Verifying that a compiler preserves concurrent value-dependent information-flow security

Robert Sison
Data61, CSIRO, Australia
UNSW Sydney, Australia
Robert.Sison@data61.csiro.au

Toby Murray
University of Melbourne, Australia
toby.murray@unimelb.edu.au

Abstract

It is common to prove by reasoning over source code that programs do not leak sensitive data. But doing so leaves a gap between reasoning and reality that can only be filled by accounting for the behaviour of the compiler. This task is complicated when programs enforce *value-dependent* information-flow security properties—in which the sensitivity of values is itself value-dependent—and furthermore especially complicated when programs exploit shared-variable concurrency.

Prior work has formally defined a notion of concurrency-aware refinement for preserving value-dependent security properties. However, that notion is considerably more complex than standard refinement definitions typically applied in the verification of semantics preservation by compilers. To date it remains unclear whether it can be applied to a realistic compiler, because there exist no general decomposition principles for separating it into smaller, more familiar, proof obligations.

In this work, we provide such a decomposition principle, which we show dramatically reduces the complexity of proving secure refinement. Further, we demonstrate its applicability to secure compilation, by proving in Isabelle/HOL the preservation of value-dependent security by a proof-of-concept compiler from an imperative While language to a generic RISC-style assembly language, for programs with shared-memory concurrency mediated by locking primitives. Finally, we execute our compiler in Isabelle on a While language model of the Cross Domain Desktop Compositor, demonstrating to our knowledge the first use of a compiler verification result to carry an information-flow security property down to the assembly-level model of a non-trivial concurrent program.

2012 ACM Subject Classification Security and privacy → Logic and verification

Keywords and phrases Secure compilation, Information flow security, Concurrency, Verification

Funding *Robert Sison*: Australian Government RTP Scholarship & Data61 Research Project Award

Acknowledgements Many thanks to Carroll Morgan, Kai Engelhardt, Gerwin Klein, Christine Rizkallah, Matthew Brecknell, and Johannes Åman Pohjola, for their very helpful feedback on earlier versions of this paper.

1 Introduction

It is well known that program translations of the kind carried out by compilers can in principle break security properties like confidentiality [8, 1]. Yet source level reasoning about confidentiality remains common [14, 13, 12]. Existing verified compilers like CompCert [10] and CakeML [9] preserve semantics, but semantics preservation alone may be insufficient to preserve confidentiality, especially for shared memory concurrent programs whose threads must guard against timing leaks in order to prevent them manifesting as storage leaks [16].

Supporting secure compilation of programs that must enforce *value-dependent* security policies poses an additional challenge, because in such policies the sensitivity of a memory location can depend on the values held in other memory locations. Thus, unlike prior work

$$
\begin{aligned}
&\mathsf{coupling\text{-}inv\text{-}pres}\ \mathcal{B}\ \mathcal{R}\ \mathcal{I}\ \equiv \\
&\forall lc_{1A}\ lc_{1C}.\ (lc_{1A}, lc_{1C}) \in \mathcal{R} \longrightarrow \\
&\quad (\forall lc'_{1C}.\ lc_{1C} \rightsquigarrow_{\mathsf{C}} lc'_{1C} \longrightarrow \\
&\qquad (\exists n\ lc'_{1A}.\ lc_{1A} \rightsquigarrow^{n}_{\mathsf{A}} lc'_{1A}\ \wedge\ (lc'_{1A}, lc'_{1C}) \in \mathcal{R}\ \wedge \\
&\qquad\quad (\forall lc_{2A}\ lc_{2C}\ lc'_{2A}.\ (lc_{1A}, lc_{2A}) \in \mathcal{B}\ \wedge\ lc_{1A} =_{\mathsf{mds}} lc_{2A}\ \wedge \\
&\qquad\qquad (lc_{2A}, lc_{2C}) \in \mathcal{R}\ \wedge\ (lc_{1C}, lc_{2C}) \in \mathcal{I}\ \wedge \\
&\qquad\qquad lc_{1C} =_{\mathsf{mds}} lc_{2C}\ \wedge\ lc_{2A} \rightsquigarrow^{n}_{\mathsf{A}} lc'_{2A}\ \wedge\ lc'_{1A} =_{\mathsf{mds}} lc'_{2A} \longrightarrow \\
&\qquad\qquad\quad (\exists lc'_{2C}.\ lc_{2C} \rightsquigarrow_{\mathsf{C}} lc'_{2C}\ \wedge\ lc'_{1C} =_{\mathsf{mds}} lc'_{2C}\ \wedge \\
&\qquad\qquad\qquad (lc'_{2A}, lc'_{2C}) \in \mathcal{R}\ \wedge\ (lc'_{1C}, lc'_{2C}) \in \mathcal{I}))))
\end{aligned}
$$

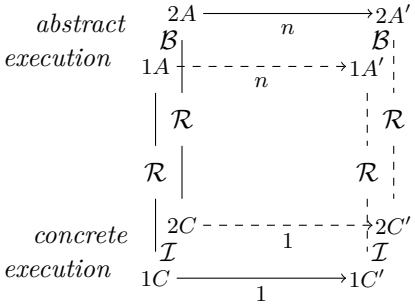


Figure 1 Definition, graphical depiction of refinement preservation for secure-refinement (Def. 6)

on secure compilation [3], preserving security under refinement requires a refinement relation that is strong enough to preserve those memory contents on which the policy depends.

In prior work [16], we presented a definition for a notion of value-dependent security-preserving refinement that is compositional for concurrent programs: by applying it to each thread individually, one can derive a secure refinement of the concurrent composition.

