must hold initially, i.e. cover its initial thread-private and mode states at concrete and abstract level (resp. cms_C, cms_A), for all initial memories that satisfy the concrete $INIT_C$ requirement.
  - The abstract bisimulation relation $\mathscr{B}$ for thread i must hold initially, i.e. must relate its initial thread-private and mode state to itself, for all pairs of memories that are low-equal modulo that mode state, and that both satisfy the abstract $INIT_A$ requirement.

Given all these assumptions, Theorem 2.23 yields a whole-system noninterference property $\mathsf{sys\text{-}secure}_{INIT_C}$ for the resulting concurrent program (with the list of initial thread-private and mode states cms_C) that assumes that the initial memory satisfies $INIT_C$.

3 Source language: `While` with mutex locks

In this section, we give a focused presentation of our compiler's source language, centered on its properties that enable *the composition* of per-thread proofs of CVDNI-preserving refinement to the compiler's target `RISC` language. Our Isabelle/HOL supplement provides full formalisations of its semantics, and of instances of all per-thread proof techniques for proving CVDNI itself (developed for `While` by Sison (2020); Murray *et al.* (2016b,c)).

`While` with mutex locks (hereafter `While`) is a generic imperative language with support for conditional looping, consisting of the commands *cmd* over arithmetic expressions *exp*:

$$\begin{aligned} exp &::= n \mid v \mid exp \oplus exp \\ cmd &::= \textbf{skip} \mid cmd\,;cmd \mid \textbf{if } exp \textbf{ then } cmd \textbf{ else } cmd \textbf{ fi} \mid \\ &\quad\ \textbf{while } exp \textbf{ do } cmd \textbf{ od} \mid v := exp \mid \textbf{stop} \mid \\ &\quad\ \textbf{lock}(k) \mid \textbf{unlock}(k) \end{aligned}$$

The language is parameterised over shared program-variable identifiers $v :: Var$, shared lock-variable identifiers $k :: Lock$, constant values $n :: Val$, and binary arithmetic operators $\oplus :: Val \Rightarrow Val \Rightarrow Val$ that each have a big-step evaluation semantics; these induce a big-step evaluation semantics for *exp* as a whole. The commands *cmd* then have a small-step operational semantics, wherein **skip** and variable assignment $v := exp$ execute in one step to **stop** (which itself does not step to anything); conditional branch **if** steps to the appropriate *cmd* depending on whether its expression evaluates to zero; and conditional loop **while** steps to an **if**-conditional between either (1) the loop body sequenced with a repetition of the **while** command, or (2) **stop**. Finally the sequential command $c_1\,;c_2$ executes to c_2 when c_1 executes to **stop**, and to $c_1'\,;c_2$ (c_1' being c_1's destination) otherwise. Of these aforementioned commands, only variable assignments can modify the shared memory (program-variables only), and none can directly modify the mode state or lock-variables.

We will give special focus to the addition to the `While` language of the mutex synchronisation primitives $\textbf{lock}(k)$ and $\textbf{unlock}(k)$, which are the sole means of modifying lock variables and mode state. These replace both the ad-hoc mode annotations and the $\textbf{await}(v)$

synchronisation primitive that were previously offered for `While` by Murray *et al.* (2016b). After briefly noting here how `While` instantiates the underlying theory from Section 2, we will present these new primitives' operational semantics, which depends on the program developer supplying details of the locking discipline as a parameter (Section 3.1) subject to some restrictions (Section 3.2). We will then prove that global-modes-compatibility (Definition 2.12) is invariant for systems of `While` programs running concurrently (Section 3.3), subject to some initial conditions (Section 3.4). Discharging this once-off noncompositional proof obligation is crucial in enabling both composition of per-thread noninterference properties (using Theorem 2.8), and compositional whole-system secure refinement of noninterference down to `RISC` by our compiler (using Theorem 2.23).

`While` instantiates the concurrent value-dependent noninterference theory described in Section 2.2. This instantiation assumes that the underlying concurrent execution model (e.g. operating system, scheduler) for the `While` language prevents threads from seeing each others' current program location. Thus the `While` program command $c :: cmd$ being executed (understood as the current program location) is modelled as the thread-private state of the local configuration triple: $\langle c, mds, mem\rangle_{\mathsf{w}}$. (The subscript $_{\mathsf{w}}$ distinguishes `While` program triples from `RISC` ones, which are subscripted $_{\mathsf{r}}$.)

To ease formalisation of $\mathbf{lock}(k)$ and $\mathbf{unlock}(k)$, we instantiate the shared $mem :: Mem$ type as a total mapping from a sum type to values *Val*. This sum type, with constructors $\mathsf{Lock}, \mathsf{Var}$, distinguishes lock-variable identifiers $k :: Lock$ (which can only be read or written by the lock primitives) from program-variable identifiers $v :: Var$ (which can be read or written by the rest of the commands). In Isabelle/HOL's datatype notation, this is:

$$Mem \triangleq (\mathsf{Lock}\ Lock \mid \mathsf{Var}\ Var) \Rightarrow Val$$

For readability, we will elide this distinction between *Lock* and *Var*—or applications of their constructors Lock and Var—from the presentation whenever clear from the context.

3.1 Locking discipline and its semantics

The program developer provides the details of the program's locking discipline in the form of a *lock interpretation* parameter $lock\text{-}interp :: Lock \Rightarrow (Var\ set \times Var\ set)$, which gives for each lock the two non-overlapping sets of program-variables over which acquiring the lock grants exclusive permission to write, (resp.) read and write. For readability, this presentation will elide *lock-interp* from the arguments of definitions, and use the notation $varsNoW, varsNoRW :: Lock \Rightarrow Var\ set$ to refer to its fst and snd projection.

Alongside encoding the mutex primitives' usual effect on control flow—most crucially, $\mathbf{lock}(k)$ should refuse to proceed meaningfully if the lock k is already held—we will now specify for them an evaluation semantics that furthermore encodes the permissions implied by the locking discipline, as assumptions and guarantees expressed in the mode state. This semantics assumes that, initially, no locks are held, and all threads are making guarantees not to access the variables they govern (conditions we will define formally in Section 3.4).

The following two helpers specify how acquiring (resp. releasing) a lock affects the mode state under a given lock interpretation *lock-interp*. When a thread acquires a lock it gains more assumptions, and makes fewer guarantees about the region of memory concerned:

Definition 3.1 (Impact on mode state *mds* of acquiring lock k)**.**

$$\begin{aligned} mds \oplus k \triangleq \lambda\, m.\ \mathsf{case}\ m\ \mathsf{of}\ & \mathbf{GuarNoW} \Rightarrow mds\ \mathbf{GuarNoW} - varsNoW\ k \\ \mid\ & \mathbf{AsmNoW} \Rightarrow mds\ \mathbf{AsmNoW} \cup varsNoW\ k \\ \mid\ & \mathbf{GuarNoRW} \Rightarrow mds\ \mathbf{GuarNoRW} - varsNoRW\ k \\ \mid\ & \mathbf{AsmNoRW} \Rightarrow mds\ \mathbf{AsmNoRW} \cup varsNoRW\ k \end{aligned}$$

The converse occurs when releasing a lock: the thread drops the assumptions it was making about that region of memory, and once again makes guarantees not to access it.

Definition 3.2 (Impact on mode state *mds* of releasing lock k)**.**

$$\begin{aligned} mds \ominus k \triangleq \lambda\, m.\ \mathsf{case}\ m\ \mathsf{of}\ & \mathbf{GuarNoW} \Rightarrow mds\ \mathbf{GuarNoW} \cup varsNoW\ k \\ \mid\ & \mathbf{AsmNoW} \Rightarrow mds\ \mathbf{AsmNoW} - varsNoW\ k \\ \mid\ & \mathbf{GuarNoRW} \Rightarrow mds\ \mathbf{GuarNoRW} \cup varsNoRW\ k \\ \mid\ & \mathbf{AsmNoRW} \Rightarrow mds\ \mathbf{AsmNoRW} - varsNoRW\ k \end{aligned}$$

The operational semantics for $\mathbf{lock}(k)$ is then given by two rules: LOCKACQ when lock k is available, and LOCKSPIN when it is already held. For these, we use predicate $\mathsf{ev}_{Lock} :: Val \Rightarrow bool$ with designated constants $\mathsf{True}_{Lock}, \mathsf{False}_{Lock} :: Val$ to indicate that the lock is, resp. is not held—i.e. $\mathsf{ev}_{Lock}(\mathsf{True}_{Lock}) = \mathsf{True}$, and $\mathsf{ev}_{Lock}(\mathsf{False}_{Lock}) = \mathsf{False}$.[3]

Apart from impacting the mode state as already specified (by Definition 3.1), attempting to acquire an available lock will succeed in the usual manner, setting the lock-variable to the designated constant (True_{Lock}) to prevent subsequent lock acquisition attempts:

$$\frac{\neg \mathsf{ev}_{Lock}\ (mem\ (\mathsf{Lock}\ k)) \qquad mem' = mem[\mathsf{Lock}\ k \mapsto \mathsf{True}_{Lock}] \qquad mds' = mds \oplus k}{\langle \mathbf{lock}(k), mds, mem \rangle_{\mathsf{w}} \rightsquigarrow_{\mathsf{w}} \langle \mathbf{stop}, mds', mem' \rangle_{\mathsf{w}}}\ \textsc{LockAcq}$$

Attempting to acquire an already-held lock results in a stuttering evaluation step:

$$\frac{\mathsf{ev}_{Lock}\ (mem\ (\mathsf{Lock}\ k))}{\langle \mathbf{lock}(k), mds, mem \rangle_{\mathsf{w}} \rightsquigarrow_{\mathsf{w}} \langle \mathbf{lock}(k), mds, mem \rangle_{\mathsf{w}}}\ \textsc{LockSpin}$$

Then, the operational semantics for $\mathbf{unlock}(k)$ is given by two rules, of which only one, LOCKREL, will ever be used by programs that follow locking discipline. This rule requires that the mode state *mds* is consistent with the present thread having previously acquired the lock k: In short, it should have all the assumptions, but none of the guarantees, associated with the variables governed by the lock. To specify this, we define the following helper:

Definition 3.3 (Mode state is consistent with holding a lock k)**.**

$$\begin{aligned} & \mathsf{lock\text{-}held\text{-}mds\text{-}correct}\ mds\ k \triangleq \\ & \quad \forall x.\ (x \in varsNoW\ k \longrightarrow x \notin mds\ \mathbf{GuarNoW} \wedge x \in mds\ \mathbf{AsmNoW}) \wedge \\ & \qquad (x \in varsNoRW\ k \longrightarrow x \notin mds\ \mathbf{GuarNoRW} \wedge x \in mds\ \mathbf{AsmNoRW}) \end{aligned}$$

[3] All three of $\mathsf{ev}_{Lock}, \mathsf{True}_{Lock}, \mathsf{False}_{Lock}$ are parameters that are set by the user of the theory, with the proviso that their choice of parameters satisfy that $\mathsf{ev}_{Lock}(\mathsf{True}_{Lock})$ and $\neg \mathsf{ev}_{Lock}(\mathsf{False}_{Lock})$ hold as required.

With that condition satisfied, the LOCKREL rule specifies that an **unlock**(k) will proceed successfully, to enact lock release on the memory and mode state as expected:

$$\frac{\mathsf{lock\text{-}held\text{-}mds\text{-}correct}\ \mathit{mds}\ k \qquad \mathit{mem}' = \mathit{mem}[\mathsf{Lock}\ k \mapsto \mathsf{False}_{\mathit{Lock}}] \qquad \mathit{mds}' = \mathit{mds} \ominus k}{\langle \mathbf{unlock}(k), \mathit{mds}, \mathit{mem}\rangle_{\mathsf{w}} \rightsquigarrow_{\mathsf{w}} \langle \mathbf{stop}, \mathit{mds}', \mathit{mem}'\rangle_{\mathsf{w}}}\ \textsc{LockRel}$$

To ensure that the `While` evaluation semantics is defined for all possible configurations, the LOCKINVALID rule defines a stuttering evaluation step for attempts to **unlock**(k) that violate the locking discipline due to not having previously acquired the lock k:

$$\frac{\neg\ \mathsf{lock\text{-}held\text{-}mds\text{-}correct}\ \mathit{mds}\ k}{\langle \mathbf{unlock}(k), \mathit{mds}, \mathit{mem}\rangle_{\mathsf{w}} \rightsquigarrow_{\mathsf{w}} \langle \mathbf{unlock}(k), \mathit{mds}, \mathit{mem}\rangle_{\mathsf{w}}}\ \textsc{LockInvalid}$$

As mode state is nominally a form of ghost state, having the operational semantics appear to depend on it in this manner is rather unusual. To remove the semantics' reliance on ghost state, the program developer must use a check for local-mode-compliance (Definition 2.11) that only ever admits programs that satisfy the lock-held-mds-correct check whenever attempting to **unlock**(k). For such programs, the operational semantics is equivalent to one that (1) omits the lock-held-mds-correct check from the LOCKREL rule, and (2) omits the LOCKINVALID rule from the `While`-language semantics entirely. An example of such a check is included in our Isabelle/HOL supplement.

3.2 Restrictions on locking disciplines

Here we lay out some cleanliness conditions on locking disciplines, giving particular focus to those relevant to our locking semantics (Section 3.1), and to our verification efforts for `While`'s global compositionality property (Section 3.3) and our compiler (Section 5).

Of these, only one is a hard consequence of the underlying CVDNI theory we presented in Section 2: The per-thread CVDNI property com-secure (Definition 2.7) effectively compels us to enforce that secrets are never allowed to leak into the locking state. Otherwise, mode state would become tainted upon any attempt to acquire a lock whose status is secret, which would violate com-secure's requirement that modes-equality must be maintained at all times (note the $=_{\mathsf{mds}}$ enforced by strong-low-bisim-mm, Definition 2.4). To ensure that com-secure will always treat the locking state as an untrusted sink, we impose the following requirement on the $\mathscr{L}$ parameter supplied by the program developer:

Proposition 3.4 ($\mathscr{L}$ must permanently assign Low classification to all lock-variables k)**.**

$$\forall k\ \mathit{mem}.\ \mathscr{L}\ \mathit{mem}\ (\mathsf{Lock}\ k) = \mathsf{Low}$$

The remaining restrictions are consequences of various simplifications of convenience.

First, note that the type signature of the *lock-interp* parameter (given in Section 3.1) only allows locks to govern program variables, not other locks. We justify this simplification with the fact that if some lock k governed lock k', then k would already have to be held whenever acquiring k'—otherwise, the change to k' would violate a no-write assumption implied by the locking discipline. This, however, would make k' entirely redundant with k.

Second, the lock acquisition and release semantics we gave in Section 3.1 is rather simplified, in that releasing a lock will drop the assumptions of all its variables from the mode state, even if another lock for that variable is still held! Thus, we signal that it only works for disciplines wherein no more than one lock governs each program variable, by asserting:

Proposition 3.5 (No variable can be managed by more than one lock)**.**

$$\forall v\ k.\ v \in \mathit{varsNoW}\ k \cup \mathit{varsNoRW}\ k \longrightarrow (\forall k'.\ v \in \mathit{varsNoW}\ k' \cup \mathit{varsNoRW}\ k' \longrightarrow k' = k)$$

We believe that it would be feasible to relax Proposition 3.5, by generalising `While`'s locking semantics to allow disciplines wherein multiple locks must be held to access a given variable. To satisfy CVDNI-preserving refinement (particularly Definition 2.13), a compiler would need to preserve the lock memory operations that implement the more sophisticated bookkeeping needed, as ours does for the current, much simpler locking semantics.

Next, we assume that the program developer has not specified any "vacuous" locks (i.e. ones that govern no variables), and that all locks grant at most one of **AsmNoW** or **AsmNoRW** (i.e. not both) on any given variable. These two assumptions allow us to exclude various pathological cases from our reasoning in Section 3.3 and Section 5, respectively:

Proposition 3.6 (Every lock governs access to some variable)**.**

$$\forall k.\ \mathit{varsNoW}\ k \cup \mathit{varsNoRW}\ k \neq \emptyset$$

Proposition 3.7 (The lock interpretation sets for any given lock k do not overlap)**.**

$$\forall k.\ \mathit{varsNoW}\ k \cap \mathit{varsNoRW}\ k = \emptyset$$

The final two restrictions simplify the possible interactions between locks and control variables: We disallow locks from being control variables, and require variables to be governed by the same lock as their control variables. In particular, they will help us establish (in Section 5.4) that the compiler produces programs that satisfy local-mode-compliance.

First, recall we mentioned that, as part of local-mode-compliance (Definition 2.11), the doesnt-read-(or-modify) assertions entail that any guarantees not to access some variable v will effectively apply also to all of v's control variables. Disallowing lock-variables from being control variables thus ensures that **lock**(k) and **unlock**(k), because they only access lock-variable k, cannot violate doesnt-read-(or-modify) for any program-variables:

Proposition 3.8 (Lock-variables k cannot be control variables)**.**

$$\forall k.\ (\mathsf{Lock}\ k) \notin \mathscr{C}$$

Finally, requiring variables to be governed by the same lock as their control variables effectively ensures they are always locked simultaneously. Apart from making it easier for programs to satisfy local-mode-compliance, this also naturally prevents leaks caused by other threads changing a variable's classification to Low when it still contains High data:

Proposition 3.9 (Variables are always governed by the same lock as their control variables)**.**

$$\forall c\ v\ k.\ \mathsf{Var}\ c \in \mathscr{C}\mathsf{vars}\ (\mathsf{Var}\ v) \longrightarrow (c \in \mathit{varsNoW}\ k = v \in \mathit{varsNoW}\ k) \wedge (c \in \mathit{varsNoRW}\ k = v \in \mathit{varsNoRW}\ k)$$

3.3 Proof of global modes compatibility as an invariant

This section will present proof that global-modes-compatibility (Definition 2.12) holds as an invariant for concurrent `While` programs (Section 3.3) when initialised to have no locks held (Section 3.4). Consequently, it is sufficient for a developer to use a local compliance check (Sison, 2020) to obtain the sound-mode-use condition (Definition 2.10) needed for per-thread security proofs to be compositional via Theorem 2.8.

Recall from Section 2.2 that this compatibility requirement formalises that for all reachable global configurations of a concurrent program, any assumptions made by any of the threads must be met by corresponding guarantees made by all of the other threads:

Definition 2.12 (Global modes compatibility)**.**

$$\textsf{global-modes-compatibility}\ gc \triangleq \forall mdss \in \textsf{reachable-mds-lists}\ gc.\ \textsf{compatible-modes}\ mdss$$

where

$$\textsf{reachable-mds-lists}\ gc \triangleq \{mdss \mid \exists cms'\ mem'\ sched.\ gc \dashrightarrow_{sched} (cms', mem') \wedge \textsf{map mds}\ cms' = mdss\}$$

$$\textsf{compatible-modes}\ mdss \triangleq \forall i\ x.\ i < \textsf{length}\ mdss \longrightarrow (x \in mdss[i]\ \textbf{AsmNoRW} \longrightarrow (\forall j < \textsf{length}\ mdss.\ j \neq i \longrightarrow x \in mdss[j]\ \textbf{GuarNoRW})) \wedge (x \in mdss[i]\ \textbf{AsmNoW} \longrightarrow (\forall j < \textsf{length}\ mdss.\ j \neq i \longrightarrow x \in mdss[j]\ \textbf{GuarNoW}))$$

The approach to establish global-modes-compatibility here will be to define three *mode management requirements* that taken together imply compatible-modes, and to prove them invariant for concurrent `While` programs when initialised such that they hold to begin with.

The first of these pertains to variables whose access is governed by some lock, according to the locking discipline. To define it, we need, alongside lock-held-mds-correct (Definition 3.3) from Section 3.1, a predicate that specifies the correct mode state for *not* holding a lock k: It should make all of the guarantees, and have none of the assumptions associated with the variables governed by k.[4] Stated formally:

Definition 3.10 (Mode state is consistent with *not* holding a lock k)**.**

$$\textsf{lock-not-held-mds-correct}\ mds\ k \triangleq \forall x.\ (x \in varsNoW\ k \longrightarrow x \in mds\ \textbf{GuarNoW} \wedge x \notin mds\ \textbf{AsmNoW}) \wedge (x \in varsNoRW\ k \longrightarrow x \in mds\ \textbf{GuarNoRW} \wedge x \notin mds\ \textbf{AsmNoRW})$$

Note that our simplifying exclusion of "vacuous" locks (Proposition 3.6) ensures we never have to deal with a case where lock-held-mds-correct *mds k* and lock-not-held-mds-correct *mds k* hold simultaneously.

The requirement on global configurations regarding these lock-managed variables is then as follows: If and only if a given lock is held by anybody, then exactly one thread has a mode state consistent with holding it; furthermore, all other threads will have a mode state consistent with not holding it. Formally, with $\textsf{mdss}\ gc \triangleq \textsf{map mds}\ (\textsf{cms}\ gc)$:

[4] Note that this not merely the negation of lock-held-mds-correct *mds k* (Definition 3.3)!

Definition 3.11 (Lock-managed variable modes are compatible with memory)**.**

$$\begin{array}{l}\mathsf{lock\text{-}managed\text{-}modes\text{-}mem\text{-}compatible}\ gc \triangleq \\ \quad \forall k.\ \mathsf{if}\ (\mathsf{ev}_{Lock}\ ((\mathsf{mem}\ gc)\ k))\ \mathsf{then} \\ \quad\quad \exists! i.\ i < \mathsf{length}\ (\mathsf{cms}\ gc) \wedge \\ \quad\quad\quad \mathsf{lock\text{-}held\text{-}mds\text{-}correct}\ (\mathsf{mdss}\ gc)[i]\ k \wedge \\ \quad\quad\quad (\forall j < \mathsf{length}\ (\mathsf{cms}\ gc).\ i \neq j \longrightarrow \\ \quad\quad\quad\quad \mathsf{lock\text{-}not\text{-}held\text{-}mds\text{-}correct}\ (\mathsf{mdss}\ gc)[j]\ k) \\ \quad\quad \mathsf{else}\ \forall i < \mathsf{length}\ (\mathsf{cms}\ gc). \\ \quad\quad\quad \mathsf{lock\text{-}not\text{-}held\text{-}mds\text{-}correct}\ (\mathsf{mdss}\ gc)[i]\ k\end{array}$$

The second requirement pertains to variables whose access is entirely ungoverned by any locks in the locking discipline. For these we specify a more direct check that if any thread in the global configuration has an assumption about access to any of these variables, then all other threads must be providing the corresponding guarantee to that assumption:

Definition 3.12 (Unmanaged variable modes are compatible)**.**

$$\begin{array}{l}\mathsf{unmanaged\text{-}var\text{-}modes\text{-}compatible}\ gc \triangleq \forall i\ x.\ i < \mathsf{length}\ (\mathsf{mdss}\ gc) \longrightarrow \\ \quad (x \notin \bigcup_{k::Lock} varsNoRW\ k \longrightarrow \\ \quad\quad (x \in (\mathsf{mdss}\ gc)[i]\ \mathbf{AsmNoRW} \longrightarrow \\ \quad\quad\quad (\forall j < \mathsf{length}\ (\mathsf{mdss}\ gc).\ j \neq i \longrightarrow x \in (\mathsf{mdss}\ gc)[j]\ \mathbf{GuarNoRW}))) \wedge \\ \quad (x \notin \bigcup_{k::Lock} varsNoW\ k \longrightarrow \\ \quad\quad (x \in (\mathsf{mdss}\ gc)[i]\ \mathbf{AsmNoW} \longrightarrow \\ \quad\quad\quad (\forall j < \mathsf{length}\ (\mathsf{mdss}\ gc).\ j \neq i \longrightarrow x \in (\mathsf{mdss}\ gc)[j]\ \mathbf{GuarNoW})))\end{array}$$

Also proved invariant is a third, minor property that enforces globally that no assumptions or guarantees are ever recorded regarding access to lock-variables:

Definition 3.13 (No assumptions and guarantees on lock variables)**.**

$$\begin{array}{l}\mathsf{no\text{-}lock\text{-}mds}\ mds \triangleq \forall l\ m.\ \mathsf{Lock}\ l \notin mds\ m \\ \mathsf{no\text{-}lock\text{-}mds\text{-}gc}\ gc \triangleq \forall mds \in \mathsf{set}\ (\mathsf{mdss}\ gc).\ \mathsf{no\text{-}lock\text{-}mds}\ mds\end{array}$$

This follows trivially from (1) our simplification (discussed in Section 3.2) only to allow locks to protect access to program variables and not other locks, and (2) the resulting fact that no `While` primitives ever touch any mode state pertaining to lock variables. Thus, further details on this third management requirement will be elided.

We then have straightforwardly from their definitions that together, these three mode management requirements imply compatible modes for a given global configuration:

Lemma 3.14 (Management requirements ensure compatibility)**.**

$$\frac{\mathsf{lock\text{-}managed\text{-}modes\text{-}mem\text{-}compatible}\ gc \qquad \mathsf{unmanaged\text{-}var\text{-}modes\text{-}compatible}\ gc \qquad \mathsf{no\text{-}lock\text{-}mds\text{-}gc}\ gc}{\mathsf{compatible\text{-}modes}\ (\mathsf{mdss}\ gc)}$$

Proofs of invariance then proceed by induction over the single-step evaluation semantics of an arbitrary thread taking a step to progress the system to a new global configuration.

For the first management requirement (Definition 3.11):

Lemma 3.15 (Single-step preservation of lock-managed-modes-mem-compatible)**.**

$$\frac{\begin{array}{c}\textsf{lock-managed-modes-mem-compatible}\ (cms, mem) \\ \langle c_i, mds_i, mem\rangle_{\mathsf{w}} \rightsquigarrow_{\mathsf{w}} \langle c_i', mds_i', mem'\rangle_{\mathsf{w}} \qquad i < \textsf{length}\ cms \\ cms' = cms[i := (c_i', mds_i')] \qquad cms[i] = (c_i, mds_i)\end{array}}{\textsf{lock-managed-modes-mem-compatible}\ (cms', mem')}$$

Proof. By induction over the single-threaded evaluation semantics of the program at index i that is taking a step.

lock(k) preserves the property because it only allows a thread to set lock k's memory if it is not already set – it would then become the single unique thread whose mode state is consistent with holding k. Otherwise, the mode states and memory remain unchanged.

Similarly, **unlock**(k) preserves the property because its only possible change is to unset lock k's memory, and return the unique thread holding lock k to a mode state consistent with not holding k.

The other `While` commands preserve the property because they do not touch the mode state nor any lock-variables. □

For the second management requirement (Definition 3.12):

Lemma 3.16 (Single-step preservation of unmanaged-var-modes-compatible)**.**

$$\frac{\begin{array}{c}\textsf{unmanaged-var-modes-compatible}\ (cms, mem) \\ \langle c_i, mds_i, mem\rangle_{\mathsf{w}} \rightsquigarrow_{\mathsf{w}} \langle c_i', mds_i', mem'\rangle_{\mathsf{w}} \qquad i < \textsf{length}\ cms \\ cms' = cms[i := (c_i', mds_i')] \qquad cms[i] = (c_i, mds_i)\end{array}}{\textsf{unmanaged-var-modes-compatible}\ (cms', mem')}$$

Proof. Again, by induction over the single-threaded evaluation semantics of the program at index i that is taking a step.

We prove and use lemmas that **lock**(k) and **unlock**(k) do not touch any mode state pertaining to variables that are unmanaged by any locks, and that the remaining `While` commands do not touch the mode state at all. Therefore evaluation steps cannot possibly have any effect on the compatibility of modes on these variables. □

These single-step evaluation results lift easily to invariance results over the global multi-step evaluation semantics quantified over arbitrary schedules. These invariance results, with the fact that the management requirements ensure compatibility (Lemma 3.14), yield in a straightforward manner the desired global compatibility invariance theorem:

Theorem 3.17 (Mode management requirements ensure global compatibility)**.**

$$\frac{\textsf{lock-managed-modes-mem-compatible}\ gc \qquad \textsf{unmanaged-var-modes-compatible}\ gc \qquad \textsf{no-lock-mds-gc}\ gc}{\textsf{global-modes-compatibility}\ gc}$$

3.4 Initial conditions ensuring global modes compatibility

We now define conditions on memory and mode state consistent with no locks being held, and show that initialising a system under these conditions is enough to satisfy the global compatibility part (Definition 2.12) of the sound-mode-use side condition (Definition 2.10) of the compositionality theorem for our security property (Theorem 2.8).

We define the following predicate for initial memory:

Definition 3.18 (A requirement for initial memory that no locks are held)**.**

$$\mathsf{no\text{-}locks\text{-}held}\ \mathit{mem} \triangleq \forall k.\ \neg \mathsf{ev}_{\mathit{Lock}}\ (\mathit{mem}\ k)$$

We then define an initial mode state $\mathsf{mds}_0 :: \mathit{Mode} \Rightarrow \mathit{Var\ set}$ that provides all guarantees demanded by the lock interpretation parameters $\mathit{varsNoW}, \mathit{varsNoRW}$ (described in Section 3.1) for all lock variables in the system, and makes no assumptions:

Definition 3.19 (Initial mode state mds_0)**.**

$$\begin{aligned}\mathsf{mds}_0 \triangleq\ \lambda\ m.\ \mathsf{case}\ m\ \mathsf{of}\ &\mathbf{GuarNoW} \Rightarrow \bigcup_{k::\mathit{Lock}} \mathit{varsNoW}\ k\\ &\mid \mathbf{GuarNoRW} \Rightarrow \bigcup_{k::\mathit{Lock}} \mathit{varsNoRW}\ k\\ &\mid \mathbf{AsmNoW} \Rightarrow \emptyset\\ &\mid \mathbf{AsmNoRW} \Rightarrow \emptyset\end{aligned}$$

We are then able to show that these conditions are enough to satisfy the requirements we just showed (in Section 3.3) ensure global modes compatibility for `While`:

Lemma 3.20 (Initialising with $\mathsf{no\text{-}locks\text{-}held}, \mathsf{mds}_0$ ensures global modes compatibility)**.**

$$\frac{\mathsf{no\text{-}locks\text{-}held}\ \mathit{mem} \qquad \forall (c, \mathit{mds}) \in \mathsf{set}\ \mathit{cms}.\ \mathit{mds} = \mathsf{mds}_0}{\mathsf{global\text{-}modes\text{-}compatibility}\ (\mathit{cms}, \mathit{mem})}$$

Proof. Theorem 3.17 obliges us to show that the mode management conditions (Definitions 3.11, 3.12, and 3.13) hold. This follows straightforwardly from all the relevant definitions. □

4 Target language: `RISC` with mutex locks

Here we introduce `RISC` with mutex locks (hereafter `RISC`), the target of our compiler. This is a generic RISC-style assembly language based on the `RISC` architecture targeted by the compilation scheme of Tedesco *et al.* (2016). A `RISC` program text is a list of `RISC` instructions I, each optionally associated with a label:

$$\begin{aligned}I &::= [l :]B\\ B &::= \mathbf{Load}\ r\ v \mid \mathbf{Store}\ v\ r \mid \mathbf{Jmp}\ l \mid \mathbf{Jz}\ l\ r \mid \mathbf{Nop}\\ &\quad\ \mathbf{MoveK}\ r\ n \mid \mathbf{MoveR}\ r\ r \mid \mathbf{Op}\ \oplus\ r\ r\\ &\quad\ \mathbf{LockAcq}\ k \mid \mathbf{LockRel}\ k\end{aligned}$$

Here we fix the types of the constant values $n :: Val$, binary arithmetic operators $\oplus :: Val \Rightarrow Val \Rightarrow Val$, shared program variables $v :: Var$, and shared lock variables $k :: Lock$ to be the same as those for the source `While` language being compiled. Thus, the only new types here compared to Section 3 are for the register identifiers $r :: Reg$, and labels $l :: Lab$.

`RISC` has a small-step operational semantics that is largely unchanged from Tedesco *et al.* (2016), in that each step updates a distinguished *program counter* register, which captures the current thread's program location as an index into its `RISC` program text. The instructions **MoveK**, **MoveR**, **Load**, and **Store**, for moving values to and between the registers and shared memory, and the "no-op" instruction **Nop**, all increment the program counter; the "jump if zero" instruction **Jz** l r updates it to the index of the instruction at l if r contains zero (else increments it); the unconditional **Jmp** l does so unconditionally.

Modifying this instruction set from Tedesco *et al.* (2016), we then customise the **Op** instruction, and add **LockAcq** k and **LockRel** k instructions, to have semantics mirroring those of $\oplus$, **lock**(k), and **unlock**(k) from `While` respectively. Whereas the `RISC` equivalents for the LOCKACQ and LOCKREL evaluation rules (described by Section 3.1 for the `While` language) increment the program counter, those for LOCKSPIN and LOCKINVALID leave it unchanged. There is no `RISC` evaluation rule that changes the program text.

Although it has only direct-addressing **Load** and **Store** instructions, our `RISC` target language is adequate for implementing all features of `While` present in Section 3, with the big-step semantics of *exp* replaced by small-step operations on registers. We relegate `RISC`'s full formal semantics to this paper's supplement Isabelle/HOL material.

Our defining **LockAcq** k and **LockRel** k to have the same operational semantics on shared memory and mode state as `While`'s **lock**(k) and **unlock**(k) has two consequences:

- Our compiler will expect the program developer to supply the details of the locking discipline for the `While` program being compiled, so as to be able to ensure that the `RISC` program it produces follows the same discipline.
- We then have that global compatibility is invariant for `RISC` execution, by a near-identical argument to the one we presented in Section 3.3, when initialised with the conditions we presented in Section 3.4. This presents one option for obtaining `RISC`-level composition of per-thread noninterference properties; however, invoking it directly will not be necessary when using the compositional whole-system secure refinement method of Section 2.5. (The alternative options and their application will be demonstrated further, respectively in Section 5.4, Section 6.3.)

As for `While` in Section 3, we instantiate here for `RISC` the CVDNI theory of Murray *et al.* (2016b) as recalled in Section 2.2, assuming that the underlying concurrency model (e.g. OS, scheduler etc.) prevents one thread from reading the program text of another. For `RISC`, we furthermore assume that the context switching mechanism ensures effectively that no thread can read or interfere with the contents of the registers (including the program counter) when active for another thread. Based on these assumptions, we model all three of the program counter register's value $pc :: nat$, `RISC` program text $P :: I\ list$, and register bank $regs :: Reg \Rightarrow Val$, as thread-private state in the local configuration triple: $\langle((pc, P), regs), mds, mem\rangle_{\mathtt{r}}$. (We use the subscript $_{\mathtt{r}}$ to distinguish `RISC` triples.)

5 Verified secure compiler for mixed-sensitivity concurrent `While` programs

This section presents the COVERN wr-compiler: the first compiler proved to preserve proofs of noninterference for mixed-sensitivity concurrent programs. By using assume–guarantee modes (Mantel *et al.*, 2011) and the decomposition principle of Section 2.4 to prove it introduces (resp.) no race conditions or timing leaks, we demonstrate the *applicability to compiler verification* of the CVDNI-preserving refinement notion of Section 2.3 originally posed by Murray *et al.* (2016b). Here the decomposition principle (Figure 4) is crucial because, in separating the concern of preventing new timing leaks, it avoids directly having to prove the cube-shaped refinement diagram (Figure 3) arising from its need to preserve a 2-safety hyperproperty (Terauchi & Aiken, 2005; Clarkson & Schneider, 2010).

To preserve security for mixed-sensitivity concurrent programs, CVDNI-preserving refinement demands small-step preservation of *the contents of all shared memory locations* including those that control value-dependent classifications and implement locks. As it is unusual for verified compilers to make such promises, we show that a valid approach is to take advantage of CVDNI's assume–guarantee framework to:

1. *test and preserve any absence of race conditions* implied (via the framework) by mutex lock-based synchronisation of access to such locations, and then

2. *use this absence of race conditions* to establish the small-step preservation of their contents demanded for security-preserving refinement.

In doing so, we prove that some optimisations the wr-compiler performs with its knowledge of the locking discipline—it avoids unnecessary **Load**s and recalculation of common subexpressions over shared memory when locked—are safe to allow without violating CVDNI.

In preserving CVDNI, the wr-compiler preserves security proofs that are produced by the program verification techniques of Sison (2020) for `While` with mutex locks, which in turn were adapted from Murray *et al.* (2016b,c). We will present such an application of our compiler, to a case study program verified using these techniques, in Section 6.

Section 5.1 will focus on the wr-compiler's particular adaptations to CVDNI (beyond the fault-resilient noninterference targeted by the original compilation scheme of Tedesco *et al.* (2016)), in the form of static checks and invariants that (resp.) test for and maintain the absence of race conditions on lock-protected shared variables. Section 5.2 formalises a ban, preserved by the wr-compiler, on secret-dependent control flow. Section 5.3 then presents formal proof (structured by our decomposition principle of Section 2.4) that the wr-compiler implements CVDNI-preserving refinement. Section 5.4 ultimately presents proofs of overall security preservation results useful to users of the wr-compiler: Namely, it can be used either to preserve security down to `RISC` for an entire concurrent `While`-language program, or to preserve the per-thread security for threads that will be run alongside others written directly in the `RISC`-language.