The essence of this notion of security-preserving refinement (presented fully in Section 2.2) is in its refinement preservation obligation (coupling-inv-pres in Figure 1). Here, the usual square-shaped commuting diagram that is commonly used to depict (semantics-preserving) refinement (Figure 2a) has been replaced by a *cube* (Figure 1). The additional dimension of this cube reflects that it preserves a 2-safety hyperproperty [5] that compares two executions rather than examining a single one. As such, it is significantly more complicated to prove than standard notions of semantics-preserving refinement typical in verified compilation [10, 9].

To date there exist no verified compilers for shared-variable concurrent programs proved to preserve value-dependent information-flow security. We argue that without a *decomposition principle* the cube-shaped refinement notion is too cumbersome to prove for realistic compilers.

In this paper, we tackle the central problem of making our notion of secure refinement applicable to verified secure compilation. Firstly, we present a decomposition principle that makes the cube-shaped notion more tractable. Secondly, we demonstrate its tractability with our major contribution: a machine-checked formal proof of concurrent value-dependent security preservation, for a proof-of-concept compiler.

In Section 3 we present our decomposition principle, which decomposes the cube (Figure 1) into three separate obligations (Figure 2). The first of these is akin to semantics-preserving refinement, while the second and third essentially ensure together that the refinement has not introduced any termination- and timing-leaks.

In Section 4 we show how the decomposition principle allows the effort to prove secure refinement for individual programs to be dramatically reduced. There, we present a side-by-side comparison of the proofs of secure refinement for a program that branches on secrets (and is thereby especially prone to introduced timing leaks), with and without the decomposition principle. We find that the decomposition reduces proof complexity by almost 50%.

In Section 5, we present our compiler and its formal verification, as an application of the decomposition principle. This compiler translates concurrent programs written in an imperative While language, with locking primitives for mediating access to shared memory, into a RISC-style assembly language. It does so by compiling each thread individually, and in doing so preserves a formal security property that remains compositional between threads. Furthermore, 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 *stable* (free from any such interference). It then leverages this tracking to avoid redundant loads from stable shared variables safely, that would otherwise be considered unsafe to omit.

All results are mechanised in Isabelle/HOL,[1] and in Section 6 we explain how, in order to validate our theory, we instantiated it so that we could execute our compiler in Isabelle. This enabled us to execute it over a While language model of the Cross Domain Desktop Compositor [4] (CDDC), a concurrent program that enforces information flow control over value-dependently classified input. To our knowledge this is the first proof of information flow security for an assembly-level model of a non-trivial concurrent program, demonstrating the power of verified secure compilation for deriving security properties of compiled code.

2 Background

To begin with, we recall relevant details of the theory presented in our previous work [16].

2.1 Concurrent value-dependent noninterference (CVDNI)

The information-flow security property CVDNI is defined in terms of two main elements:

1. a binary *strong low bisimulation modulo modes* relation $\mathcal{B}$ between program configurations, that establishes the required information-flow security property. In a manner typical of techniques for Goguen & Meseguer-style noninterference [6], it relates states that agree on their "low" portions, and demands that lock-step execution preserve that correspondence. This section will explain how it is specialised further for shared-variable concurrency.
2. a *classification* function $\mathcal{L}$ on a program configuration that determines its "low" portion, thus affecting $\mathcal{B}$'s requirements. Atypically however, $\mathcal{L}$ here can be *value-dependent* and depend on the program configuration itself – i.e. it is dynamic rather than static.

The definitions now presented are adaptations from [16] Section III-2b, with simplifications for presentation as noted. 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 denoted by a triple $\langle tps, mds, mem\rangle$:

- *tps* is the *thread-private state*, including the program text, current program position, and any statically allocated memory or storage space considered permanently thread-private by both of the attacker-model and the concurrency-model.
- $mds \in Mode \Rightarrow Var\ set$ is the *(access) mode state*, which is ghost state associating each of $Mode = \{\mathbf{AsmNoW}, \mathbf{AsmNoRW}, \mathbf{GuarNoW}, \mathbf{GuarNoRW}\}$ with (a set of) shared variables. Most pertinent to this paper are the **Asm** modes, which track assumptions made by the thread over time, about which variables will not be modified (**NoW**) or neither modified nor read (**NoRW**) by other threads. (The **Guar** modes specify corresponding guarantees that the thread makes to other threads.) For instance, if a variable $x \in Var$ is present in *mds* **AsmNoRW**, then the thread is assuming that no other threads will read nor modify x. Mode states facilitate compositional, assume-guarantee [7] style reasoning.
- $mem \in Var \Rightarrow Val$ is *shared memory* considered potentially accessible to all the threads, in the form of a total map from shared variable names to their values.

[1] The Isabelle/HOL theories are available at `https://covern.org`. The wr-compiler totals ∼7.6k lines of Isabelle proof script, and the verification + compilation of the 2-thread CDDC model totals ∼1.8k lines.

The theory is then further parameterised by the value-dependent classification function $\mathcal{L} :: (Var \Rightarrow Val) \Rightarrow Var \Rightarrow \{\mathsf{High}, \mathsf{Low}\}$, where $\mathcal{L}_{mem}\, x = \mathsf{Low}$ indicates that the value of variable x is considered observable to the attacker when the memory is mem. This allows the instantiator of the theory to specify an attacker model that can make observations of a subset of the shared memory whose scope changes dynamically over the lifetime of the program. The set $\mathcal{C} = \{y \mid \exists x.\ y \in \mathcal{C}\mathsf{vars}\ x\}$ (where $\forall y \in \mathcal{C}\mathsf{vars}\ x.\ mem_1\ y = mem_2\ y \implies \mathcal{L}_{mem_1}\, x = \mathcal{L}_{mem_2}\, x$) contains all *control variables* $y \in Var$ on whose values the classification $\mathcal{L}$ of any variable $x \in Var$ are dependent. Consequently, when two memories mem_1 and mem_2 agree on the values of all control variables in $\mathcal{C}$, we have that $\mathcal{L}_{mem_1}\, x = \mathcal{L}_{mem_2}\, x$ for all variables $x \in Var$.

The notion of observational indistinguishability used for the noninterference property is then defined over memories as follows. To support compositionality for concurrent programs, observability is relative not only to the attacker, but also to the other threads in the system, as designated by x being assumed to be *readable*: $\mathsf{readable}\ mds\ x \equiv x \notin mds$ **AsmNoRW**.

▶ **Definition 1** (Low-equivalent memories modulo modes)**.**

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

The guard $x \in \mathcal{C}$ reflects that control variables are always considered observable, regardless of whether or not they are assumed to be readable. For the rest of this paper, we will use $=^{\mathsf{Low}}_{\mathsf{mds}}$ to lift $=^{\mathsf{Low}}_{mds}$ to local program configurations, asserting also that both local configurations have the same mode state. Additionally, we will introduce the notation $lc_1 =_{\mathsf{mds}} lc_2$ to denote when local program configurations lc_1 and lc_2 are *modes-equal* (have the same mode state).

The per-thread compositional security property $\mathsf{com\text{-}secure}$ asserts the existence of a witness relation $\mathcal{B}$ for every possible observationally equivalent pair of starting configurations:

▶ **Definition 2** (Per-thread compositional CVDNI property)**.**

$$\mathsf{com\text{-}secure}\ (tps, mds) \ \equiv$$
$$\forall mem_1\ mem_2.\ mem_1 =^{\mathsf{Low}}_{mds} mem_2 \longrightarrow$$
$$\quad (\exists \mathcal{B}.\ \mathsf{strong\text{-}low\text{-}bisim\text{-}mm}\ \mathcal{B} \ \wedge\ (\langle tps, mds, mem_1\rangle, \langle tps, mds, mem_2\rangle) \in \mathcal{B})$$

where all such witness relations $\mathcal{B}$ must be a *strong low bisimulation modulo modes*:

$$\mathsf{strong\text{-}low\text{-}bisim\text{-}mm}\ \mathcal{B} \ \equiv\ \mathsf{cg\text{-}consistent}\ \mathcal{B} \ \wedge\ \mathsf{sym}\ \mathcal{B} \ \wedge$$
$$(\forall lc_1\ lc_2.\ (lc_1, lc_2) \in \mathcal{B} \ \wedge\ lc_1 =_{\mathsf{mds}} lc_2 \longrightarrow\ lc_1 =^{\mathsf{Low}}_{\mathsf{mds}} lc_2 \ \wedge$$
$$\quad (\forall lc_1'.\ lc_1 \rightsquigarrow lc_1' \longrightarrow\ (\exists lc_2'.\ lc_2 \rightsquigarrow lc_2' \ \wedge\ lc_1' =_{\mathsf{mds}} lc_2' \ \wedge\ (lc_1', lc_2') \in \mathcal{B})))$$

That is, $\mathcal{B}$ must maintain observational indistinguishability by requiring that all configuration pairs it relates that have the same mode state, are low-equivalent modulo modes.

Furthermore, it must be a *bisimulation* by being symmetric and *progressing to itself*: any step taken by one of the configurations must be able to be matched by a step taken by the configuration related to it, such that the destinations remain related by $\mathcal{B}$ (and modes-equal).

Finally—and the most crucial element ensuring the property's compositionality for concurrent programs—is the condition that $\mathcal{B}$ must be $\mathsf{cg\text{-}consistent}$: *closed under globally consistent changes* made to memory by other threads, which is to say, changes that preserve low-equivalence and are permitted by the current mode state mds. Specifically, the environment (of other threads) is permitted to change either of variable x's value or its classification only when x is *writable*: $\mathsf{writable}\ mds\ x \equiv x \notin mds$ **AsmNoW** $\wedge\ x \notin mds$ **AsmNoRW**.

▶ **Definition 3** (Closedness under globally consistent changes).

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

Theorem 3.1 of our prior work [16] then gives us that the parallel composition of com-secure programs is itself a program that enforces a system-wide value-dependent noninterference property (sys-secure, for whose details we refer the reader to Section III-2(a) of [16]).

2.2 CVDNI-preserving refinement

The proof of CVDNI-preserving refinement for a thread of a concurrent program relies on two binary relations to be nominated by the instantiator of the theory:

1. a *refinement relation* $\mathcal{R}$ 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 efforts.
2. a *coupling invariant* $\mathcal{I}$ that serves as a means of discarding unwanted pairs of local configurations of the concrete program *after the refinement*, when deriving a strong low bisimulation modulo modes for the concrete program from a given $\mathcal{B}$ and $\mathcal{R}$.

The definitions in the remainder of this section are adaptations of those from Murray et al. [16] Section V. For better readability, we present a simplified version of the requirements, in which no new shared variables are added by the refinement. Consequently we will introduce 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 language, or of differing languages.

The essence of the proof technique is to require that a number of conditions—analogous to those for strong-low-bisim-mm—be imposed on the nominated $\mathcal{R}$ and $\mathcal{I}$ in relation to a given witness relation $\mathcal{B}$ establishing CVDNI for the abstract program.

Regarding the closedness under changes by other threads that ensures compositionality for concurrency, here there are two requirements: closed-others (Definition 4) pertaining to $\mathcal{R}$, and cg-consistent (Definition 3 from Section 2.1) pertaining to $\mathcal{I}$.

▶ **Definition 4** (Closedness of refinements under changes by others).

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

Note that closed-others could be viewed as a simplification of cg-consistent considering only environmental actions that affect the memories on both sides of the relation identically, and that furthermore ensure equality of *all* shared variables, not just those judged observable.

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

▶ **Definition 5** (Preservation of modes and memory).