5.1 Preserving race-free expression evaluation

Recall from Section 2.3 that CVDNI-preserving refinement (Murray *et al.*, 2016b) demands that all shared memory contents be preserved, between each target- and source-language

configuration that it relates. This is security critical for mixed-sensitivity concurrent programs, as it ensures that any future influence of those contents on value-dependent classifications (via control variables) or readability by other threads (in the case of the `While` and `RISC` languages, via lock variables) is preserved.

The wr-compiler's approach to preserving the contents of shared memory is to ensure:

1. That values calculated by expressions are *preserved* by compilation—that is, they have the same value when written back to shared memory (or conditionally branched on) by the `RISC` program, as they did in the original `While` program; and

2. That expression evaluation is *race-free*—that is, free of any race conditions with other threads that would render the calculated expression inaccurate.

   To this end, the wr-compiler requires of the original `While` program that whenever each thread attempts to evaluate an expression, it must hold locks ensuring the stability of *all* variables referenced by the expression.

Thus, its knowledge and enforcement of the locking discipline is crucial, not only to show that its optimisations preserve CVDNI, but that *any meaningful operation* over shared memory preserves it. It therefore tests for *and rejects* programs that exhibit potential race conditions due to their failure to follow locking discipline—these result in a failed compilation.

The wr-compiler tracks two kinds of information to achieve these outcomes: the contents of registers as expressions over shared variables, and assumptions on access to variables by other threads. The structures the wr-compiler uses to do this are, respectively:

- A *register record* $\Phi :: \mathit{RegRec} \triangleq \mathit{Reg} \rightharpoonup \mathit{exp}$. This draws inspiration from that used by the compilation scheme of Tedesco *et al.* (2016) (originally of type $\mathit{Reg} \rightharpoonup \mathit{Var}$) to avoid generating unnecessary **Load** instructions to registers that already contain a variable; in addition, here we extend it to track entire expressions on shared variables.

- An *assumption record* $\mathscr{S} :: \mathit{AsmRec} \triangleq (\mathit{Var\ set} \times \mathit{Var\ set})$ that tracks which variables at a given point in the source `While` program are "stable" due to having, respectively, an **AsmNoW** or **AsmNoRW** assumption.

The wr-compiler's main function compile-cmd then outputs every register–assumption record pair (or *compilation record*) $C = (\Phi, \mathscr{S}) :: \mathit{CompRec} \triangleq \mathit{RegRec} \times \mathit{AsmRec}$ associated with the program state *before* execution of each instruction in the output `RISC` program.[5] A typical invocation to compile some $c :: \mathit{cmd}$ takes an *initial compilation record* C, and returns the *CompRec*-annotated `RISC` program $\mathit{PCs} :: (I \times \mathit{CompRec})\ \mathit{list}$ (i.e. map fst *PCs* recovers an unannotated `RISC` text), and a *final compilation record* C':

Example 5.1 (Example invocation of the COVERN wr-compiler)**.**

$$(\mathit{PCs}, l', nl', C', \mathit{failed}) = \mathsf{compile\text{-}cmd}\ C\ l\ nl\ c$$

The remainder of this section will focus on formal properties of the compilation records output alongside each `RISC` text: Section 5.1.1 will elaborate on checks enforced on input

[5] For readability, we will use regrec, asmrec to denote a *CompRec*'s (resp.) fst, snd projections.

programs with the help of *AsmRec*s, and Section 5.1.2 will present a resulting property that *RegRec*s track stable expressions, needed to prove security preservation (in Section 5.3).

Remaining details (e.g. l, l', nl, nl' for label allocation) will be relegated to appendices. We note here only that (1) compile-cmd may return True for *failed* to reject the input program, such as when it detects a race condition (described further in Section 5.1.1), or if expression depth exceeds the limit assumed by the register allocation scheme model (elided to Appendix B); also, (2) relative to the label allocation scheme (elided to Appendix A) we proved that the control flow of each program fragment compiled by the wr-compiler remains self-contained even when composed sequentially with other such fragments.

5.1.1 Requirements on inputs to the wr-compiler

We define a shared variable v to be recorded as assumed *stable* if it and all its control variables (i.e. $\mathscr{C}\,\mathsf{vars}\ v$) cannot presently be written to by another thread—that is, if they are recorded as having either of **AsmNoW** or **AsmNoRW** active on them. Formally:

Definition 5.1 (Stability of variable v according to assumption record $\mathscr{S}$)**.**

$$\mathsf{var\text{-}stable}\ \mathscr{S}\ v\ \triangleq\ v \in (\mathsf{fst}\ \mathscr{S} \cup \mathsf{snd}\ \mathscr{S})\ \wedge\ (\forall v' \in \mathscr{C}\,\mathsf{vars}\ v.\ v' \in (\mathsf{fst}\ \mathscr{S} \cup \mathsf{snd}\ \mathscr{S}))$$

For register record entries to be of any help in ensuring consistency of `While` and `RISC` expression evaluation, we exclude expression evaluation on race-prone variables by lifting the concept of stability to register records. The following predicate asserts internal consistency of the compilation record C created by compile-cmd, in the sense that the register record may only map to expressions that mention variables that are recorded as stable by the assumption record accompanying it. (Here, ran denotes the *range* of a map.)

Definition 5.2 (Stability of the register record in compilation record C)**.**

$$\mathsf{regrec\text{-}stable}\ C\ \triangleq\ \forall e \in \mathsf{ran}\ (\mathsf{regrec}\ C).\ (\forall v \in \mathsf{exp\text{-}vars}\ e.\ \mathsf{var\text{-}stable}\ (\mathsf{asmrec}\ C)\ v)$$

We then implement a collection of stability-checks :: $cmd \times CompRec \Rightarrow bool$ (called no-unstable-exprs in Sison & Murray (2019)) as a recursive function on the structure of `While` programs, that compile-cmd will use to ensure the following requirements of the given *cmd* if started with a configuration consistent with the given *CompRec*:

- The first two requirements ensure that programs comply with the locking discipline:
  - The requirement regarding reading *from* shared variables establishes the main outcome of freedom from race conditions we described at the beginning of Section 5.1: The program must not refer to expressions on any unstable variables.

    As a matter of convenience, instead of introducing a dedicated primitive to the `While` language for reading atomically from a single (otherwise-unstable) device memory location, in our case study model of Section 6 we model such interactions using a simple assignment $x := y$ protected by a "read-atomicity" lock on a shared memory location y that models the hardware interface.[6]

[6] When such atomic hardware primitives exist on a given architecture, we expect it would be straightforward for source languages to expose them and oblige their architecture-specific compilers to compile them to that single atomic instruction in the target language's semantics, which would eliminate the need for such locks.

  - If the program assigns *to* an unstable shared variable, then it must not be a lock-governed one according to the locking discipline. This prevents the violation of any guarantees not to write to the variable (due to not holding its lock).

- The remaining two requirements follow some simplifying assertions, originally made by the security type system of Murray *et al.* (2016b), that ensure mode state remains consistent after conditional branching and looping:

  - The two sides of any **if**-conditional branches in the program must both end with, effectively, the same set of locks held—to be precise, judging by their effect on the mode state, as captured by the assumption record.
  - For similar reasons, we require any **while**-loops in the program to restore the original set of locks held on loop entry (again, as captured by the assumption record) on loop termination.

  We believe these to be reasonable simplifications given that, in our setting, the set of variables governed by each lock does not change at runtime in such a way that would require access to them to be lock protected (or not) in a conditional manner.

Together, regrec-stable C and stability-checks $c\ C$ make up the main two requirements of a predicate compile-cmd-input-reqs $C\ l\ nl\ c$ imposed on the input arguments to compile-cmd. (Its other two requirements reflect that the terminated `While` program **stop** has no valid compilation, and that the initial label, if provided, must be valid—see Appendix A.) If any of these requirements are violated, compile-cmd rejects the program with $failed = \mathsf{True}$:

Definition 5.3 (Requirements on inputs to compile-cmd)**.**

$$\begin{aligned}\text{compile-cmd-input-reqs}\ C\ l\ nl\ c \triangleq\ & \text{stability-checks}\ c\ C \wedge \text{regrec-stable}\ C \wedge \\ & c \neq \textbf{\textit{stop}} \wedge (\forall x.\ l = \mathsf{Some}\ x \longrightarrow x < nl)\end{aligned}$$

5.1.2 Proof that all tracked register contents are stable

Imposing the predicate compile-cmd-input-reqs (Definition 5.3) gives us enough information to prove a lemma that compile-cmd only ever outputs stable register records, that attest to the fact that registers contain the results of evaluating expressions on stable variables.

Stated more precisely, every `RISC` program returned by a successful invocation of compile-cmd is annotated by *CompRec*s all with stable register records, and furthermore that the final *CompRec*'s register record is also stable:

Lemma 5.4 (Successful compilations output only stable register records)**.**

$$\frac{(PCs, l', nl', C', \mathsf{False}) = \text{compile-cmd}\ C\ l\ nl\ c \qquad \text{compile-cmd-input-reqs}\ C\ l\ nl\ c}{(\forall pc < \mathsf{length}\ PCs.\ \text{regrec-stable}\ (\mathsf{map}\ \mathsf{snd}\ PCs)[pc]) \wedge \text{regrec-stable}\ C'}$$

Proof. By induction on the structure of the `While` language program c, making reference to the implementation of compile-cmd.

For cases that must compile expressions, we furthermore prove and make use of a lemma by induction on the structure of expressions, making reference to the implementation of the

expression compiler function compile-expr called by compile-cmd. In essence, we prove that (sub)expressions appearing in register records must be stable, for two reasons:

First, they are always only ever subexpressions over variables that must have been stable in the input program when their contents were first loaded into registers.

Second, when compiling an **unlock**(k), the wr-compiler will always flush all register records that make reference to any variables that the **unlock**(k) makes unstable. □

5.2 Preserving a ban on secret-dependent control flow

The wr-compiler assumes that input `While` programs have no *conditional branches on* High-*sensitivity values* (High-*branching*), and therefore no secret-dependent control flow. This is a restriction commonly applied as a means to prevent all implicit flows, including timing leaks. This restriction will then be preserved by the wr-compiler for its output `RISC` programs, reflected primarily in the design of the concrete coupling invariant $\mathscr{I}_{\mathsf{wr}}$ (see Section 5.3.3).

Specifically, the wr-compiler assumes that the confidentiality of input `While` programs is witnessed by a strong low-bisimulation modulo modes with an extra requirement (supplied as a parameter, as in Section 2.2) that effectively disallows any present or past High-branching. Relying on the fact that a low-bisimulation already asserts Low-equivalence of memories, the extra requirement asserts that it furthermore pairs only configurations at the same program location, and that any **if**-conditional expressions must evaluate to the same value in both configurations' memories. Here, the helper function leftmost-cmd gives the leftmost in a sequence of ;-separated `While`-language commands:

Definition 5.5 (An extra requirement for low-bisimulations $\mathscr{B}$ to ban High-branching)**.**

$$\begin{aligned}&\mathsf{no\text{-}high\text{-}branching}\ \mathscr{B} \triangleq \\ &\forall c\ c'\ mds\ mem\ mem'.\ (\langle c, mds, mem\rangle_{\mathsf{w}}, \langle c', mds, mem'\rangle_{\mathsf{w}}) \in \mathscr{B} \longrightarrow c = c' \wedge \\ &\quad (\forall e\ c_1\ c_2.\ \mathsf{leftmost\text{-}cmd}\ c = \textbf{\textit{if}}\ e\ \textbf{\textit{then}}\ c_1\ \textbf{\textit{else}}\ c_2\ \textbf{\textit{fi}} \longrightarrow \mathsf{ev}_{\mathsf{exp}}\, mem\ e = \mathsf{ev}_{\mathsf{exp}}\, mem'\ e)\end{aligned}$$

Then, in Section 5.4, we will prove that the wr-compiler produces confidential `RISC` programs with no secret-dependent control flow, as witnessed by a low-bisimulation that asserts a similar extra requirement for `RISC` programs. In effect, this is the *pc-security* notion of Molnar *et al.* (2006), but also explicitly equating the program text:

Definition 5.6 (A pc-security–like requirement for `RISC` bisimulations $\mathscr{B}$)**.**

$$\begin{aligned}&\mathsf{pc\text{-}security}\ \mathscr{B} \triangleq\ \forall pc\ pc'\ P\ P'\ regs\ regs'\ mds\ mem\ mem'. \\ &(\langle((pc, P), regs), mds, mem\rangle_{\mathsf{r}}, \langle((pc', P'), regs'), mds, mem'\rangle_{\mathsf{r}}) \in \mathscr{B} \longrightarrow pc = pc' \wedge P = P'\end{aligned}$$

5.3 Use of the decomposition principle

Having covered the most relevant aspects of the wr-compiler's implementation, we now present the refinement relation $\mathscr{R}_{\mathsf{wr}}$ (in Section 5.3.1), pacing function $\mathsf{abs\text{-}steps}_{\mathsf{wr}}$ (in Section 5.3.2), and concrete coupling invariant $\mathscr{I}_{\mathsf{wr}}$ (in Section 5.3.3), parameters we use to apply the decomposition principle we presented in Section 2.4 to prove (in Section 5.3.4) that

successful compilations are legitimised by secure-refinement (Definition 2.15)—the desired confidentiality-preserving notion of refinement for mixed-sensitivity concurrent programs.

The strategy laid out by the decomposition principle will be to prove that these parameters satisfy decomp-refinement-safe (Definition 2.19) for a targeted class of input `While`-language programs—ones with no secret-dependent control flow, as we specified in Section 5.2—meaning (for such programs) we can use the parameters to enforce that wr-compiler introduces no secret-dependent inconsistencies in termination, timing behaviour, or assume–guarantee modes.

In doing so we avoid a direct proof of the cube-shaped refinement diagram (Figure 3) of Murray *et al.* (2016b)—which would have involved reasoning about both $\mathscr{R}_{\mathsf{wr}}$ and $\mathscr{I}_{\mathsf{wr}}$ at once—and instead prove (with the assistance of $\mathsf{abs\text{-}steps}_{\mathsf{wr}}$) a square-shaped refinement diagram for $\mathscr{R}_{\mathsf{wr}}$ (Figure 4a) more typically found in compiler verification.

5.3.1 Refinement relation $\mathscr{R}_{\mathsf{wr}}$ and its invariants

In this section we introduce the refinement relation $\mathscr{R}_{\mathsf{wr}}$ that characterises compilation of programs from `While` to `RISC` using the wr-compiler, and prove it satisfies the two properties demanded of $\mathscr{R}_{\mathsf{wr}}$ (alone) by formal secure-refinement (Definition 2.15):

1. Preservation of modes and all contents of shared memory (preserves-modes-mem, Definition 2.13), and

2. Closedness under changes by other threads (closed-others, Definition 2.14).

An actual proof of refinement (using the square-shaped diagram of Figure 4a) for $\mathscr{R}_{\mathsf{wr}}$ will be deferred to Section 5.3.2, which introduces the $\mathsf{abs\text{-}steps}_{\mathsf{wr}}$ function pacing it.

Just like the earlier example of a secure refinement relation (in Figure 2), the refinement relation $\mathscr{R}_{\mathsf{wr}}$ pairs abstract (here, `While`-language) with concrete (here, `RISC`-language) program configurations. For example, the `if_expr` case of $\mathscr{R}_{\mathsf{wr}}$ relates the expression-evaluation part of the `While` command **if** e **then** c_1 **else** c_2 **fi**, with the corresponding part of the `RISC` program obtained by running compile-cmd on it, including the conditional jump **Jz** after expression evaluation. (This case is depicted in Figure 10, and a relevant excerpt of the compile-cmd implementation provided in Figure 11 for comparison, both on page 63 of Appendix C. An informal description of all the cases of $\mathscr{R}_{\mathsf{wr}}$, their purpose, and the invariants they maintain, can also be found in Appendix C.)

We define almost all the cases of $\mathscr{R}_{\mathsf{wr}}$ to assert at least one successful run of compile-cmd (where $\mathit{failed} = \mathsf{False}$). We then define a guard that we impose to restrict the scope of $\mathscr{R}_{\mathsf{wr}}$ only to consider local program configurations consistent with the relevant compilation record produced by compile-cmd. In short, this ensures the actual values in the register bank *regs* equal any expression the register record says they should have, as evaluated under the current *mem*; and furthermore, that the assumption record is consistent with the **AsmNoW** and **AsmNoRW** modes in the actual *mds*. Formally:

Definition 5.7 (Configuration consistency requirements for compiled commands)**.**

compiled-cmd-config-consistent C *regs mds mem* $\triangleq$

regrec-mem-consistent (regrec C) *regs mem* $\wedge$ asmrec-mds-consistent (asmrec C) *mds*

where

regrec-mem-consistent Φ *regs mem* $\triangleq$ $\forall r\ e.\ \Phi\ r = \mathsf{Some}\ e \longrightarrow$ *regs* $r = \mathsf{ev}_{\mathsf{exp}}$ *mem* e

(Consistency between register record, register bank, and shared memory)

asmrec-mds-consistent $\mathscr{S}$ *mds* $\triangleq$ $\mathscr{S} = (mds\ \mathbf{AsmNoW},\ mds\ \mathbf{AsmNoRW})$

(Consistency between an assumption record and a mode state)

Apart from using compiled-cmd-config-consistent to restrict the scope of $\mathscr{R}_{\mathsf{wr}}$ in this manner, we will also impose it in Section 5.3.4 as *initial configuration requirements* for compiled programs: Only configurations obeying them may be used to initialise a `RISC` program compiled by the wr-compiler with initial *CompRec C*.

The cases of $\mathscr{R}_{\mathsf{wr}}$ also tend to assert regrec-stable (Definition 5.2), which we already proved holds for all compilation records produced by the wr-compiler (Lemma 5.4).

Finally, whenever a case of $\mathscr{R}_{\mathsf{wr}}$ is inductive (e.g. the `if_expr` case, for its nested calls to compile-cmd for each of its "then" and "else" branches) it quantifies over all configurations that obey compiled-cmd-config-consistent (Definition 5.7) and regrec-stable (Definition 5.2) relative to the initial compilation record given to each nested call to compile-cmd.

With $\mathscr{R}_{\mathsf{wr}}$ thus specified, we can now prove the two requirements for secure-refinement that pertain to $\mathscr{R}_{\mathsf{wr}}$ alone: preserves-modes-mem (Definition 2.13), and closed-others (Definition 2.14). In short, preserves-modes-mem is largely enforced by the definition of $\mathscr{R}_{\mathsf{wr}}$, but closed-others relies in part on $\mathscr{R}_{\mathsf{wr}}$ only ever talking about stable register records:

Lemma 5.8 ($\mathscr{R}_{\mathsf{wr}}$ preserves modes and memory)**.**

$$\mathsf{preserves\text{-}modes\text{-}mem}\ \mathscr{R}_{\mathsf{wr}}$$

Proof. By induction on the structure of $\mathscr{R}_{\mathsf{wr}}$.