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

In the general case, the final major requirement for CVDNI-preservation is to prove $\mathcal{R}$ closed under the pairwise executions of the concrete and abstract programs with the aforementioned cube-shaped diagram (coupling-inv-pres, Figure 1) whose edges are given by relations in $\mathcal{B}$, $\mathcal{R}$, and $\mathcal{I}$. All then that remains is for the nominated coupling invariant $\mathcal{I}$ to be symmetric, and the predicate secure-refinement puts together all the requirements:

▶ **Definition 6** (Requirements for secure refinement of the per-thread CVDNI property).

$$\text{secure-refinement}\ \mathcal{B}\ \mathcal{R}\ \mathcal{I} \equiv$$
$$\text{preserves-modes-mem}\ \mathcal{R} \wedge \text{closed-others}\ \mathcal{R} \wedge \text{cg-consistent}\ \mathcal{I} \wedge \text{sym}\ \mathcal{I} \wedge \text{coupling-inv-pres}\ \mathcal{B}\ \mathcal{R}\ \mathcal{I}$$

Theorem 5.1 of our prior work [16] gives us that under the aforementioned conditions,

$$\mathcal{B}_{\mathsf{C}}\text{of}\ \mathcal{B}\ \mathcal{R}\ \mathcal{I} \equiv \{(lc_{1C}, lc_{2C}) \mid \exists lc_{1A}\ lc_{2A}.\ (lc_{1A}, lc_{1C}) \in \mathcal{R} \wedge (lc_{2A}, lc_{2C}) \in \mathcal{R} \wedge (lc_{1A}, lc_{2A}) \in \mathcal{B} \wedge lc_{1C} =^{\text{Low}}_{\text{mds}} lc_{2C} \wedge (lc_{1C}, lc_{2C}) \in \mathcal{I}\}$$

is a witness strong-low-bisim-mm for the concrete program:

$$\text{strong-low-bisim-mm}\ \mathcal{B} \wedge \text{secure-refinement}\ \mathcal{B}\ \mathcal{R}\ \mathcal{I} \implies \text{strong-low-bisim-mm}\ (\mathcal{B}_{\mathsf{C}}\text{of}\ \mathcal{B}\ \mathcal{R}\ \mathcal{I}).$$

3 Decomposition principle for CVDNI-preserving refinement

We now present our first contribution: an alternative way of proving secure-refinement (Definition 6) that does away with the use of the cube-shaped, two-sided refinement obligation coupling-inv-pres $\mathcal{B}\ \mathcal{R}\ \mathcal{I}$ (depicted by Figure 1), by decomposing its concerns into (1) proving $\mathcal{R}$ closed under the pairwise executions of the concrete and abstract programs alone using a square-shaped diagram (depicted by Figure 2a, which is akin to ordinary semantics-preserving refinement), and (2) a number of smaller and more separable obligations gathered together under the side-condition predicate decomp-refinement-safe.

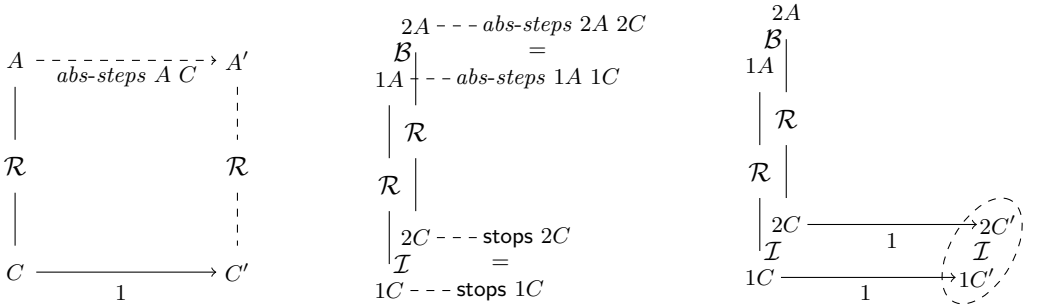
▶ **Definition 7** (Decomposed requirements for CVDNI-preserving secure refinement).

$$\text{secure-refinement-decomp}\ \mathcal{B}\ \mathcal{R}\ \mathcal{I}\ \textit{abs-steps} \equiv$$
$$\text{preserves-modes-mem}\ \mathcal{R} \wedge \text{closed-others}\ \mathcal{R} \wedge \text{cg-consistent}\ \mathcal{I} \wedge \text{sym}\ \mathcal{I} \wedge$$
$$\text{decomp-refinement-safe}\ \mathcal{B}\ \mathcal{R}\ \mathcal{I}\ \textit{abs-steps} \wedge (\forall lc_A\ lc_C.\ (lc_A, lc_C) \in \mathcal{R} \longrightarrow (\forall lc'_C.\ lc_C \rightsquigarrow_{\mathsf{C}} lc'_C \longrightarrow (\exists lc'_A.\ lc_A \rightsquigarrow_{\mathsf{A}}^{(\textit{abs-steps}\ lc_A\ lc_C)} lc'_A \wedge (lc'_A, lc'_C) \in \mathcal{R})))$$

The decomposition requires the provision of a new refinement parameter that we will call *abs-steps* or the *pacing function*, whose role is to dictate the pace of the refinement by returning the number of abstract steps that ought to be taken for a single concrete step, for a given abstract-concrete local configuration pair related by $\mathcal{R}$. The side-conditions on all of the refinement parameters (depicted by Figures 2b, 2c) are then defined as follows:

▶ **Definition 8** (Side-conditions for CVDNI-preserving refinement decomposition).

$$\text{decomp-refinement-safe}\ \mathcal{B}\ \mathcal{R}\ \mathcal{I}\ \textit{abs-steps} \equiv \forall lc_{1A}\ lc_{2A}\ lc_{1C}\ lc_{2C}.\ (lc_{1A}, lc_{2A}) \in \mathcal{B} \wedge lc_{1A} =_{\text{mds}} lc_{2A} \wedge (lc_{1A}, lc_{1C}) \in \mathcal{R} \wedge (lc_{2A}, lc_{2C}) \in \mathcal{R} \wedge (lc_{1C}, lc_{2C}) \in \mathcal{I} \wedge lc_{1C} =_{\text{mds}} lc_{2C} \longrightarrow \text{stops}\ lc_{1C} = \text{stops}\ lc_{2C} \wedge \textit{abs-steps}\ lc_{1A}\ lc_{1C} = \textit{abs-steps}\ lc_{2A}\ lc_{2C} \wedge (\forall lc'_{1C}\ lc'_{2C}.\ lc_{1C} \rightsquigarrow_{\mathsf{C}} lc'_{1C} \wedge lc_{2C} \rightsquigarrow_{\mathsf{C}} lc'_{2C} \longrightarrow (lc'_{1C}, lc'_{2C}) \in \mathcal{I} \wedge lc'_{1C} =_{\text{mds}} lc'_{2C})$$



(a) Refinement preservation for relation $\mathcal{R}$ under program execution paced by abs-steps

(b) Consistency of pacing and stopping behaviour, to prevent timing and termination leaks

(c) Closedness of the coupling invariant relation $\mathcal{I}$ under lockstep program execution

Figure 2 Graphical depictions of refinement decomposition obligations

On the intuitive meaning of the side-conditions in Definition 8:

- stops lc_{1C} = stops lc_{2C} ensures that the refinement has not introduced any termination leaks, by asserting *consistent stopping behaviour* for $\mathcal{I}$-related concrete program configurations, which we know to be observationally indistinguishable.
- *abs-steps* lc_{1A} lc_{1C} = *abs-steps* lc_{2A} lc_{2C} ensures that the refinement has not introduced any timing leaks, by asserting *consistency of the pace of the refinement* for $\mathcal{R}$-related program configurations, which we again know to be observationally indistinguishable.
- The final $\forall$-quantified clause asserts $\mathcal{I}$'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 availability of all of $\mathcal{B}$, $\mathcal{R}$, and $\mathcal{I}$ grants the ability to leverage facts known about a particular program verification technique or compiler (as captured by $\mathcal{B}$ and $\mathcal{R}$, respectively), in the proofs of the side conditions even just on $\mathcal{I}$ alone.

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

▶ **Theorem 9** (Soundness of secure-refinement-decomp)**.**

$$\text{secure-refinement-decomp}\ \mathcal{B}\ \mathcal{R}\ \mathcal{I}\ \textit{abs-steps} \implies \text{secure-refinement}\ \mathcal{B}\ \mathcal{R}\ \mathcal{I}$$

In the interests of brevity we relegate proof sketches for all results to Appendix B, and for fuller details we refer the reader to our Isabelle/HOL formalisation.

We now devote our attention to two applications of this new decomposition principle. See Section 4 for an instantiation for the proof of CVDNI-preservation for the refinement of a program that branches on a secret. See Section 5.5 for an instantiation for the proof of CVDNI-preservation by a compiler.

4 Proof effort comparison

To demonstrate how the decomposition principle reduces proof complexity and effort, we returned to the example refinement discussed in Section V-E of our previous work [16], of a program that branches on a sensitive value. We use proof size as a proxy for proof effort,

since the former is known to be strongly linearly correlated with the latter [20]. Formalised in Isabelle/HOL as `EgHighBranchRevC.thy` [15], the proof line count for that theory stood at about 4.8K 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 7), the proof line count drops from 3.6K to approx. 2K, a 46% reduction. As would be expected, the bulk of the deletions are from the full cube-shaped refinement diagram proof (Figure 1) of secure-refinement (Definition 6) for that example refinement, the surviving parts of which just become the square-shaped refinement diagram proof (Figure 2a) of secure-refinement-decomp without much modification. The deletions are replaced by the newly added proofs of the three sub-obligations of decomp-refinement-safe (Definition 8).

5 The Covern wr-compiler

Having presented our new decomposition principle for CVDNI-preserving refinement, we now turn to our compiler, whose most notable features for formal proof of secure refinement are:

1. Its implementation tracks variable stability (Section 5.4) responsive to use of locking primitives, to know when accesses to shared variables are safe to optimise, and when register contents can be still be considered consistent with shared variable contents.
2. Its verification uses a pacing function (Section 5.5.2) and coupling invariant (Section 5.5.3) as the decomposition demands, to ensure it does not introduce timing leaks.

First, we describe its source and target languages, and parameters to the compilation.

5.1 Source language

The Covern wr-compiler—short for *While-to-RISC compiler*—takes the simple imperative language with while-looping and lock-based synchronisation targeted by the Covern program logic [14], which we will refer to as `While`, consisting of the commands *cmd*:

$$
\begin{aligned}
\mathit{exp} &::= n \mid v \mid \mathit{exp} \ \oplus \mathit{exp} \\
\mathit{cmd} &::= \mathbf{skip} \mid \mathit{cmd} \,;\, \mathit{cmd} \mid \mathbf{if}\ \mathit{exp}\ \mathbf{then}\ \mathit{cmd}\ \mathbf{else}\ \mathit{cmd} \mid \\
&\qquad \mathbf{while}\ \mathit{exp}\ \mathbf{do}\ \mathit{cmd} \mid v := \mathit{exp} \mid \\
&\qquad \mathbf{lock}(k) \mid \mathbf{unlock}(k)
\end{aligned}
$$

The language is parameterised over a type of values *Val*, and binary operators $\oplus :: \mathit{Val} \Rightarrow \mathit{Val} \Rightarrow \mathit{Val}$. Constants $n \in \mathit{Val}$; $v \in \mathit{Var}$ and $k \in \mathit{Lock}$ are (resp.) shared program- and lock-variables. The semantics of the locking primitives $\mathbf{lock}(k)$ and $\mathbf{unlock}(k)$ is informed by a locking discipline provided by the user of the theory as a parameter (see Section 5.3.2). We leave for future work adding support for pointers and arrays, which we believe will be straightforward because our assume-guarantee framework already provides the means to encode the memory footprint of a command in a way that depends on values in memory.