For all cases of $(lc_w, lc_r) \in \mathscr{R}_{\mathsf{wr}}$, $lc_w =^{\mathsf{mem}}_{\mathsf{mds}} lc_r$ is either asserted directly by the guards or obtainable from the inductive hypothesis. □

Lemma 5.9 ($\mathscr{R}_{\mathsf{wr}}$ is closed under changes by others)**.**

$$\mathsf{closed\text{-}others}\ \mathscr{R}_{\mathsf{wr}}$$

Proof. By induction on the structure of $\mathscr{R}_{\mathsf{wr}}$.

Changes by others (Definition 2.14) only modify writable variables the same way for both configurations, so preservation of $=^{\mathsf{mem}}_{\mathsf{mds}}$ is immediate. Also, regrec-mem-consistent is unaffected because by Lemma 5.4, compile-cmd only creates regrec-stable records—i.e. referring to no writable variables. No other $\mathscr{R}_{\mathsf{wr}}$ guards mention shared memory. □

5.3.2 Refinement pacing function abs-steps$_{\mathsf{wr}}$

In this section we nominate a pacing function, abs-steps$_{\mathsf{wr}}$, specifying the number of evaluation steps with which a `While` program should simulate each step of the `RISC` program

to which the wr-compiler compiled it. Using the square-shaped "refinement preservation" diagram of Figure 4a (part of Definition 2.18), we then prove that the $\mathscr{R}_{\mathsf{wr}}$ relation we introduced in Section 5.3.1 is a refinement when "paced" by $\mathsf{abs\text{-}steps}_{\mathsf{wr}}$ in this manner.

Here we define $\mathsf{abs\text{-}steps}_{\mathsf{wr}}$ to depend only on the current program location; consequently, as long as the wr-compiler introduces no secret-dependent control flow, it will also introduce no timing leaks—that is, no secret-dependent variations to the pacing of the program, as disallowed by Figure 4b (part of Definition 2.19)—which we will be obliged to prove in Section 5.3.4. To this end, $\mathsf{abs\text{-}steps}_{\mathsf{wr}}$ primarily looks at the form of the `RISC` instruction (sometimes `While` command) about to be executed, dividing them into three categories:

- Instructions output by compile-expr: **Load**, **Op**, and **MoveK**. For these, $\mathsf{abs\text{-}steps}_{\mathsf{wr}}$ returns 1 if the leftmost-cmd (the leftmost in a sequence of ;-separated commands) of the `While` program is "**while** e **do** c **od**", to allow it to step to "**if** e **then** $(c$; **while** e **do** c **od**$)$ **else stop fi**" concurrently with the first `RISC` step of the compiled expression itself. Otherwise, $\mathsf{abs\text{-}steps}_{\mathsf{wr}}$ returns 0, to indicate the `While` program standing still while the `RISC` program takes *new* steps to evaluate the expression.
- "Epilogue" steps: **Jmp** and **Nop** when used for control flow at the end of a smaller compiled program in the context of a larger one. For these, $\mathsf{abs\text{-}steps}_{\mathsf{wr}}$ returns 0.
- All other `RISC` instructions are assumed to proceed at a lockstep pace with the `While` command they were compiled from, and for these $\mathsf{abs\text{-}steps}_{\mathsf{wr}}$ returns 1.

Having nominated $\mathsf{abs\text{-}steps}_{\mathsf{wr}}$ and $\mathscr{R}_{\mathsf{wr}}$, we now have the parameters over which we are obliged, by secure-refinement-decomp (Definition 2.18), to prove refinement preservation (Figure 4a). To this end, we prove firstly that every step of execution of a `RISC` program, produced by the wr-compiler from a `While` program, maintains the consistency demanded by compiled-cmd-config-consistent between configurations and compilation records:

Lemma 5.10 (Successfully compiled programs maintain config consistency requirements)**.**

$$\frac{\begin{array}{c}(PCs, l', nl', C', failed) = \mathsf{compile\text{-}cmd}\ C\ l\ nl\ c \qquad \mathsf{compile\text{-}cmd\text{-}input\text{-}reqs}\ C\ l\ nl\ c \\ failed = \mathsf{False} \qquad pc < \mathsf{length}\ PCs \qquad P = \mathsf{map}\ \mathsf{fst}\ PCs \qquad Cs = \mathsf{map}\ \mathsf{snd}\ PCs \\ \mathsf{compiled\text{-}cmd\text{-}config\text{-}consistent}\ Cs[pc]\ regs\ mds\ mem \\ \langle((pc,P),regs),mds,mem\rangle_{\mathsf{r}} \rightsquigarrow_{\mathsf{r}} \langle((pc',P),regs'),mds',mem'\rangle_{\mathsf{r}})\end{array}}{\mathsf{compiled\text{-}cmd\text{-}config\text{-}consistent}\ (\texttt{if}\ pc' < \mathsf{length}\, P\ \texttt{then}\ Cs[pc']\ \texttt{else}\ C')\ regs'\ mds'\ mem'}$$

Proof. Unfolding Definition 5.7, we in fact prove it separately for regrec-mem-consistent and asmrec-mds-consistent, both times by induction on the structure of `While` program c.

In each case, we use the simplifiers for the compile-cmd implementation to yield the corresponding `RISC` program fragment in question, and then prove the lemma for each of the possible locations of pc in the compiled program. For both proofs, there is some trickiness in accounting for (and ruling out) which destination pc' must be considered for each of these cases of pc, particularly for those `While` programs that compile to `RISC` programs that may have jumps in them.

Control flow trickiness aside, the intuition for regrec-mem-consistent is that it tests the correctness of the compilation of expressions. For this we prove a sublemma for maintenance of compiled-cmd-config-consistent, by induction on the structure of expressions e

that are encountered in the `While` programs **if** e **then** c_1 **else** c_2 **fi**, **while** e **do** c' **od**, and $v := e$. Additionally, **unlock**(k) flushes register record entries mentioning variables that are to become unstable, and **while** e **do** c' **od** conservatively flushes entries to force evaluation of the loop condition expression. This is safe trivially because flushing entries can never make a consistent register record inconsistent. The rest of the cases for c are straightforward because they do not touch the register record.

Then for asmrec-mds-consistent, the substantial part of the proof is as a test of the correctness of the compiler's bookkeeping of assumptions being consistent with the semantics of **lock**(k) and **unlock**(k). The other cases for c do not touch the mode state. □

Also, we must prove a correctness lemma for the expression compiler:

Lemma 5.11 (Correctness of the expression compiler)**.**

$$(PCs, r, C', \mathsf{False}) = \mathsf{compile\text{-}expr}\ C\ A\ l\ e \implies (\mathsf{regrec}\ C')\ r = \mathsf{Some}\ e$$

Proof. By induction on the structure of expressions e, using the simplification rules for the implementation of compile-expr, and also relying on assumptions of correctness of the register allocation scheme supplied by the instantiator of the theory. □

Armed with these facts, we can now prove the main refinement preservation result:

Lemma 5.12 ($\mathscr{R}_{\mathsf{wr}}$ is a refinement paced by $\mathsf{abs\text{-}steps}_{\mathsf{wr}}$)**.**

$$\begin{aligned}\forall lc_w\ lc_r.\ (lc_w, lc_r) \in \mathscr{R}_{\mathsf{wr}} \longrightarrow\ & (\forall lc_r'.\ lc_r \rightsquigarrow_{\mathsf{r}} lc_r' \longrightarrow \\ & (\exists lc_w'.\ lc_w \rightsquigarrow_{\mathsf{w}}^{(\mathsf{abs\text{-}steps}_{\mathsf{wr}}\ lc_w\ lc_r)} lc_w' \wedge (lc_w', lc_r') \in \mathscr{R}_{\mathsf{wr}}))\end{aligned}$$

Proof. By induction on the structure of $\mathscr{R}_{\mathsf{wr}}$. (Refer to Appendix C for an informal description of all cases of $\mathscr{R}_{\mathsf{wr}}$.)

The base case `stop` is immediate, as it pertains to a terminated `While` and `RISC` program. The base cases that proceed in one step to a terminating program configuration (`skip_nop`, `assign_store`, `lock_acq`, `lock_rel`) are fairly straightforward because after dealing with the single step, the resulting obligation can then be handled by the `stop` case. This leaves the last remaining base case `assign_expr`, which proceeds in one step either to itself, or to `assign_store`. In all these cases, we use Lemma 5.10 to obtain the preservation of the guards demanded by the $\mathscr{R}_{\mathsf{wr}}$ introduction rule for the destination configuration of the step. Particularly, the `assign_store` case must make use of regrec-mem-consistent and the correctness of compile-expr (Lemma 5.11) to ensure that once the evaluated expression is written back to shared memory, $lc_w' =_{\mathsf{mds}}^{\mathsf{mem}} lc_r'$ holds as demanded by the `stop` case.

The inductive cases that concern expression evaluation (`if_expr`, `while_expr`) are much like `assign_expr` in that they have the possibility of progressing in one step to themselves. Unlike `assign_expr` however, their other possibility is a conditional jump based on the result of that expression. Again we use Lemma 5.11 to obtain that the result is an accurate calculation of the expression, and this time we prove by the two different cases whether `if_expr` ends up in `if_c1` or `if_c2`, or if `while_expr` ends up in `while_inner` or at `stop` (having jumped to the exit label). In these cases, the guards over which the inductive references to $\mathscr{R}_{\mathsf{wr}}$ have been quantified are versatile enough to discharge themselves (when

`*_expr` steps to itself), or to discharge any reachable initial starting state for the nested compiled `RISC` program, given that Lemma 5.10 ensures the invariance of these guards.

This just leaves the inductive cases that pertain to configurations inside a nested compiled `RISC` program (`if_c1`, `if_c2`, `while_inner`), or at the end of one (`epilogue_step`, `while_loop`). In these cases, the inductive hypotheses obtained from the inductive reference to $\mathscr{R}_{\mathsf{wr}}$ are always enough to satisfy the guards demanded by the possible destination cases. Like in the proof of Lemma 5.10, the trickiness mostly comes from accounting for all the possible cases of control flow (ruling out spurious destinations) that need to be considered. □

5.3.3 Concrete coupling invariant $\mathscr{I}_{\mathsf{wr}}$

The next element needed is the concrete coupling invariant $\mathscr{I}_{\mathsf{wr}}$. Recall from Section 5.2 that the no-high-branching requirement (Definition 5.5) ensures that input `While` programs have no secret-dependent control flow; here we choose $\mathscr{I}_{\mathsf{wr}}$ to ensure that the wr-compiler has not introduced any *new* secret-dependent control flow in the output `RISC` program.

We define $\mathscr{I}_{\mathsf{wr}}$ formally to assert that the witness strong low-bisimulation (modulo modes) to be derived for the output program only pairs local configurations that are at the same location $pc = pc'$ of the same `RISC` program $P = P'$:

Definition 5.13 (Concrete coupling invariant $\mathscr{I}_{\mathsf{wr}}$ for compiled programs)**.**

$$\mathscr{I}_{\mathsf{wr}} \triangleq \{(\langle((pc,P),regs),mds,mem\rangle_{\mathsf{r}}, \langle((pc',P'),regs'),mds',mem'\rangle_{\mathsf{r}}) \mid (pc,P) = (pc',P')\}$$

From this definition, pc-security (Definition 5.6) is clearly immediate for any concrete bisimulation $\mathscr{B}_{\mathsf{C}}\mathsf{of}\ \mathscr{B}\ \mathscr{R}\ \mathscr{I}_{\mathsf{wr}}$ (Definition 2.16) derived using $\mathscr{I}_{\mathsf{wr}}$.

5.3.4 Proof of CVDNI-preserving refinement

With $\mathscr{R}_{\mathsf{wr}}$, $\mathsf{abs\text{-}steps}_{\mathsf{wr}}$, and $\mathscr{I}_{\mathsf{wr}}$ nominated, we are ready to prove confidentiality-preserving refinement using the decomposition principle secure-refinement-decomp (Definition 2.18).

To this end, we now prove the suitability of these three parameters, for `While` programs that do not branch on High-sensitivity values (as we specified earlier, in Section 5.2):

Lemma 5.14 ($\mathscr{R}_{\mathsf{wr}}, \mathsf{abs\text{-}steps}_{\mathsf{wr}}, \mathscr{I}_{\mathsf{wr}}$ are safe for secure-refinement decomposition)**.**

$$\frac{\mathsf{strong\text{-}low\text{-}bisim\text{-}mm}\ \mathscr{B} \qquad \mathsf{no\text{-}high\text{-}branching}\ \mathscr{B}}{\mathsf{decomp\text{-}refinement\text{-}safe}\ \mathscr{B}\ \mathscr{R}_{\mathsf{wr}}\ \mathscr{I}_{\mathsf{wr}}\ \mathsf{abs\text{-}steps}_{\mathsf{wr}}}$$

Proof. Unfolding Definition 2.19 gives us the following obligations. (See also Figure 4.)

For consistent stopping behaviour, we prove a lemma that `RISC` programs stop if and only if their pc is outside the program text P, i.e. $pc > \mathsf{length}\ P$. Because $\mathscr{I}_{\mathsf{wr}}$ equates pc and P for the two configurations, then clearly both have identical stopping behaviour.

For consistency of change in timing behaviour, $\mathsf{abs\text{-}steps}_{\mathsf{wr}}$ depends only on `While` and `RISC` program locations, and no-high-branching and $\mathscr{I}_{\mathsf{wr}}$ forces them (respectively) to be equal for the local configurations under consideration.

For closedness of $\mathscr{I}_{\mathsf{wr}}$ under lockstep execution, the only non-straightforward cases to consider are conditional branching, and the locking primitives. For conditional branching, we use no-high-branching for $\mathscr{B}$ with memory preservation via $\mathscr{R}_{\mathsf{wr}}$ (Lemma 5.8) to ensure that the conditional branching outcome is the same on both sides.

Finally, as the only operations that touch mode state, the locking primitives are the only non-straightforward cases for modes-equality maintenance under lockstep execution. As all lock memory is classified Low (Proposition 3.4), we use strong-low-bisim-mm for $\mathscr{B}$ with memory preservation via $\mathscr{R}_{\mathsf{wr}}$ to ensure the RISC configurations behave consistently. □

Lemma 5.15 ($\mathscr{R}_{\mathsf{wr}}, \mathsf{abs\text{-}steps}_{\mathsf{wr}}, \mathscr{I}_{\mathsf{wr}}$ meet decomposed secure-refinement requirements)**.**

$$\frac{\mathsf{strong\text{-}low\text{-}bisim\text{-}mm}\ \mathscr{B} \qquad \mathsf{no\text{-}high\text{-}branching}\ \mathscr{B}}{\mathsf{secure\text{-}refinement\text{-}decomp}\ \mathscr{B}\ \mathscr{R}_{\mathsf{wr}}\ \mathscr{I}_{\mathsf{wr}}\ \mathsf{abs\text{-}steps}_{\mathsf{wr}}}$$

Proof. Unfolding Definition 2.18, the obligations pertaining only to $\mathscr{R}_{\mathsf{wr}}$ and $\mathsf{abs\text{-}steps}_{\mathsf{wr}}$ are discharged by Lemma 5.12, Lemma 5.9, and Lemma 5.8. Pertaining to $\mathscr{I}_{\mathsf{wr}}$: Clearly $\mathscr{I}_{\mathsf{wr}}$ is symmetric, and furthermore it is cg-consistent (Definition 2.6) because the actions over which $\mathscr{I}_{\mathsf{wr}}$ must be closed modify only the shared memory, and $\mathscr{I}_{\mathsf{wr}}$ places only restrictions on the program text and current location. The final obligation (regarding decomp-refinement-safe) is discharged by Lemma 5.14. □

From this it follows immediately via Theorem 2.20 that $\mathscr{R}_{\mathsf{wr}}$ with the help of $\mathscr{I}_{\mathsf{wr}}$ describes a confidentiality-preserving refinement for non-High-branching `While` programs:

Corollary 5.16 ($\mathscr{R}_{\mathsf{wr}}$ is a secure refinement for non-High-branching programs)**.**

$$\frac{\mathsf{strong\text{-}low\text{-}bisim\text{-}mm}\ \mathscr{B} \qquad \mathsf{no\text{-}high\text{-}branching}\ \mathscr{B}}{\mathsf{secure\text{-}refinement}\ \mathscr{B}\ \mathscr{R}_{\mathsf{wr}}\ \mathscr{I}_{\mathsf{wr}}}$$

Finally we prove that successful compilation produces a RISC program related by $\mathscr{R}_{\mathsf{wr}}$ to its input `While` program, when started with corresponding (same mds, mem) and reasonable (according to compiled-cmd-config-consistent) initial configurations:

Theorem 5.17 (Successful compilations are refinements in $\mathscr{R}_{\mathsf{wr}}$)**.**

$$\frac{\begin{array}{c}(PCs, l', nl', C', failed) = \mathsf{compile\text{-}cmd}\ C\ l\ nl\ c \qquad \mathsf{compile\text{-}cmd\text{-}input\text{-}reqs}\ C\ l\ nl\ c \\ failed = \mathsf{False} \qquad \mathsf{compiled\text{-}cmd\text{-}config\text{-}consistent}\ C\ regs\ mds\ mem \qquad P = \mathsf{map\ fst}\ PCs\end{array}}{(\langle c, mds, mem\rangle_{\mathsf{w}}, \langle((0,P), regs), mds, mem\rangle_{\mathsf{r}}) \in \mathscr{R}_{\mathsf{wr}}}$$

Proof. By induction on the structure of the `While`-language.

The compiler input and initial configuration conditions we impose allow us to have each of **skip**, $cmd\,;cmd$, **if** exp **then** cmd **else** cmd **fi**, **while** exp **do** cmd **od**, $v := exp$, **lock**(k), and **unlock**(k) and their compiled output meet the guards of the introduction rules for the cases `skip`, `seq`, `if_expr`, `while_expr`, `assign_expr`, `lock_acq`, and `lock_rel` of $\mathscr{R}_{\mathsf{wr}}$ (described further in Appendix C) that we designed for them, respectively. □

5.4 Proof of compositional noninterference preservation

Going beyond the level of detail of our presentation in Sison & Murray (2019), we now present the final few steps to obtain preservation of whole-system security for concurrent compositions of RISC threads when all are obtained via compilation by the wr-compiler (Section 5.4.1). In addition to this, we obtain preservation of per-thread compositional security for each program thread compiled, and other properties that may be useful for their composition with RISC threads proved secure directly at the RISC level (Section 5.4.2).