We assume that the underlying concurrent execution model (e.g. OS, scheduler) for the `While` language prevents threads from seeing each others' current program location, and thus (as in previous work [16, 13]) the `While` program command $c \in \mathit{cmd}$ still remaining to be executed we model as thread-private state: $\langle c, \mathit{mds}, \mathit{mem} \rangle_{\mathtt{w}}$. In contrast, all program variables $v \in \mathit{Var}$ and lock variables $k \in \mathit{Lock}$ reside in the shared memory.

5.2 Target language

The wr-compiler's target is a generic RISC-style assembly language like that of [21] except with lock-based synchronisation primitives added, which we will refer to as `RISC`:

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

The language is parameterised over the same value type Val and binary operators $\oplus$, shared program variables $v \in Var$ and shared lock variables $k \in Lock$ as the `While` language. For the time being, direct-addressing **Load** and **Store** instructions are adequate for `RISC` to implement all existing `While` features, and we expect adding indirect addressing to `RISC` to be as straightforward as adding pointer and array support to `While`.

`RISC` program texts (typically named) P are just lists of binary instructions I, each optionally associated with a label $l \in Lab$. We assume that the underlying concurrency model for the `RISC` language (e.g. OS, scheduler etc.) prevents one thread from reading the program code (instructions) of another, as well as another's registers (including the program counter). Thus, we model the distinguished program counter register pc, program text P, and register bank $regs$ as thread-private state: $\langle((pc, P), regs), mds, mem\rangle_{\mathsf{r}}$.

Apart from its adaptation in this manner to the format of our local configuration triples, its evaluation semantics follows that of the `RISC` target of [21]. This semantics assumes sequentially consistent execution.

5.3 Compilation parameters

We leave certain concerns underspecified as parameters to the theory, but rely on the user-provided parameters to fulfil a few modest requirements.

5.3.1 Register allocation scheme

We generalise over the register allocation scheme, which we model by two functions reg_alloc and reg_alloc_cached on the *register record* Φ (see Section 5.4) and the set A of registers currently in use by the program. In order to avoid loading from memory unnecessarily, the compiler may first call $reg\_alloc\_cached\ \Phi\ A\ v$ to identify whether there is an unused register (i.e. not in A) that Φ records as already containing the variable v. When the compiler needs a fresh register, it will call $reg\_alloc\ \Phi\ A$ to allocate an unused register. Since neither function may return a register that is in A, the allocator is permitted to fail if it runs out of registers. Spilling of register contents to memory is not addressed here, and we leave a treatment of register spilling to future work.

5.3.2 Locking discipline

Like the Covern logic [14], we assume that the `While` language program being compiled follows a certain locking discipline, about which the compiler has knowledge, so as to ensure that the `RISC` program it produces follows the same discipline.

The user of the theory provides the necessary details of the locking discipline in the form of a *lock interpretation* parameter: $lock\text{-}interp :: Lock \Rightarrow (Var\ set \times Var\ set)$, which for each lock gives the two non-overlapping sets of program variables over which acquiring the lock grants exclusive permission to write, (resp.) read and write. These permissions are then

reflected in the semantics of the `While` and `RISC` locking primitives such that they act on the mode state to achieve the following: a thread that does not possess a lock k guarantees it does not write (resp. read or write) program variables in the fst (resp. snd) of *lock-interp* k, and conversely, a thread that possesses lock k assumes that no other threads access those program variables in that same manner.

Regarding lock interpretations and the way they interact with the user-provided value-dependent classification function $\mathcal{L}$ (see Section 2.1), we inherit a few cleanliness conditions from that earlier work [14], chief of which are that lock variables k cannot be control variables, a lock variable k governing access to a program variable v must govern the same kind of access to all of v's control variables, and $\mathcal{L}$ must classify all lock variables as Low.

Note finally that not all program variables are required to have their access governed by a lock, and that in fact the choice should be made carefully with respect to the intended security outcomes. Intuitively, the user of the theory should typically model the permanent untrusted output sinks of the whole concurrent program as program variables in shared memory for which $\mathcal{L}$ *always returns* Low. Furthermore, they should ensure that access to such variables is *not governed by any lock*, i.e. such variables are absent from *lock-interp*, since in general the attacker cannot be assumed to follow the locking protocol.

5.4 Compiler implementation and tracking of shared variable stability

We chose as a starting point the *fault-tolerant noninterference*-preserving compilation scheme of [21], on the basis of their preserving a property similar to CVDNI with respect to resilience to changes made by an environment. Aiming to repurpose that for concurrent compositionality, we adapt it to Isabelle, implementing it as a primitive recursive function:

$$\begin{aligned}&\textsf{compile-cmd} :: \mathit{CompRec} \Rightarrow \mathit{Lab\ option} \Rightarrow \mathit{Lab} \Rightarrow \mathit{cmd} \Rightarrow \\ &\qquad (I \times \mathit{CompRec})\ \mathit{list} \times \mathit{Lab\ option} \times \mathit{Lab} \times \mathit{CompRec} \times \mathit{bool}\end{aligned}$$

where we choose $\mathit{Lab} = \mathit{nat}$ for `RISC` instruction labels, and the *compilation record* type $\mathit{CompRec}$ is bookkeeping maintained by the compiler that we will describe further below.

A typical invocation to compile a `While` program $c \in \mathit{cmd}$ takes the form:

$$(\mathit{PCs}, l', nl', C', \mathit{failed}) = \textsf{compile-cmd}\ C\ l\ nl\ c \tag{1}$$

Here, compile-cmd takes an *initial compilation record* C, an optional *entry label* l, and the *next available label* nl, and for the benefit of the next invocation returns an optional *exit label* l' if one is used by the program just compiled, the *new next available label* nl', and a *final compilation record* C'. We leave discussion of the label allocation scheme and its impact on achieving sequential composability for the compiled `RISC` programs to Appendix B.2.

In addition to the output `RISC` program $P \in I\ \mathit{list}$ itself, a call to compile-cmd also outputs every $\mathit{CompRec}$ associated with the state of the program just before executing every instruction in P. These are returned zipped up together with P as the *CompRec-annotated `RISC` program* $\mathit{PCs} \in (I \times \mathit{CompRec})\ \mathit{list}$. ($P$ can trivially be recovered as map fst PCs.)

Finally, compile-cmd may return True for *failed* to reject the input program in various circumstances, such as propagating a failure by the register allocator (see Section 5.3.1).

In the style of the compilation scheme on which it was based [21], the wr-compiler maintains a *register record* (typically named) $\Phi \in \mathit{reg} \rightharpoonup \mathit{exp}$, i.e. a partial map of registers to expressions on shared variables. In addition to using it to compile away any unnecessary loads from variables in shared memory, we also use it to ensure that an expression calculated by `RISC` in registers is equal to the value of the expression as if it had all been calculated by

`While` in one step. This is especially important when writing the result of an expression back to shared memory, because the refinement is required to maintain all shared memory values.

Additionally, we have added to the wr-compiler the responsibility of maintaining an *assumption record*, which it uses primarily to assert and maintain the absence of data races on shared memory. Each assumption record (typically named) $\mathcal{S} \in (\mathit{Var\ set} \times \mathit{Var\ set})$ is a pair tracking the set of variables on which (resp.) **AsmNoW**, **AsmNoRW** assumptions are currently active at a given point in the program being compiled. As a secondary concern we also use it to assert that the two sides of any if-conditional branches act consistently on the mode state, and that while-loops restore the original mode state on termination.

A compilation record e.g. $C = (\Phi, \mathcal{S}) \in \mathit{CompRec}$ is then just a pair of a register record and an assumption record. For readability, we will use regrec and asmrec respectively to denote a $\mathit{CompRec}$'s fst and snd projections.

To explain how the compilation record is used to prevent data races, and to ensure consistency of expression evaluation between source and target program, firstly we must introduce the concept of *stability* of a variable v according to an assumption record $\mathcal{S}$:

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

In short, this means that the variable and all its control variables ($\mathcal{C}$vars v) are recorded as having either of **AsmNoW** or **AsmNoRW** active on them.

For register record entries to be of any help in ensuring consistency of `While` and `RISC` expression evaluation, we exclude expression evaluation on data 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.)

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

To ensure that an input `While` program maintains register record stability, we define the predicate no-unstable-exprs $c\ C$ to capture the requirement that a program c if started with a configuration consistent with compilation record C will never access a lock-protected variable without holding the relevant lock. (On the side, it also uses the assumption record to check the secondary concerns that if-conditional branches and while-loops keep consistent, resp. restore the mode state on termination.) We implement this predicate as a simple static check carried out by a primitive recursive function on the structure of `While` programs.

Together, regrec-stable and no-unstable-exprs 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, which gives us enough information to prove a lemma that compile-cmd only ever outputs stable register records. Full details of these we leave to Appendix B.

5.5 Proof of CVDNI-preserving compilation

We can now present the refinement relation $\mathcal{R}_{\mathsf{wr}}$ (Section 5.5.1), pacing function $\mathsf{abs\text{-}steps}_{\mathsf{wr}}$ (Section 5.5.2), and coupling invariant $\mathcal{I}_{\mathsf{wr}}$ (Section 5.5.3) that we use with our new decomposition principle (of Section 3) to prove that the wr-compiler preserves CVDNI (Section 5.5.4).

5.5.1 Refinement relation $\mathcal{R}_{\mathsf{wr}}$ and its invariants

For full details of $\mathcal{R}_{\mathsf{wr}}$, we refer the reader to the Isabelle formalisation. We provide an informal description of all of its cases, with emphasis on their purpose and on the invariants they maintain, in Appendix A. Here, we will focus on its most notable characteristics for the purpose of understanding why it is suitable to describe a CVDNI-preserving compilation.

To serve as an illustrative example, we present for comparison the case of the implementation of compile-cmd for the `While` program **if** e **then** c_1 **else** c_2, alongside the corresponding part of the refinement relation $\mathcal{R}_{\mathsf{wr}}$, namely the introduction rule $\mathcal{R}_{\mathsf{wr}}$`.if_expr`. Both have been adapted slightly to improve the clarity of the presentation. In the code listing, $\Phi \sqcap_R \Phi'$ denotes the subset of mappings on which Φ and Φ' agree.

Listing 1 Implementation of compile-cmd case for **if** e **then** c_1 **else** c_2

```
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))
```

The `if_expr` case of $\mathcal{R}_{\mathsf{wr}}$ relates the expression evaluation part of **if** e **then** c_1 **else** c_2 with the corresponding part (including the conditional jump **Jz** after expression evaluation) of the `RISC` program obtained by running compile-cmd on it. (Variables ignored are in gray.)

▶ **Example 10** (Introduction rule for case `if_expr` of $\mathcal{R}_{\mathsf{wr}}$).