5.4.1 Whole-system security preservation

To use the whole-system refinement theorem (Theorem 2.23), we are obliged to show that, in addition to establishing a secure-refinement (Definition 2.15, which we just showed in Section 5.3), the wr-compiler also preserves local-mode-compliance as demanded by compositional-refinement (Definition 2.22). Then, as we noted in Section 2.5, there is no need for us to prove preservation of the non-compositional global-modes-compatibility condition—the whole-system refinement theorem takes care of that.

The local compliance preservation result follows from a property of the refinement relation, $\mathscr{R}_{\mathsf{wr}}$. Here, "respects-own-guarantees" is from Definition 2.11:

Lemma 5.18 (Each step from a RISC configuration in $\mathscr{R}_{\mathsf{wr}}$ respects its own guarantees)**.**

$$\frac{(\langle c, mds, mem\rangle_{\mathsf{w}}, \langle((pc, P), regs), mds, mem\rangle_{\mathsf{r}}) \in \mathscr{R}_{\mathsf{wr}}}{\mathsf{respects\text{-}own\text{-}guarantees}\ (((pc, P), regs), mds)}$$

Proof. By induction on the structure of $\mathscr{R}_{\mathsf{wr}}$.

Knowing that the While command does not access lock-governed variables without holding the relevant lock (via the stability-checks asserted as part of compile-cmd-input-reqs by every relevant case of $\mathscr{R}_{\mathsf{wr}}$), we are obliged to show that the RISC instruction paired to it by $\mathscr{R}_{\mathsf{wr}}$ similarly respects the guarantee modes implied by the locking discipline (as specified in Section 3.1). We do so with a mixed Isar/"apply"-style proof that exercises the relevant cases of the RISC semantics, using lemmas about control flow under sequential composition (mentioned in Section 5.1; see also Appendix A). Propositions 3.8 and 3.9 also play a role in excluding certain cases from consideration. □

Lemma 5.19 (Refinements in $\mathscr{R}_{\mathsf{wr}}$ ensure local mode compliance)**.**

$$\frac{(\langle c, mds, mem\rangle_{\mathsf{w}}, \langle((pc, P), regs), mds, mem\rangle_{\mathsf{r}}) \in \mathscr{R}_{\mathsf{wr}}}{\mathsf{local\text{-}mode\text{-}compliance}\ \langle((pc, P), regs), mds, mem\rangle_{\mathsf{r}}}$$

Proof. Unfolding Definition 2.11, we must show that what was proved by Lemma 5.18 holds for every RISC configuration reachable from $\langle((pc, P), regs), mds, mem\rangle_{\mathsf{r}}$.

First, we prove a lemma that establishes that every such reachable RISC configuration is also paired by $\mathscr{R}_{\mathsf{wr}}$ to some While configuration. Specifically, we prove that $\mathscr{R}_{\mathsf{wr}}$ is closed under a notion of "pairwise reachability under mode-permitted havoc", wherein:

1. Every one step by the RISC program is matched by either zero or one step by the While program, as specified by $\mathsf{abs\text{-}steps}_{\mathsf{wr}}$ (Section 5.3.2).

2. Between each evaluation step, arbitrary changes are allowed to occur to the memory locations judged by the mode state to be writable (Definition 2.5).

Because all such RISC configurations reachable from the initial one are in $\mathscr{R}_{\mathsf{wr}}$, it then follows from Lemma 5.18 that they respect their own guarantees, as required. □

We then initialise the compiler with an empty C_0 :: *CompRec* that knows nothing about the register contents, and does not assume any variables to be stable:

Definition 5.20 (Empty compilation record C_0)**.**

$$\mathsf{C}_0 \triangleq ((\lambda\_\,.\,\mathsf{None}),(\emptyset,\emptyset))$$

With these definitions we have the desired consistency result:

Lemma 5.21 (Initial $\mathsf{C}_0, \mathsf{mds}_0$ are consistent with no-locks-held)**.**

$$\mathsf{no\text{-}locks\text{-}held}\ \mathit{mem} \implies \mathsf{compiled\text{-}cmd\text{-}config\text{-}consistent}\ \mathsf{C}_0\ \mathit{regs}\ \mathsf{mds}_0\ \mathit{mem}$$

Proof. This is straightforward by unfolding Definitions 5.7, 3.18, 3.19, and 5.20, also relying on the cleanliness conditions Proposition 3.5 and Proposition 3.7 on locking disciplines specified in Section 3.2. □

We now have enough information to derive a whole-system security result, for concurrent RISC programs obtained by running the wr-compiler on any list "*cs*" of secure While commands (one for each thread in the program). As we explained in Section 5.2, the COVERN wr-compiler's preservation of security is only for programs with no-high-branching (Definition 5.5); furthermore, so that we can derive global compatibility for multiple of these programs run concurrently as threads (as per Section 3.4), we will impose no-locks-held (Definition 3.18) as an initial condition. Therefore, the security preservation theorem we choose to prove here demands that every thread of the input While program be $\mathsf{com\text{-}secure}^{\mathsf{no\text{-}high\text{-}branching}}_{\mathsf{no\text{-}locks\text{-}held}}$ (Definition 2.7, with additional requirements as specified). It then promises that the output program is $\mathsf{sys\text{-}secure}_{\mathsf{no\text{-}locks\text{-}held}}$:

Theorem 5.22 (Secure threads compiled by the wr-compiler form a secure system)**.**

$$\begin{array}{l}
\mathsf{length}\ \mathit{cms}_r = \mathsf{length}\ \mathit{cs} \wedge \\
\forall i < \mathsf{length}\ \mathit{cms}_r.\ \exists l\ \mathit{nl}\ \mathit{PCs}\ l'\ \mathit{nl}'\ C'\ \mathit{regs}. \\
\quad \mathsf{com\text{-}secure}^{\mathsf{no\text{-}high\text{-}branching}}_{\mathsf{no\text{-}locks\text{-}held}}\ (\mathit{cs}[i], \mathsf{mds}_0) \wedge \\
\quad (\forall \mathit{mem}.\ \mathsf{no\text{-}locks\text{-}held}\ \mathit{mem} \longrightarrow \mathsf{local\text{-}mode\text{-}compliance}\ \langle c, \mathsf{mds}_0, \mathit{mem}\rangle_{\mathsf{w}}) \wedge \\
\quad (\mathit{PCs}, l', \mathit{nl}', C', \mathsf{False}) = \mathsf{compile\text{-}cmd}\ \mathsf{C}_0\ l\ \mathit{nl}\ \mathit{cs}[i] \wedge\ \mathsf{compile\text{-}cmd\text{-}input\text{-}reqs}\ \mathsf{C}_0\ l\ \mathit{nl}\ \mathit{cs}[i] \wedge \\
\quad \mathit{cms}_r[i] = (((0, \mathsf{map\ fst}\ \mathit{PCs}), \mathit{regs}), \mathsf{mds}_0)
\end{array}$$

$$\mathsf{sys\text{-}secure}_{\mathsf{no\text{-}locks\text{-}held}}\ \mathit{cms}_r$$

Proof. We invoke Theorem 2.23, supplying:

- no-locks-held for the *INIT* parameter at both While and RISC level.

- $\mathscr{B}_{\mathsf{all}}$, $\mathscr{R}_{\mathsf{wr}}$, $\mathscr{I}_{\mathsf{wr}}$ to be respectively the witness bisimulation, refinement relation, and coupling invariant for all compiled threads, where we define $\mathscr{B}_{\mathsf{all}}$ to be the arbitrary union of all strong low-bisimulations modulo modes that disallow high-branching:

$$\mathscr{B}_{\mathsf{all}} \triangleq \bigcup \{\, \mathscr{B} \mid \mathsf{strong\text{-}low\text{-}bisim\text{-}mm}\ \mathscr{B} \wedge \mathsf{no\text{-}high\text{-}branching}\ \mathscr{B} \,\}$$

- mds_0 to be the initial mode state for all `While` threads in cs.

The first thing we must prove is that the original program satisfies sound-mode-use (Definition 2.10) when initialised with mds_0 and no-locks-held; we have the local part from this theorem's local-mode-compliance assumption, and the global part from Lemma 3.20.

We then discharge the demands of compositional-refinement $\mathscr{B}_{\mathsf{all}}\ \mathscr{R}_{\mathsf{wr}}\ \mathscr{I}_{\mathsf{wr}}$ (Definition 2.22) using Corollary 5.16, Lemma 5.19, and by unfolding Definition 5.13.

It only remains for us to show that the initial `RISC`–`While` and `While`–`While` configuration pairs of interest are captured respectively by $\mathscr{R}_{\mathsf{wr}}$ and $\mathscr{B}_{\mathsf{all}}$. We obtain the former using this theorem's assumptions and Lemma 5.21 to discharge the guards of Theorem 5.17. Finally, we use the assumption that the original program is $\mathsf{com\text{-}secure}^{\mathsf{no\text{-}high\text{-}branching}}_{\mathsf{no\text{-}locks\text{-}held}}$ and unfold Definition 2.7 to obtain that there exists some strong-low-bisim-mm $\mathscr{B}$ that enforces no-high-branching for every configuration pair with low-equal memories (modulo mds_0) and no-locks-held initially; therefore, these state pairs must all be captured by $\mathscr{B}_{\mathsf{all}}$. □

5.4.2 Per-thread compositional security preservation

For system developers who may want to run programs compiled from `While` to `RISC` concurrently with other programs written directly in `RISC`, per-thread security preservation results may be useful. To compose the security proofs for those threads, direct `RISC`-level lemmas for the "sound-mode-use" side conditions of the compositionality theorem (Theorem 2.8) will also be needed. We therefore present these as an alternative method to obtain compositional security results for `RISC` programs, applicable when only partially produced by compilation from `While` by the wr-compiler.

Given the facts we established in Section 5.3, we have straightforwardly that such programs' executions are captured by the bisimulation derived from $\mathscr{B}, \mathscr{R}_{\mathsf{wr}}, \mathscr{I}_{\mathsf{wr}}$, when started with reasonable initial configurations corresponding to those paired by $\mathscr{B}$:

Lemma 5.23 (Programs witnessed by $\mathscr{B}$ are captured by $\mathscr{B}_{\mathsf{C}}\mathsf{of}\ \mathscr{B}\ \mathscr{R}_{\mathsf{wr}}\ \mathscr{I}_{\mathsf{wr}}$ once compiled)**.**

$$\frac{\begin{array}{c}\mathsf{strong\text{-}low\text{-}bisim\text{-}mm}\ \mathscr{B} \qquad (\langle c, mds, mem_1\rangle_{\mathsf{w}}, \langle c, mds, mem_2\rangle_{\mathsf{w}}) \in \mathscr{B} \\ (PCs, l', nl', C', failed) = \mathsf{compile\text{-}cmd}\ C\ l\ nl\ c \qquad \mathsf{compile\text{-}cmd\text{-}input\text{-}reqs}\ C\ l\ nl\ c \\ failed = \mathsf{False} \qquad \mathsf{compiled\text{-}cmd\text{-}config\text{-}consistent}\ C\ regs\ mds\ mem_1 \qquad P = \mathsf{map\ fst}\ PCs \\ \mathsf{compiled\text{-}cmd\text{-}config\text{-}consistent}\ C\ regs\ mds\ mem_2\end{array}}{(\langle((0,P), regs), mds, mem_1\rangle_{\mathsf{r}}, \langle((0,P), regs), mds, mem_2\rangle_{\mathsf{r}}) \in \mathscr{B}_{\mathsf{C}}\mathsf{of}\ \mathscr{B}\ \mathscr{R}_{\mathsf{wr}}\ \mathscr{I}_{\mathsf{wr}}}$$

Proof. Straightforward from the definition of $\mathscr{B}_{\mathsf{C}}\mathsf{of}$ (Definition 2.16), using Theorem 5.17 to show membership of $\mathscr{R}_{\mathsf{wr}}$, and the definition of strong-low-bisim-mm (Definition 2.4) to show that the memories are low-equal modulo modes, as required by $\mathscr{B}_{\mathsf{C}}\mathsf{of}$. Finally, membership of $\mathscr{I}_{\mathsf{wr}}$ (Definition 5.13) follows from the fact that the paired configurations are at the same location (program counter 0) of the same program P. □

We are ready to state the per-thread security preservation result formally. Given an input `While` command that satisfies $\mathsf{com\text{-}secure}^{\mathsf{no\text{-}high\text{-}branching}}_{\mathsf{no\text{-}locks\text{-}held}}$ with mds_0 initially, it promises that the `RISC` program output by the wr-compiler is $\mathsf{com\text{-}secure}^{\mathsf{pc\text{-}security}}_{\mathsf{no\text{-}locks\text{-}held}}$ with mds_0:

Theorem 5.24 (Preservation of per-thread confidentiality by the wr-compiler)**.**

$$\frac{\mathsf{com\text{-}secure}^{\mathsf{no\text{-}high\text{-}branching}}_{\mathsf{no\text{-}locks\text{-}held}}\ (c, \mathsf{mds}_0) \qquad (PCs, l', nl', C', \mathsf{False}) = \mathsf{compile\text{-}cmd}\ \mathsf{C}_0\ l\ nl\ c \qquad \mathsf{compile\text{-}cmd\text{-}input\text{-}reqs}\ \mathsf{C}_0\ l\ nl\ c}{\mathsf{com\text{-}secure}^{\mathsf{pc\text{-}security}}_{\mathsf{no\text{-}locks\text{-}held}}\ (((0, \mathsf{map\ fst}\ PCs), regs), \mathsf{mds}_0)}$$

Proof. We are given by $\mathsf{com\text{-}secure}^{\mathsf{no\text{-}high\text{-}branching}}_{\mathsf{no\text{-}locks\text{-}held}}$ (Definition 2.7) that for low-equal starting configurations (modulo modes) of c with no locks held, there exists some witness $\mathscr{B}$ satisfying both strong-low-bisim-mm and no-high-branching.

From this and Lemma 5.23 we have that the output program's corresponding execution is captured by a `RISC` semantics-level relation $\mathscr{B}_{\mathsf{C}}\mathsf{of}\ \mathscr{B}\ \mathscr{R}_{\mathsf{wr}}\ \mathscr{I}_{\mathsf{wr}}$ derived from this $\mathscr{B}$, with Lemma 5.21 discharging the compiled-cmd-config-consistent requirements.

Corollary 5.16 then gives us that $\mathsf{secure\text{-}refinement}\ \mathscr{B}\ \mathscr{R}_{\mathsf{wr}}\ \mathscr{I}_{\mathsf{wr}}$ holds, and from this and strong-low-bisim-mm $\mathscr{B}$ using Theorem 2.17 we have $\mathsf{strong\text{-}low\text{-}bisim\text{-}mm}\ (\mathscr{B}_{\mathsf{C}}\mathsf{of}\ \mathscr{B}\ \mathscr{R}_{\mathsf{wr}}\ \mathscr{I}_{\mathsf{wr}})$. This is enough to show $\mathsf{com\text{-}secure}^{\mathsf{pc\text{-}security}}_{\mathsf{no\text{-}locks\text{-}held}}$ for the `RISC` program, by Definition 2.7; as Section 5.3.3 noted, pc-security (Definition 5.6) is immediate from the definition of $\mathscr{I}_{\mathsf{wr}}$. □

To prove a whole-system security result at the `RISC` level for the compiled program, we must also prove sound-mode-use (Definition 2.10). To that end, we prove a local and global result for `RISC` programs output by the wr-compiler when given a secure `While` program. The former follows from the local compliance result in the preceding section:

Lemma 5.25 (Threads compiled by the wr-compiler obey local compliance)**.**

$$\frac{(PCs, l', nl', C', \mathsf{False}) = \mathsf{compile\text{-}cmd}\ \mathsf{C}_0\ l\ nl\ c \qquad \mathsf{compile\text{-}cmd\text{-}input\text{-}reqs}\ \mathsf{C}_0\ l\ nl\ c \qquad \mathsf{no\text{-}locks\text{-}held}\ mem}{\mathsf{local\text{-}mode\text{-}compliance}\ \langle(((0, \mathsf{map\ fst}\ PCs), regs), \mathsf{mds}_0, mem\rangle_{\mathsf{r}}}$$

Proof. We use Theorem 5.17 and Lemma 5.21 to obtain membership in $\mathscr{R}_{\mathsf{wr}}$, which then allows us to use Lemma 5.19. □

Then we prove invariance of global modes compatibility (as in Section 3.3) for compiled `RISC` programs, due to `RISC`'s identical semantics to `While` regarding locking and modes:

Lemma 5.26 (Initialising `RISC` with $\mathsf{no\text{-}locks\text{-}held}, \mathsf{mds}_0$ ensures global compatibility)**.**

$$\frac{\mathsf{no\text{-}locks\text{-}held}\ mem \qquad \forall(((pc, P), regs), mds) \in \mathsf{set}\ cms_r.\ mds = \mathsf{mds}_0}{\mathsf{global\text{-}modes\text{-}compatibility}\ (cms_r, mem)}$$

Proof. We firstly prove versions of Lemma 3.15, Lemma 3.16, and Theorem 3.17 for `RISC`, following exactly the same reasoning as we did in Section 3.3 for `While`. This is because the `RISC` instructions **LockAcq** k and **LockRel** k are (like **lock**(k) and **unlock**(k) in `While`)

the only ones in their language that modify mode state, and their semantics regarding mode state and lock memory are identical to those of the **lock**(k) and **unlock**(k) commands. The present result then follows for the same reason that Lemma 3.20 did for `While`. □

With this result, it is now possible to invoke Theorem 2.8 to compose `RISC`-level per-thread security and mode compliance, whether they were obtained via the wr-compiler (using Theorem 5.24 and Lemma 5.25, respectively), or proved directly at `RISC` level.

We remark that, for programs wholly compiled by the wr-compiler, Theorem 5.22 can be subsumed by a whole-system preservation result that no longer demands local-mode-compliance for each thread, due to our ability to obtain it directly at `RISC` level:

Theorem 5.27 (Secure threads compiled by the wr-compiler form a secure system)**.**

$$\frac{\begin{array}{l}\forall i < \mathsf{length}\ cms_r.\ \exists c\ l\ nl\ PCs\ l'\ nl'\ C'\ regs.\\ \quad \mathsf{com\text{-}secure}^{\mathsf{no\text{-}high\text{-}branching}}_{\mathsf{no\text{-}locks\text{-}held}}\ (c, \mathsf{mds}_0) \wedge\\ \quad (PCs, l', nl', C', \mathsf{False}) = \mathsf{compile\text{-}cmd}\ \mathsf{C}_0\ l\ nl\ c\ \wedge\ \mathsf{compile\text{-}cmd\text{-}input\text{-}reqs}\ \mathsf{C}_0\ l\ nl\ c\ \wedge\\ \quad cms_r[i] = (((0, \mathsf{map\ fst}\ PCs), regs), \mathsf{mds}_0)\end{array}}{\mathsf{sys\text{-}secure}_{\mathsf{no\text{-}locks\text{-}held}}\ cms_r}$$

Proof. By Theorem 2.8 and unfolding Definition 2.10, we are required to prove security and local mode compliance for every thread of the compiled `RISC` program, and global modes compatibility between them all as a whole, assuming no-locks-held and using mds_0 initially. These requirements are immediate using Theorem 5.24, Lemma 5.25, and Lemma 5.26. □

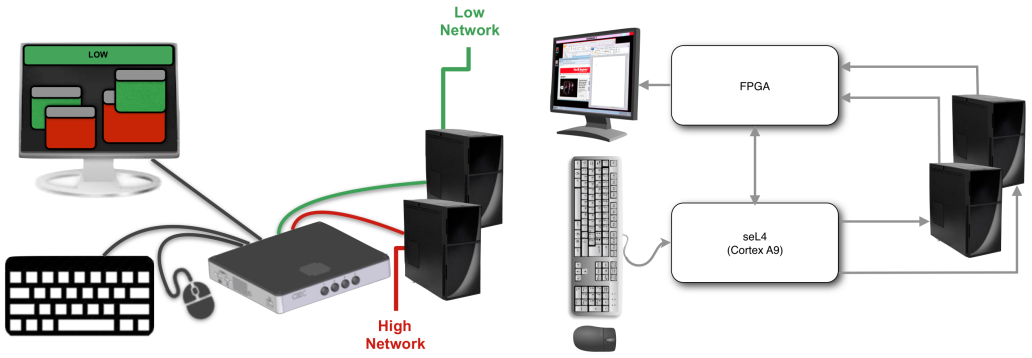
6 Case study: Cross Domain Desktop Compositor input handler

This section presents—as the main case study for the COVERN wr-compiler—a mixed-sensitivity concurrent program whose source-level noninterference properties are preserved by verified secure compilation down to an assembly-level model.

The Cross Domain Desktop Compositor (CDDC) of Beaumont *et al.* (2016) is a desktop device that gives trusted users the option of replacing multiple monitors, keyboards, and mice with a single multi-level secure user interface (via a single monitor, keyboard, and mouse, as depicted in Figure 5a) when using several desktop computers simultaneously.

Here we present as case study a program (replacing customised hardware) that handles the incoming mouse and keyboard inputs to the CDDC. This program has served as a particularly good case study, because it features both of the characteristics for which proving information-flow security is this work's main focus:

- Concurrency—here, between *software components* whose execution is interleaved (by the seL4 operating-system microkernel (Klein *et al.*, 2014)), and that interact via shared memory.
- Mixed-sensitivity reuse—here, of system resources (notably the input devices) and memory locations, for input whose sensitivity level can be different at different times.



(a) CDDC hardware use-case setup.

The bar painted at the top of the screen indicates the computer set to receive all keyboard events. Mouse events are delivered to the owner of the topmost window underneath the mouse cursor.

(b) CDDC hardware architecture.

The HID switch—implemented in software on top of seL4—runs on an ARM Cortex A9 core, and operates a compositor device implemented (as in Beaumont *et al.* (2016)) using an FPGA.

Figure 5: Functional schematics for Cross Domain Desktop Compositor hardware. Reproduced from Murray *et al.* (2018).

By exercising the COVERN wr-compiler on a `While` model of this case study, we show this compiler verification-based approach to be feasible for obtaining the preservation of noninterference properties proved at `While` level, straightforwardly and for little extra effort, down to a `RISC` model of the program.

The section will proceed as follows. Following an overview in Section 6.1 of the main characteristics of the case study, Section 6.2 presents the formal security properties proved about its `While` model—as our focus is its compilation, further details on this model and the proof techniques used to prove these properties at `While` level are left to Sison (2020). Section 6.3 then presents the formal preservation of security properties down to a `RISC` model, obtained from running the verified wr-compiler of Section 5 on the `While` model.

6.1 Overview of the case study

The case study is a software implementation of the *human interface device (HID) switch* in the CDDC (see Figure 5b). In short, this part of the CDDC is responsible for determining the destination of all *HID input* (*keyboard* and *mouse* device) events, and ensuring that the user remains informed of that destination (by operating a *video compositor* device, which renders display elements for that purpose on a shared monitor, as depicted in Figure 5a).

6.1.1 Information-flow security

The HID switch's responsibilities are security critical, as the CDDC is intended to provide an interface to multiple desktop computers belonging to different security domains; hence, the user of the CDDC is expected to choose the sensitivity of the data they input, based on the computer to which they expect it to be delivered. Furthermore, part of the CDDC's functionality is to allow users to choose which computer they are interacting with, by clicking on (accordingly responsive) display elements using the mouse. Thus, the desired information-

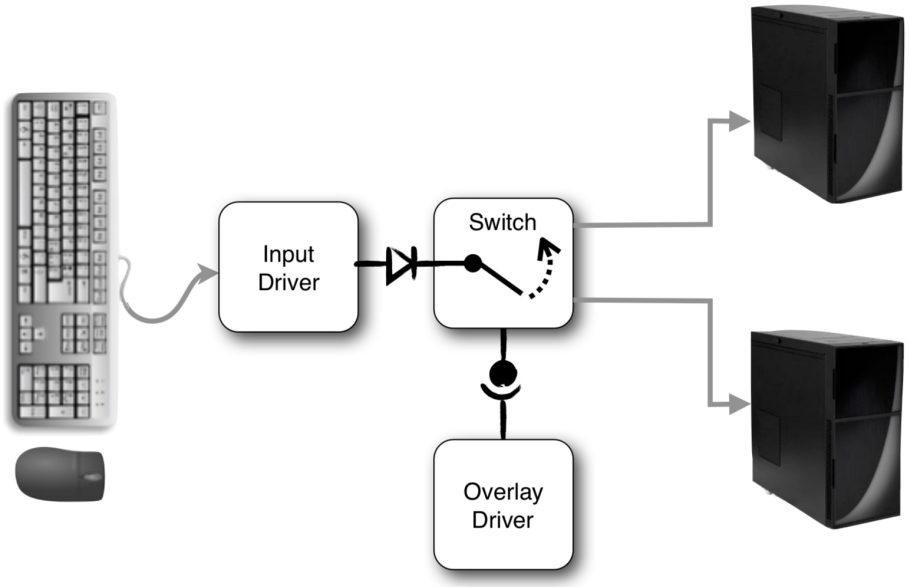


Figure 6: Functional schematic of seL4 component architecture for CDDC HID switch. Reproduced from Murray *et al.* (2018).

flow security property for the HID switch is that, in providing this functionality, it never delivers inputs to a destination contrary to the user's expectations.

We simplify analysis to the classic $\mathsf{High} \not\rightarrow \mathsf{Low}$ security policy over the basic two-point $\{\mathsf{High}, \mathsf{Low}\}$ security lattice, and model the HID switch to service only two potential destination computers.[7] One computer is designated as belonging to the High security domain, and is the only legitimate destination for High-sensitivity input events; the other is designated as belonging to the Low security domain. The hardware and connections that the SWITCH component uses to forward events to these computers are modelled as shared variables classified statically: one High, the other Low (as depicted in Figure 7b). The attacker is then considered to be an entity that can read at any time from the Low-classified one.

6.1.2 Shared-variable concurrency

The software implementation (replacing the original FPGA-based implementation (Beaumont *et al.*, 2016)) of the CDDC's HID switch is a system of software components written in C, that all run in user mode on top of the seL4 microkernel (Klein *et al.*, 2014).

Here, we have abstracted from the seL4-based C implementation's details, to model in the `While` language the basic functionality of its three main software components (as depicted in Figure 6) as a shared-variable concurrent program of three threads:

- The INPUT driver is responsible for taking events from input-device interfaces and placing them on an input-event buffer for consumption by the SWITCH (Figure 7a).

[7] Aside from presenting a more minimal case study, any verification for an arbitrary security lattice can be reduced to multiple applications of verification to the basic $\mathsf{High} \not\rightarrow \mathsf{Low}$ policy, with the locations reclassified appropriately. Furthermore, the design of the CDDC's HID switch program is symmetrical for each user.

```
lock(hid_read_atomicity_lock);
temp := hid_keyboard_available;
unlock(hid_read_atomicity_lock);
if (temp != 0) then
  lock(input_event_lock);
  input_event_data := 0;
  input_event_type := KEYBOARD;
  input_event_data :=
      ↪ hid_keyboard_source;
  unlock(input_event_lock)
else
  skip
fi
```

(a) Receipt from input device by INPUT driver. The `hid_keyboard_source` variable is value-dependently classified by the value of its sole control variable, `indicated_domain` (modelling trusted user input to the keyboard).

```
if (current_event_type = KEYBOARD) then
  if (active_domain = DOM_LOW) then
    output_event_buffer0 :=
        ↪ current_event_data
  else
    output_event_buffer1 :=
        ↪ current_event_data
  fi
else
  skip
fi
```

(b) Delivery to output device by SWITCH. The output-event buffers 0 and 1 are statically classified Low and High respectively (modelling an attacker-controlled computer that receives all data written to buffer 0).

Figure 7: Examples of external device interactions by the CDDC HID switch, as modelled in `While`—here, for the keyboard events. The full model is in the Isabelle/HOL supplement.

- The SWITCH is responsible for inspecting all input events on the buffer from the INPUT driver, querying the compositor device (as modelled in Figure 8a) and OVERLAY driver to determine if any constitute a user-directed change to the destination of subsequent events, and if so, updating the compositor device to display that change (as modelled in Figure 8b). Finally, it is responsible for delivering all events to their destination computer via the appropriate *output-device* interface (Figure 7b).
- The OVERLAY driver is responsible for servicing remote procedure calls (RPCs, made by the SWITCH) that query a subset of the compositor-device interface, regarding the position of certain mouse-clickable elements the compositor is rendering as part of a visual overlay on the trusted user's video monitor. (As no mixed-sensitivity reuse occurs in this part of the model, we leave its details to the Isabelle/HOL supplement.)

The device interfaces, shared buffers (for input events and RPC mechanisms), and local variables used by each component are all modelled as program variables in shared memory. Consequently in the `While` model, mutex locks are used to model all synchronisation and restriction of concurrent access by the components to those variables.

So that we do not need to add separate `While` semantics for interacting with private as opposed to shared memory, we model thread-private memory as shared program variables protected by a permanently held lock acquired at initialisation time (e.g. as in Figure 9a). We consider this to be a stand-in for the memory isolation properties established by the underlying operating system between the program threads that it hosts.

6.1.3 Mixed-sensitivity reuse

Inherently to the CDDC's role as a multi-level secure user interface, its HID switch receives data of differing sensitivity levels (at different times) from a single set of input device mem-

```
compositor_cursor_position :=
    ↪ current_event_data;

lock(compositor_read_atomicity_lock);
cursor_domain :=
    ↪ compositor_domain_under_cursor;
unlock(compositor_read_atomicity_lock);

if (cursor_domain = DOM_INVALID) then
  cursor_domain := active_domain
else
  skip
fi
```

(a) Querying the compositor to determine the topmost domain under the mouse cursor.

```
if (switch_state_mouse_down = 0 &&
    current_event_data = MOUSE_DOWN &&
    active_domain != cursor_domain) then
  active_domain := cursor_domain;
  lock(input_event_lock);
  input_event_data := 0;
  input_event_type := NONE;
  hid_keyboard_source := 0;
  indicated_domain := active_domain;
  unlock(input_event_lock)
else
  skip
fi
```

(b) Instructing the compositor to indicate a change to the active domain.

Figure 8: Excerpts of the SWITCH component interfacing with the compositor device.

```
/* Permanently grab this lock */
lock(switch_private_lock);
current_event_data := 0;
current_event_type := NONE;

lock(input_event_lock);
input_event_data := 0;
hid_keyboard_source := 0;
indicated_domain := active_domain;
unlock(input_event_lock)
```

(a) Initialising private variables, input-event buffer, and compositor-indicated domain, to an arbitrary initial value for `active_domain`. Zeroing the data fields prevents leaking any High-sensitivity data they might initially contain.

```
lock(input_event_lock);
if (indicated_domain = active_domain)
then
  current_event_type :=
      ↪ input_event_type;
  current_event_data :=
      ↪ input_event_data
else
  skip
fi;
unlock(input_event_lock)
```

(b) Copying from the input-event buffer to private variables. The security analysis shows that repeating the previous event is a safe course of action when the environment misbehaves by violating `indicated_domain = active_domain`.

Figure 9: Excerpts of the SWITCH component interacting with the input-event buffer.

ory locations (e.g. as modelled in Figure 7a, for the keyboard events), rather than from those of distinct device sets for each sensitivity level.

Furthermore, the HID switch propagates all input event data (regardless of sensitivity) through a single set of memory locations (the input-event buffer and SWITCH-internal copies of its contents, as modelled in Figure 9), rather than duplicating those memory locations for each security domain. Consequently in the `While` model, all of these memory locations that are subject to mixed-sensitivity reuse are assigned value-dependent classifications, reflecting the trusted user's expectation of the sensitivity level of the data they contain:

- To model a user that we trust to type sensitive information into the keyboard only when the compositor device indicates the High domain computer is *active* (i.e. set to receive all keyboard events), we have the INPUT driver draw keyboard events from a shared variable named `hid_keyboard_source` (as depicted in Figure 7a) that has classification dependent on a control variable `indicated_domain` modelling the rel-

evant state of the compositor (here, DOM_HIGH is a designated constant):

$$\begin{cases} \mathsf{High}, & \text{if } \texttt{indicated\_domain} = \texttt{DOM\_HIGH} \\ \mathsf{Low}, & \text{otherwise.} \end{cases}$$

- In contrast, as clicking on composited user interface elements has the potential ability to change the future indicated_domain (which, as a control variable, is never allowed to receive any High-sensitivity data), the model trusts the user not to encode sensitive information into the mouse input in any way. Thus, the INPUT driver always draws mouse events from a statically Low-classified shared variable.

  Consequently, as the data portion input_event_data of the input-event buffer[8] between the INPUT driver and SWITCH may carry either keyboard data of value-dependent sensitivity or Low-sensitivity mouse data, we assign it a classification dependent on the values of both its control portion and the indicated_domain:

$$\begin{cases} \mathsf{High}, & \text{if } \texttt{input\_event\_type} = \texttt{KEYBOARD} \wedge \ \texttt{indicated\_domain} = \texttt{DOM\_HIGH} \\ \mathsf{Low}, & \text{otherwise.} \end{cases}$$

- Finally, we model the seL4-based SWITCH component's copying of the event from the buffer into its own local variables, giving its data portion a classification dependent on its own private view of the currently active domain (modelled as active_domain):

$$\begin{cases} \mathsf{High}, & \text{if } \texttt{current\_event\_type} = \texttt{KEYBOARD} \wedge \ \texttt{active\_domain} = \texttt{DOM\_HIGH} \\ \mathsf{Low}, & \text{otherwise.} \end{cases}$$

  To ensure active_domain remains authoritative with what is composited by the CDDC into the display, in the While model the SWITCH initialises indicated_domain to match the initial value of active_domain (as depicted in Figure 9a), updates it whenever active_domain changes (as depicted in Figure 8b), and checks at runtime that active_domain = indicated_domain when copying data from the buffer to its own private variables (as depicted in Figure 9b).

The CVDNI properties' (1) value dependence on control variables, (2) quantification over all initial values for the control variables and (3) assumptions of environmental havoc on write-unprotected shared variables between evaluation steps (Definition 2.6) then ensure noninterference between High inputs and Low-classified sinks, regardless of the initial and dynamically changing sensitivity of all such locations subject to mixed-sensitivity reuse.

6.2 CVDNI properties of the While model to be preserved

This section will now give a brief formal exposition of the security properties of the CDDC HID switch's While-language model that our compiler will preserve down to RISC.

As the per-thread proof techniques for While that we used for the case study are outside the scope of this paper, we note only that they consist of an adaptation to mutex locks by

[8] We model in While only a single-place buffer, which could easily be extended to a buffer of arbitrary size by duplicating the same basic pattern of access, classification, and lock-protection, for multiple places.

Sison (2020) of a security type system and local mode compliance check developed by Murray *et al.* (2016b,c). Nevertheless, we have provided their full formalisation in our Isabelle/HOL supplement, and we refer the reader to these prior works on their design, and particularly to Sison (2020) for further discussion on their application to this case study.

In short, from applying local type checks on the `While`-language commands for each of the three software components (INPUT, SWITCH, and OVERLAY) to obtain per-thread security (com-secure, Definition 2.7) and modes compliance (local-mode-compliance, Definition 2.11), we have from Theorem 2.8 that the concurrent program of all three components satisfies the whole-system security property (sys-secure, Definition 2.9) as instantiated to specify that no locks are held initially.

So that we can use the approach we gave in Section 3 to obtain the global modes compatibility part of the sound-mode-use side-condition (Definition 2.10), we specify no-locks-held (Definition 3.18) as the *INIT* requirement on memory, and use the initial mode state mds_0 (Definition 3.19) for all of the components in the system.

This no-locks-held predicate and mds_0 are both defined relative to a lock interpretation parameter that we supply (as required by Section 3.1) for the CDDC model. The locks in the CDDC model fall under the following categories:

- The locks coordinating inter-component interactions grant exclusive read–write access to the shared variables they govern.
- There are also locks granting the SWITCH and INPUT components exclusive read–write access to a set of "private" variables each, for internal use. The components acquire these prior to entering their main loop, and never release them.
- Finally the model uses read-atomicity locks—a practice introduced in Section 5.1.1. These grant exclusive write access to shared variables used to model hardware interfaces, to make explicit an assumption (normally implicit in the atomicity of expression evaluation in the `While` language) that these variables will not have their value changed by the environment during a simple assignment from those variables.

  Note that these read-atomicity locks are not needed to prove confidentiality for the `While` model, but rather we add them to satisfy the requirements demanded by the wr-compiler so that it can preserve confidentiality (via small-step semantic preservation) down to the `RISC` model.

The `While`-language proof techniques we apply to each thread of the program yield $\mathsf{com\text{-}secure}^{\mathsf{no\text{-}high\text{-}branching}}$, a stronger version of the per-thread CVDNI property that enforces no-high-branching (Definition 5.5). Furthermore, we have trivially from the definition of com-secure (Definition 2.7) that if a program is secure without imposing any initial conditions, then it remains secure if we impose any *INIT* parameter arbitrarily. Therefore, for each thread we have $\mathsf{com\text{-}secure}^{\mathsf{no\text{-}high\text{-}branching}}_{\mathsf{no\text{-}locks\text{-}held}}$ (Definition 2.7, with $\mathit{INIT} \triangleq$ no-locks-held and $\mathit{EXTRA} \triangleq$ no-high-branching):

Lemmas 6.1 (Per-thread confidentiality results for CDDC `While` model)**.**

$$\mathsf{com\text{-}secure}^{\mathsf{no\text{-}high\text{-}branching}}_{\mathsf{no\text{-}locks\text{-}held}}\ (\textsc{Overlay}, \mathsf{mds}_0)$$
$$\mathsf{com\text{-}secure}^{\mathsf{no\text{-}high\text{-}branching}}_{\mathsf{no\text{-}locks\text{-}held}}\ (\textsc{Input}, \mathsf{mds}_0)$$
$$\mathsf{com\text{-}secure}^{\mathsf{no\text{-}high\text{-}branching}}_{\mathsf{no\text{-}locks\text{-}held}}\ (\textsc{Switch}, \mathsf{mds}_0)$$

From this and Theorem 2.8, using local compliance checks and Lemma 3.20 to discharge the sound-mode-use (Definition 2.10) side condition, we have a whole-system confidentiality theorem for the system of all three components running concurrently:

Theorem 6.2 (Whole-system confidentiality result for the CDDC `While` model)**.**

$$\mathsf{sys\text{-}secure}_{\mathsf{no\text{-}locks\text{-}held}}\ [(\textsc{Overlay}, \mathsf{mds}_0), (\textsc{Input}, \mathsf{mds}_0), (\textsc{Switch}, \mathsf{mds}_0)]$$

6.3 Confidentiality-preserving compilation to `RISC` model

We now turn to applying the COVERN wr-compiler of Section 5 to our `While`-language model of the CDDC's HID switch; we then have automatically that it preserves the security properties presented in Section 6.2 down to the compiler's `RISC`-language output.

The wr-compiler is *executable* in the Isabelle proof assistant. Using Isabelle's `eval` tactic, we execute the wr-compiler's main function, compile-cmd (whose implementation was described in Section 5.1) on the `While`-language models for all three of the CDDC's INPUT driver, SWITCH, and OVERLAY driver components, to obtain their `RISC`-language compilations. (Recall from Section 5.1 that we obtain the `RISC` text trivially as the map fst of the *CompRec*-annotated `RISC` program, which is the fst output of compile-cmd.)

Definition 6.3 (`RISC`-language program texts of CDDC model's components)**.**

$$\begin{aligned}
\textsc{Overlay}_{\texttt{RISC}} &\triangleq \mathsf{map\ fst\ (fst\ (compile\text{-}cmd\ C_0\ None\ 0\ \textsc{Overlay}))}\\
\textsc{Input}_{\texttt{RISC}} &\triangleq \mathsf{map\ fst\ (fst\ (compile\text{-}cmd\ C_0\ None\ 0\ \textsc{Input}))}\\
\textsc{Switch}_{\texttt{RISC}} &\triangleq \mathsf{map\ fst\ (fst\ (compile\text{-}cmd\ C_0\ None\ 0\ \textsc{Switch}))}
\end{aligned}$$

Our approach to obtain per-thread confidentiality for each of these `RISC` texts will be to use the theorem of its preservation by the wr-compiler (Theorem 5.24). Recall, this was:

Theorem 5.24 (Preservation of per-thread confidentiality by the wr-compiler)**.**

$$\frac{\begin{array}{cc}\mathsf{com\text{-}secure}^{\mathsf{no\text{-}high\text{-}branching}}_{\mathsf{no\text{-}locks\text{-}held}}\ (c, \mathsf{mds}_0) & \\ (PCs, l', nl', C', \mathsf{False}) = \mathsf{compile\text{-}cmd}\ C_0\ l\ nl\ c & \mathsf{compile\text{-}cmd\text{-}input\text{-}reqs}\ C_0\ l\ nl\ c\end{array}}{\mathsf{com\text{-}secure}^{\mathsf{pc\text{-}security}}_{\mathsf{no\text{-}locks\text{-}held}}\ (((0, \mathsf{map\ fst}\ PCs), regs), \mathsf{mds}_0)}$$

Then, for compile-cmd to execute successfully (i.e. to return *failed* = False), the model must pass the stability-checks discussed in Section 5.1. All three of OVERLAY, INPUT, and SWITCH pass the checks (1) because they use locks to protect the atomicity of reads from (otherwise unstable) variables used to model hardware interfaces, and (2) as a consequence of having passed the local security and mode compliance checks mentioned in Section 6.2.

We are now in a position to prove a whole-system confidentiality result for the compiled `RISC` model—here, with each thread's register bank initialised to zero: $\mathsf{regs}_0 \triangleq (\lambda\_\ .\ 0)$.

Theorem 6.4 (Whole-system confidentiality result for the CDDC `RISC` model)**.**

$$\begin{aligned}\mathsf{sys\text{-}secure}_{\mathsf{no\text{-}locks\text{-}held}}\ [&(((0, \textsc{Overlay}_{\texttt{RISC}}), \mathsf{regs}_0), \mathsf{mds}_0),\\ &(((0, \textsc{Input}_{\texttt{RISC}}), \mathsf{regs}_0), \mathsf{mds}_0),\\ &(((0, \textsc{Switch}_{\texttt{RISC}}), \mathsf{regs}_0), \mathsf{mds}_0)]\end{aligned}$$

Proof. A few approaches are available; we obtained formal proofs of this theorem in Isabelle/HOL using all three of the following alternatives (unfolding Definition 6.3):

Option 1. Use either of Theorem 5.22 or Theorem 5.27, both of which established whole-system security for RISC outputs of the wr-compiler when executed on $\mathsf{com\text{-}secure}^{\mathsf{no\text{-}high\text{-}branching}}_{\mathsf{no\text{-}locks\text{-}held}}$ While programs (which we have here from Lemmas 6.1). This is the easiest option to take for programs that are already verified in the While language, and then compiled successfully to RISC by the wr-compiler. It is possible to take here because all of $\text{OVERLAY}_{\texttt{RISC}}$, $\text{INPUT}_{\texttt{RISC}}$, and $\text{SWITCH}_{\texttt{RISC}}$ were obtained in this manner.

Option 2. Use Theorem 5.24 and Lemma 5.25 to obtain $\mathsf{com\text{-}secure}^{\mathsf{pc\text{-}security}}_{\mathsf{no\text{-}locks\text{-}held}}$ and local-mode-compliance (resp.) for each of $\text{OVERLAY}_{\texttt{RISC}}$, $\text{INPUT}_{\texttt{RISC}}$, and $\text{SWITCH}_{\texttt{RISC}}$, then use Theorem 2.8 directly to obtain $\mathsf{sys\text{-}secure}_{\mathsf{no\text{-}locks\text{-}held}}$. This option can be used for systems where some of the threads are written directly in RISC; for such threads, $\mathsf{com\text{-}secure}^{\mathsf{pc\text{-}security}}_{\mathsf{no\text{-}locks\text{-}held}}$ and local-mode-compliance would need to be proved directly at RISC level. However, Lemma 5.26 still discharges the global-modes-compatibility requirement for RISC, provided all threads are initialised with mds_0 and no-locks-held.

Option 3. Use Theorem 2.23 directly. This option can be used for systems where all the RISC threads are secure refinements (according to Definition 2.15) of the threads of some While program that satisfied sound-mode-use with no-locks-held initially, but some were obtained by other means than the wr-compiler (i.e. not all via the refinement $\mathscr{R}_{\mathsf{wr}}$). □

7 Related work

First in Section 7.1, we describe other recent and related works that address concerns of noninterference proof compositionality in a concurrent setting (of the kind we tackled in Section 3). The remaining sections focus on related works on verified compilation: The works in Section 7.2 and Section 7.3, like ours, focus on compilation preserving a form of noninterference. In Section 7.4 we describe our work's relationship with varieties of robust property preservation, and other compilation verification efforts in Section 7.5.

7.1 Compositionality of concurrent noninterference proofs

Alternative approaches exist to establishing the non-compositional global modes compatibility condition we proved as invariant to concurrent While executions in Section 3. For the precursor (non–value-dependent) notion of concurrent noninterference to CVDNI, Mantel *et al.* (2011) originally proposed that such a condition be met by a non-compositional *may happen in parallel* analysis (e.g. Masticola & Ryder (1993)). Then, instead of demanding the explicit declaration of the sorts of guarantees implied by locking discipline (as we do), Mantel *et al.* (2015) proposed automating their inference and proof of the compatibility condition using a reachability analysis making use of dynamic pushdown networks. We leave adapting and implementing such an approach for our CVDNI setting to future work.

We note also that, like the CVDNI theory and our work of Section 3, recent work by Frumin *et al.* (2021) concerns compositionality of machine-checked proof efforts for noninterference in a concurrent setting that are obtained potentially via a variety of proof techniques. They model more fine-grained synchronisation than we do here, via atomic compare-and-swap operations that can be used to implement mutex locking primitives. However, they

do not study compilation as a means of preserving such proofs, which is the focus of our work here. We believe that the CVDNI refinement notions we presented could support certain cases of compilation between different synchronisation primitives, provided only new thread-private state is needed (like the registers in `RISC`), and the shared variable interactions can be proved as preserved. For example, we expect mutex locking primitives (with slightly different semantics to ours here) could feasibly be refined to a compare-and-swap–based implementation in this way—this we also leave to future work.

7.2 Noninterference-preserving compilation

Tedesco *et al.* (2016) present a type-directed compilation scheme that preserves a *fault-resilient* noninterference property. The compilation scheme of our wr-compiler was inspired by theirs. Like our com-secure CVDNI security property that wr-compiler preserves, Tedesco *et al.*'s security property is also *strong bisimulation*-based (Sabelfeld & Sands, 2000). But where our property accounts (via mode states) for *controlled interference* by other threads, theirs instead quantifies over all possible interference by the environment with the memory contents. While this simplifies their task of proving that their security property is preserved under compilation—as it need not require the compiler to preserve the contents of memory—it means their security property cannot capture value-dependent noninterference. In contrast, our wr-compiler must obey our secure-refinement notion's requirement that memory contents are preserved.

The line of work most relevant to ours is that which was conducted (concurrently) by Barthe *et al.* (2020), wherein they achieved the remarkable result of proving that a modification of the CompCert C compiler (Leroy, 2009) preserves the *cryptographic constant-time* class of noninterference (2-safety) properties. Their proof approach was to use various notions of *constant-time simulation* (CT-simulation) first presented by Barthe *et al.* (2018), originally intended for application to the Jasmin compiler (Almeida *et al.*, 2017). Although not targeting programs with concurrency or mixed-sensitivity reuse (as our work does), CT-simulation shares in common with the refinement notions used by this paper that it in essence rests on a simulation diagram that is cube-shaped, as it must preserve a 2-safety hyperproperty. We submit that Barthe *et al.* (2020) broadly validates the argument we made in Sison & Murray (2019), that decomposing such cube-shaped diagrams into square-shaped ones is what will make them feasible to apply to the verification of fully fledged compilers like CompCert—noting that they described the only compilation pass they proved with their non-decomposed, cube-shaped diagram as "not especially pleasant because the diagrams are difficult to exploit" (Barthe *et al.*, 2020).

Note that the refinement theory of Barthe *et al.* (2018, 2020) preserves security via refinement phrased in terms of *forward simulation* (Leroy, 2009)—that is, each step of the abstract program must be simulated by the target program. In contrast, our theory presented here is instead targeted towards preserving refinement via *backward simulations*,[9] in which each step of the concrete (compiled) program must be simulated by the abstract program. This difference arises because in our setting we need to account for leakage that might occur

[9] Again, as commonly referred to in the compiler verification literature from Leroy (2009) onwards. This is not to be confused with the "backward simulations" of concurrency verification (Lynch & Vaandrager, 1996) and data refinement (de Roever & Engelhardt, 1998; Cavalcanti & Naumann, 2002), where the refined program instead simulates the original, and where simulation proceeds from the end of the program back to the beginning.

and be visible only in intermediate states. In their setting, in contrast, leakage that occurs in intermediate states remains visible forever in the concrete program semantics via a *leakage trace*. It remains unclear whether we could have adopted a similar approach in our work, thereby enabling a (simpler) forward simulation argument. In particular, it is not clear what the semantics of leakage traces should be for a language that supports both value-dependent classification and shared-memory concurrency as ours does.

7.3 Concurrency-compositional noninterference-preserving compilation

Neither of the above consider per-thread compositional compilation of concurrent, shared memory programs, nor value-dependent noninterference policies – the focus of our theory and compiler. Barthe *et al.* (2010, 2007a) however did aim to preserve noninterference of multithreaded programs by compilation, extending a prior *(security) type-preserving* compilation approach (Barthe *et al.*, 2004, 2007b). Their noninterference property however was termination- and timing-*insensitive*, so preventing internal timing leaks relied on the scheduler disallowing certain interleavings between threads. Also, their type-preservation argument was derived from a big-step semantics preservation property for their compiler. Here we instead rely on preservation of a small-step semantics (specifically memory contents), which is necessary for us to preserve value-dependent security under compilation, as well as to avoid imposing non-standard requirements on the scheduler.

7.4 Robust property preservation

Other recent works have improved on *fully abstract compilation* (surveyed by Patrignani *et al.* (2019)) by mapping out the spectrum (Abate *et al.*, 2019) or developing specific forms (Patrignani & Garg, 2019) of *robust property preservation*, concerned with *robustness* of source program (hyper)properties to concrete *adversarial* contexts. Like Tedesco *et al.* (2016), these works differ from ours in quantifying over a wider range of hostile interference. They also focus prominently on changes to data types, which we do not support. Thus, as a 2-safety hyperproperty quantifying over a lesser range of interference, we expect CVDNI-preservation to be implied by R2HSP (robust 2-hypersafety preservation), but do not expect it to imply any other secure compilation criterion on Abate *et al.*'s spectrum.

While recently Patrignani & Garg (2019) instantiated their *robustly safe compilation* for shared-memory fork-join concurrent programs, it only preserves (1-)safety properties. Previously however, Patrignani & Garg (2017) proved their *trace-preserving compilation* preserves k-safety hyperproperties (Clarkson & Schneider, 2010), including noninterference properties. However, it disallows the removal or addition of trace entries, which would be necessary to change the passage of time as seen in the observable trace events. Thus it excludes the sorts of changes to pacing carried out by our compiler (regulated by *abs-steps*) and studied as optimisations by the two other works (Tedesco *et al.*, 2016; Barthe *et al.*, 2020) on timing-sensitive security-preserving compilation mentioned above.

7.5 Compiler verification in general

Finally, there has been much work on large-scale verified compilation (Leroy, 2009; Kumar *et al.*, 2014) some of which has also treated compilation of shared-memory concurrent

programs (Lochbihler, 2018) including taking weak-memory consistency into account (Podkopaev *et al.*, 2019). Our work here does not consider the effects of weak-memory models. In particular, such models are often defined axiomatically rather than operationally. Our notion of secure refinement and our decomposition principle (Definitions 2.15 and 2.18, respectively) are defined assuming an operational semantics for the source and target languages.

Our work differs to prior work on verified concurrent compilation, in that it formalises and proves a compiler's ability to use information about the application's locking protocol, both to exclude unsafe access to shared variables, and conversely to know when it is safe to allow optimisations on shared variables that would typically be excluded.

8 Conclusion

To our knowledge, we have presented the first mechanised verification that a compiler preserves concurrent, value-dependent noninterference. To this end, we provided a general decomposition principle for compositional, secure refinement. Although our compiler is a proof-of-concept targeting simple source and target languages, we nevertheless applied it to produce a verified assembly-level model of an input-handling system for the CDDC (Beaumont *et al.*, 2016), a nontrivial mixed-sensitivity concurrent program.

We expect this decomposition principle to remain applicable in reducing noninterference-refinement proof efforts for compilers that overcome the specific limitations of ours here. For example, a compiler that inserts padding to equalise the time taken on either side of a High-branch—which may change when it expands expressions into multiple instructions—may instantiate the decomposition principle with a more sophisticated concrete coupling invariant that does not require pc-security.

This work serves to demonstrate that verified security-preserving compilation for mixed-sensitivity concurrent programs is now within reach, by augmenting traditional proof obligations for verified compilation (e.g. square-shaped semantics preservation) with those specific to security (e.g. absence of termination- and timing-leaks) as depicted in Figure 4. We hope that this work paves the way for future large-scale verified security-preserving compilation efforts.

Acknowledgements

We would like to thank our anonymous referees, and to thank again all those who provided feedback on the conference version of this paper (Sison & Murray, 2019) and on Robert Sison's PhD thesis (Sison, 2020). This paper describes research that was conducted during Robert's PhD candidature at UNSW Sydney and CSIRO's Data61, which was funded by an Australian Government Research Training Program (RTP) Scholarship and a CSIRO Data61 Research Project Award. We thank the Trustworthy Systems group at CSIRO's Data61 for cultivating an excellent working and learning environment.

Conflicts of Interest

None

References

Abate, C., Blanco, R., Garg, D., Hritcu, C., Patrignani, M. and Thibault, J. (2019) Journey beyond full abstraction: Exploring robust property preservation for secure compilation. *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019* pp. 256–271. IEEE.

Almeida, J. B., Barbosa, M., Barthe, G., Blot, A., Grégoire, B., Laporte, V., Oliveira, T., Pacheco, H., Schmidt, B. and Strub, P.-Y. (2017) Jasmin: High-assurance and high-speed cryptography. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17, pp. 1807–1823. ACM.

Barthe, G., Basu, A. and Rezk, T. (2004) Security types preserving compilation: (extended abstract). Steffen, B. and Levi, G. (eds), *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, Italy, January 11-13, 2004, Proceedings*. Lecture Notes in Computer Science 2937, pp. 2–15. Springer.

Barthe, G., Rezk, T., Russo, A. and Sabelfeld, A. (2007a) Security of multithreaded programs by compilation. Biskup, J. and López, J. (eds), *Computer Security - ESORICS 2007, 12th European Symposium On Research In Computer Security, Dresden, Germany, September 24-26, 2007, Proceedings*. Lecture Notes in Computer Science 4734, pp. 2–18. Springer.

Barthe, G., Rezk, T. and Basu, A. (2007b) Security types preserving compilation. *Comput. Lang. Syst. Struct.* **33**(2):35–59.

Barthe, G., Rezk, T., Russo, A. and Sabelfeld, A. (2010) Security of multithreaded programs by compilation. *ACM Trans. Inf. Syst. Secur.* **13**(3):21:1–21:32.

Barthe, G., Grégoire, B. and Laporte, V. (2018) Secure compilation of side-channel countermeasures: The case of cryptographic "constant-time". *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018* pp. 328–343. IEEE Computer Society.

Barthe, G., Blazy, S., Grégoire, B., Hutin, R., Laporte, V., Pichardie, D. and Trieu, A. (2020) Formal verification of a constant-time preserving C compiler. *Proc. ACM Program. Lang.* **4**(POPL):7:1–7:30.

Beaumont, M., McCarthy, J. and Murray, T. (2016) The cross domain desktop compositor: Using hardware-based video compositing for a multi-level secure user interface. Schwab, S., Robertson, W. K. and Balzarotti, D. (eds), *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016* pp. 533–545. ACM.

Cavalcanti, A. and Naumann, D. A. (2002) Forward simulation for data refinement of classes. Eriksson, L.-H. and Lindsay, P. A. (eds), *FME 2002:Formal Methods—Getting IT Right* pp. 471–490. Springer Berlin Heidelberg.

Clarkson, M. R. and Schneider, F. B. (2010) Hyperproperties. *J. Comput. Secur.* **18**(6):1157–1210.

de Roever, W. P. and Engelhardt, K. (1998) *Data Refinement: Model-oriented Proof Theories and their Comparison*. Cambridge Tracts in Theoretical Computer Science, vol. 46. Cambridge University Press.

Focardi, R., Gorrieri, R. and Panini, V. (1995) The security checker: a semantics-based tool for the verification of security properties. *Proceedings The Eighth IEEE Computer Security Foundations Workshop* pp. 60–69.

Frumin, D., Krebbers, R. and Birkedal, L. (2021) Compositional non-interference for fine-grained concurrent programs. *42nd IEEE Symposium on Security and Privacy (S&P'21), to appear; CoRR* **abs/1910.00905**.

Jones, C. B. (1981) *Development Methods for Computer Programs including a Notion of Interference*. D.Phil. thesis, University of Oxford.

Kaufmann, T., Pelletier, H., Vaudenay, S. and Villegas, K. (2016) When constant-time source yields variable-time binary: Exploiting curve25519-donna built with msvc 2015. *Cryptology and Network Security* pp. 573–582. Springer International Publishing.

Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R. and Heiser, G. (2014) Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems* **32**(1):2:1–2:70.

Kumar, R., Myreen, M., Norrish, M. and Owens, S. (2014) CakeML: A verified implementation of ML. Peter Sewell (ed), *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* pp. 179–191. ACM Press.

Leroy, X. (2009) A formally verified compiler back-end. *J. Autom. Reason.* **43**(4):363–446.

Lochbihler, A. (2018) Mechanising a type-safe model of multithreaded java with a verified compiler. *Journal of Automated Reasoning* **61**(1):243–332.

Lynch, N. and Vaandrager, F. (1996) Forward and backward simulations. *Inf. Comput.* **128**(1):1–25.

Mantel, H., Sands, D. and Sudbrock, H. (2011) Assumptions and guarantees for compositional noninterference. *IEEE Computer Security Foundations Symposium* pp. 218–232. IEEE.

Mantel, H., Müller-Olm, M., Perner, M. and Wenner, A. (2015) Using dynamic pushdown networks to automate a modular information-flow analysis. *25th International Symposium on Logic Based Program Synthesis and Transformation (LOPSTR)*.

Masticola, S. P. and Ryder, B. G. (1993) Non-concurrency analysis. *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP '93, pp. 129–138. ACM.

Molnar, D., Piotrowski, M., Schultz, D. and Wagner, D. (2006) The program counter security model: Automatic detection and removal of control-flow side channel attacks. *Proceedings of the 8th International Conference on Information Security and Cryptology*. ICISC'05, pp. 156–168. Springer-Verlag.

Murray, T. (2015) On high-assurance information-flow-secure programming languages. *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security* pp. 43–48.

Murray, T., Sison, R., Pierzchalski, E. and Rizkallah, C. (2016a) Compositional security-preserving refinement for concurrent imperative programs. *Archive of Formal Proofs* June. `http://isa-afp.org/entries/Dependent_SIFUM_Refinement.shtml`, Formal proof development.

Murray, T., Sison, R., Pierzchalski, E. and Rizkallah, C. (2016b) Compositional verification and refinement of concurrent value-dependent noninterference. *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016* pp. 417–431. IEEE Computer Society.

Murray, T., Sison, R., Pierzchalski, E. and Rizkallah, C. (2016c) A dependent security type system for concurrent imperative programs. *Archive of Formal Proofs* June. `http://isa-afp.org/entries/Dependent_SIFUM_Type_Systems.html`, Formal proof development.

Murray, T., Sison, R. and Engelhardt, K. (2018) COVERN: A logic for compositional verification of information flow control. *European Symposium on Security and Privacy* pp. 16–30. IEEE.

Patrignani, M. and Garg, D. (2017) Secure Compilation and Hyperproperty Preservation. *IEEE 30th Computer Security Foundations Symposium, CSF 2017, Santa Barbara, USA, August 21 - 25, 2017*. CSF'17.

Patrignani, M. and Garg, D. (2019) Robustly safe compilation. *Programming Languages and Systems* pp. 469–498. Springer International Publishing.

Patrignani, M., Ahmed, A. and Clarke, D. (2019) Formal approaches to secure compilation: A survey of fully abstract compilation and related work. *ACM Comput. Surv.* **51**(6):125:1–125:36.

Podkopaev, A., Lahav, O. and Vafeiadis, V. (2019) Bridging the gap between programming languages and hardware weak memory models. *Proc. ACM Program. Lang.* **3**(POPL):69:1–69:31.

Sabelfeld, A. and Sands, D. (2000) Probabilistic noninterference for multi-threaded programs. *Proceedings of the 13th IEEE Workshop on Computer Security Foundations*. CSFW '00, pp. 200–. IEEE Computer Society.

Sison, R. (2020) *Proving Confidentiality and Its Preservation Under Compilation for Mixed-Sensitivity Concurrent Programs*. PhD thesis, University of New South Wales, Sydney. `http://doi.org/10.26190/5fab5c0a76454`.

Sison, R. and Murray, T. (2019) Verifying That a Compiler Preserves Concurrent Value-Dependent Information-Flow Security. Harrison, J., O'Leary, J. and Tolmach, A. (eds), *10th International Conference on Interactive Theorem Proving (ITP 2019)*. Leibniz International Proceedings in Informatics (LIPIcs) 141, pp. 27:1–27:19. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Staples, M., Jeffery, R., Andronick, J., Murray, T., Klein, G. and Kolanski, R. (2014) Productivity for proof engineering. *Empirical Software Engineering and Measurement* p. 15.

Tedesco, F. D., Sands, D. and Russo, A. (2016) Fault-resilient non-interference. *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016* pp. 401–416. IEEE Computer Society.

Terauchi, T. and Aiken, A. (2005) Secure information flow as a safety problem. Hankin, C. and Siveroni, I. (eds), *Static Analysis* pp. 352–367. Springer Berlin Heidelberg.

Volpano, D. and Smith, G. (1998) Probabilistic noninterference in a concurrent language. *Proceedings. 11th IEEE Computer Security Foundations Workshop (Cat. No.98TB100238)* pp. 34–43.

A Label allocation and sequential composability

The wr-compiler fixes the label type $Lab \triangleq nat$ to allow it to ensure freshness merely by using the highest natural number reached so far on a "next label" counter (the argument *nl* in Example 5.1); it then increments the counter before passing it to subsequent calls, and outputs the next available unused label on return (the return value nl' in the example).

Relative to this scheme, we prove that two *consecutively compiled* `RISC` programs—in the sense that the relevant outputs from the first call are fed directly into the second call—only ever jump to locations within themselves (and not in the other).

Specifically, we define two `RISC` programs P_1, P_2 to be joinable if they are both:

- joinable-forward: P_1 only ever jumps to labels that are either
  - labelling an instruction in P_1 itself, or
  - the label of the very first instruction in P_2.
- joinable-backward: P_2 does not jump to any of the labels of instructions in P_1.

The lemma we prove then says that two `RISC` programs output by consecutive invocations of the wr-compiler are joinable.

Proving that the control flow of programs compiled by the wr-compiler always remains self-contained in this manner facilitates reasoning about their sequential composition.

B Register allocation scheme model

Like Tedesco *et al.* (2016) we generalise over the (user-supplied) register allocation scheme, and assume there are enough registers to service the maximum depth of expressions in the source program. We leave for future work the modelling and analysis of a compiler phase that spills register contents to memory, in order to make this assumption unnecessary.

Here we model the (user-supplied) register allocation scheme with two functions *reg_alloc* and *reg_alloc_cached* on the *register record* Φ (see Section 5.1) and the set A

of registers whose contents are needed to evaluate the current expression. To avoid loading from memory unnecessarily, the compiler may first call *reg_alloc_cached* $\Phi\ A\ v$ to identify a register that Φ records as already containing the variable v. When the compiler needs a fresh register, it will call *reg_alloc* $\Phi\ A$. Neither function is allowed to allocate a register in A, so the allocator is permitted to fail if it cannot find any suitable register. However, registers typically become available again as expression evaluation is resolved.

C Informal descriptions of cases of refinement relation $\mathscr{R}_{\mathsf{wr}}$

C.1 Base cases

- `stop`: This case relates a terminated `While` program with a terminated `RISC` program (i.e. one where the program counter is at the length of the program text).
- `skip_nop`: This case relates the `While` program **skip** with the configuration where the program counter is at the start of the `RISC` program [**Nop**].
- `assign_expr`: This case relates the expression evaluation part (for the expression e) of the `While` program $v := e$ with the corresponding part of the `RISC` program obtained by compiling it with the wr-compiler.
- `assign_store`: As for `assign_expr`, but for the very last **Store** instruction that commits the result of the expression evaluation back to shared memory variable v.

  It asserts additionally that v must be stable if lock-governed, and non-lock-governed otherwise. This prevents threads from violating the locking discipline (see Section 3.1).
- `lock_acq`: This case relates **lock**(k) with **LockAcq** k.
- `lock_rel`: This case relates **unlock**(k) with **LockRel** k.

C.2 Inductive cases

- `seq`: This case relates the `While` program $c_1\,;c_2$ with the *concatenation* $P_1 @ P_2$ of the `RISC` programs P_1 and P_2 that are respectively the outputs of successful consecutive compilation (see Appendix A) of c_1 and c_2 by the wr-compiler. It is intended for cases where the `While` (resp. `RISC`) program is currently in c_1 (resp. P_1).

  It is an inductive case of $\mathscr{R}_{\mathsf{wr}}$, in that:

  - c_1 is required to be related by $\mathscr{R}_{\mathsf{wr}}$ to the present location in P_1.
  - For all local configurations that obey the compiled-cmd-config-consistent requirements, c_2 is required to be related by $\mathscr{R}_{\mathsf{wr}}$ to the first instruction of P_2. This quantification ensures that $\mathscr{R}_{\mathsf{wr}}$ remains closed when execution progresses from the first program to the second program.

  It asserts that P_1 and P_2 are joinable (Appendix A), which is particularly relevant here to ensure that P_1 can only jump to locations within or at the end of itself (i.e. the start of P_2).

- `join`: This case relates a `While` program c with an offset $pc > \mathsf{length}\ P_1$ into a `RISC` program $P_1 @ P_2$, assuming the inductive hypothesis that c is related by $\mathscr{R}_{\mathsf{wr}}$ with the offset $pc - \mathsf{length}\ P_1$ into the `RISC` program P_2 alone.

  It is intended primarily for cases where the `While` (resp. `RISC`) program is currently in the c_2 (resp. P_2) of some consecutively compiled $c_1 ; c_2$ (resp. P_1 concatenated with P_2) but applies more broadly to allow any prepend of dead, unreachable instructions onto the front of a `RISC` program without breaking $\mathscr{R}_{\mathsf{wr}}$.

  It also asserts that P_1 and P_2 are joinable, which is important here to ensure that P_2 cannot jump back into P_1.

- `if_expr`: This case relates the expression evaluation part (for the expression e) of the `While` program **if** e **then** c_1 **else** c_2 **fi** with the corresponding part (including the conditional jump **Jz** at the end of expression evaluation) of the `RISC` program obtained by compiling it with the wr-compiler.

  It relies on both c_1 and c_2 being related by $\mathscr{R}_{\mathsf{wr}}$ to its compiled `RISC` counterparts when started with initialisation states judged valid by compiled-cmd-config-consistent.

  This case is depicted in full in Figure 10, on page 63; for comparison, Figure 11 depicts the relevant part of the compile-cmd implementation.

- `if_c1`: This case relates some `While` program c_1' reachable from c_1 with the corresponding part within the c_1 part of the `RISC` program obtained by compiling **if** e **then** c_1 **else** c_2 **fi** with the wr-compiler.

  It relies on c_1 being related by $\mathscr{R}_{\mathsf{wr}}$ to its compiled `RISC` counterpart at the appropriate program counter offset.

- `if_c2`: As for `if_c1`, but for c_2.

- `epilogue_step`: This case relates a terminated `While` program to the silent control flow steps navigating to the end of a `RISC` program from the end of the "then" and "else" branches of a compiled if-conditional.

  It works only for the "epilogue" step forms: **Jmp** and **Nop** (see Section 5.3.2).

  It is inductive in that it asserts closedness of $\mathscr{R}_{\mathsf{wr}}$ over pairwise reachability from the pair currently under consideration—the only case to do so directly.

- `while_expr`: This case relates the `While` program (**while** e **do** c **od**)'s initial intermediate step to **if** e **then** $(c\,;$ **while** e **do** c **od**$)$ **else stop fi**, and its expression evaluation part, with the expression evaluation and conditional jump of the `RISC` program that **while** e **do** c **od** was compiled to by compile-cmd.

  It relies on c being related by $\mathscr{R}_{\mathsf{wr}}$ to its compiled `RISC` counterpart when started with initialisation states judged valid by compiled-cmd-config-consistent.

- `while_inner`: This case relates some program c_I ; **while** e **do** c **od** reachable from c ; **while** e **do** c **od** to the loop body part of the `RISC` program compiled from **while** e **do** c **od**.

It relies on c_I being related by $\mathscr{R}_{\mathsf{wr}}$ to its compiled RISC counterpart at the appropriate program counter offset.

It also carries around the same reliance on c being related by $\mathscr{R}_{\mathsf{wr}}$ to its compiled RISC counterpart for all initialisation states judged valid by compiled-cmd-config-consistent.

- while_loop: This case handles epilogue steps for the inner loop body program, and the final jump back to the beginning of the While-loop.

  It requires $\mathscr{R}_{\mathsf{wr}}$ to relate the terminated While program to the end of the compiled loop body, and furthermore also carries around the same reliance on c being related by $\mathscr{R}_{\mathsf{wr}}$ to its compiled RISC counterpart for all initialisation states judged valid by compiled-cmd-config-consistent.

$$\frac{\begin{array}{c} c = \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \textbf{ fi} \qquad \mathsf{compile\text{-}cmd\text{-}input\text{-}reqs}\ C\ l\ nl\ c \\ (PCs, l', nl_2, C', \mathsf{False}) = \mathsf{compile\text{-}cmd}\ C\ l\ nl\ c \qquad (P_e, r, C_1, \mathsf{False}) = \mathsf{compile\text{-}expr}\ C\ \varnothing\ l\ e \\ (P_1, l_1, nl_1, C_2, \mathsf{False}) = \mathsf{compile\text{-}cmd}\ C_1\ \mathsf{None}\ (\mathsf{Suc}\ (\mathsf{Suc}\ nl))\ c_1 \qquad pc \leq \mathsf{length}\ P_e \\ (P_2, l_2, nl_2, C_3, \mathsf{False}) = \mathsf{compile\text{-}cmd}\ C_1\ (\mathsf{Some}\ nl)\ nl_1\ c_2 \qquad C_{pc} = (\mathsf{map\ snd}\ PCs)[pc] \\ \mathsf{compiled\text{-}cmd\text{-}config\text{-}consistent}\ C_{pc}\ regs\ mds\ mem \qquad \mathsf{regrec\text{-}stable}\ C_{pc} \\ \forall mds'\ mem'\ regs'.\ \mathsf{compiled\text{-}cmd\text{-}config\text{-}consistent}\ C_1\ regs'\ mds'\ mem' \wedge \mathsf{regrec\text{-}stable}\ C_1 \\ \longrightarrow ((\langle c_1, mds', mem'\rangle_{\mathsf{w}}, \langle((0, \mathsf{map\ fst}\ P_1), regs'), mds', mem'\rangle_{\mathsf{r}}) \in \mathscr{R}_{\mathsf{wr}} \wedge \\ (\langle c_2, mds', mem'\rangle_{\mathsf{w}}, \langle((0, \mathsf{map\ fst}\ P_2), regs'), mds', mem'\rangle_{\mathsf{r}}) \in \mathscr{R}_{\mathsf{wr}}) \end{array}}{(\langle c, mds, mem\rangle_{\mathsf{w}}, \langle((pc, \mathsf{map\ fst}\ PCs), regs), mds, mem\rangle_{\mathsf{r}}) \in \mathscr{R}_{\mathsf{wr}}}$$

Figure 10: Introduction rule for case if_expr of refinement relation $\mathscr{R}_{\mathsf{wr}}$.
This case pertains to the expression-evaluation part of an **if**-conditional compiled by compile-cmd (see Figure 11). Variables ignored are in gray.

```
compile_cmd C l nl (If e c1 c2) =
  (let (Pe, r, C1, faile) = (compile_expr C {} l e);
       (br, nl') = (nl, Suc nl); (ex, nl'') = (nl', Suc nl');
       (P1, l1, nl1, C2, fail1) = (compile_cmd C1 None nl'' c1);
       (P2, l2, nl2, C3, fail2) = (compile_cmd C1 (Some br) nl1 c2);
       (* Pre-compilation check ensures asmrec C2 = asmrec C3 *)
       C' = (regrec C2 ⊓R regrec C3, asmrec C2)
   in (Pe @ [((if Pe = [] then l else None, Jz br r), C1)] @
       P1 @ [((l1, Jmp ex), C2)] @ P2 @ [((l2, Nop'), C3)],
       Some ex, nl2, C', faile ∨ fail1 ∨ fail2))
```

Figure 11: Excerpt of wr-compiler implementation: case for **if**-conditionals.
This case of the Isabelle/HOL function compile-cmd compiles the While command **if** e **then** c_1 **else** c_2 **fi**. Here, @ denotes concatenation between two RISC program texts, and $\Phi \sqcap_R \Phi'$ denotes the subset of mappings on which the register records Φ and Φ' agree.