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

This is a fairly typical case of $\mathcal{R}_{\mathsf{wr}}$ in a number of respects:

Firstly, there is a direct reference to the call to compile-cmd for the given `While` program. Secondly, various guards (compiled-cmd-config-consistent introduced below, and regrec-stable defined in Section 5.4) are asserted in order to restrict the scope of $\mathcal{R}_{\mathsf{wr}}$ only to consider wellformed local program configurations that line up with the conditions captured by the compilation record. Thirdly, the inductive references to $\mathcal{R}_{\mathsf{wr}}$ for P_1 and P_2, the branches of the conditional *that have not been reached yet*, are quantified over all configurations that obey the guards compiled-cmd-config-consistent and regrec-stable relative to C_1, the initial compilation record for each of the sub-calls to compile-cmd for those sub-programs.

The guard compiled-cmd-config-consistent mentioned above asserts that the compilation record C is consistent with the registers *regs*, memory *mem* and mode state *mds*.

compiled-cmd-config-consistent C *regs mds mem* $\equiv$
  $(\forall r\ e.\ (\mathsf{regrec}\ C)\ r = \mathsf{Some}\ e \longrightarrow regs\ r = \mathsf{ev}_{\mathsf{exp}}\ mem\ e) \wedge$
  $\mathsf{asmrec}\ C = (mds\ \mathbf{AsmNoW},\ mds\ \mathbf{AsmNoRW})$

Firstly, for all entries in register record mapping some register r to some expression e, the value held in r of the register bank *regs* must match the value of e if evaluated under memory *mem*. Secondly, the assumption record must consist exactly of the program variables the mode state *mds* says have **AsmNoW**, **AsmNoRW** on them respectively.

As we will see in Theorem 17, compiled-cmd-config-consistent also serves 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 compilation record C.

With $\mathcal{R}_{\mathsf{wr}}$ specified, we then prove the two requirements for secure-refinement-decomp that pertain to $\mathcal{R}_{\mathsf{wr}}$ alone: preserves-modes-mem (Definition 5) and closed-others (Definition 4).

▶ **Lemma 11** ($\mathcal{R}_{\mathsf{wr}}$ preserves modes and memory)**.** preserves-modes-mem $\mathcal{R}_{\mathsf{wr}}$

▶ **Lemma 12** ($\mathcal{R}_{\mathsf{wr}}$ is closed under changes by others)**.** closed-others $\mathcal{R}_{\mathsf{wr}}$

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

We now nominate an *abs-steps* function, determining the pace at which `While` programs progress in comparison to the `RISC` programs that they are compiled to by the wr-compiler.

To assist here and elsewhere, we define a primitive recursive helper leftmost-cmd that given a sequence of ;-separated `While` commands, strips all but the first: given $c_1\ ;\ c_2$ it returns leftmost-cmd c_1, and given any other `While` program c it returns c.

Our pacing function abs-steps$_{\mathsf{wr}}$ primarily looks at the form of the `RISC` program instruction about to be executed. The `RISC` instructions are divided into three categories:

- Instructions output by compile-expr: **Load**, **Op**, and **MoveK**. For those instructions, abs-steps$_{\mathsf{wr}}$ returns 1 if the leftmost-cmd of the `While` program is some **while** *exp* **do** *cmd*, and 0 otherwise – the 0 indicates the `While` program standing still while the `RISC` program takes *new* steps executing the expression evaluation. Returning 1 accounts for the semantics of **while** *exp* **do** *cmd* taking 1 step to **if** *exp* **then** (*cmd*;**while** *exp* **do** *cmd*) **else stop** prior to evaluating its conditional expression, which abs-steps$_{\mathsf{wr}}$ allows to occur concurrently with the first `RISC` step of the compiled expression itself.
- "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, abs-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 abs-steps$_{\mathsf{wr}}$ returns 1.

Having nominated abs-steps$_{\mathsf{wr}}$ and $\mathcal{R}_{\mathsf{wr}}$, we now have the parameters over which we are obliged to prove refinement preservation (Figure 2a) as demanded by secure-refinement-decomp (Definition 7). To this end, we prove firstly (elided to Appendix B) 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. Also, we must prove a correctness lemma for the expression compiler:

▶ **Lemma 13.** $(PCs, r, C', \mathsf{False}) = \mathsf{compile\text{-}expr}\ C\ A\ l\ e \implies (\mathsf{regrec}\ C')\ r = \mathsf{Some}\ e$

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

▶ **Lemma 14** ($\mathcal{R}_{\mathsf{wr}}$ is a refinement paced by $\mathsf{abs\text{-}steps}_{\mathsf{wr}}$).

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

5.5.3 Coupling invariant $\mathcal{I}_{\mathsf{wr}}$

The coupling invariant $\mathcal{I}_{\mathsf{wr}}$ is defined as follows:

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

In other words, $\mathcal{I}_{\mathsf{wr}}$ asserts that we only need compare local configurations that are at the same location $pc = pc'$ of the same `RISC` program $P = P'$. The effect of this is to assert that the wr-compiler has not introduced any *new* branching on sensitive values.

5.5.4 Successful compilations are CVDNI-preserving refinements

We specify that a relation $\mathcal{B}$ can describe only `While`-programs with no branching on sensitive (High-classified) values (if used as a witness strong-low-bisim-mm for com-secure), as follows:

$$\mathsf{no\text{-}high\text{-}branching}\ \mathcal{B} \equiv$$
$$\forall c\ c'\ mds\ mem\ mem'.\ (\langle c, mds, mem\rangle_{\mathsf{w}}, \langle c', mds, mem'\rangle_{\mathsf{w}}) \in \mathcal{B}\ \longrightarrow\ c = c'\ \wedge$$
$$(\forall e\ c_1\ c_2.\ \mathsf{leftmost\text{-}cmd}\ c = \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \longrightarrow\ \mathsf{ev}_{\mathsf{exp}}\ mem\ e = \mathsf{ev}_{\mathsf{exp}}\ mem'\ e)$$

That is, it refuses to relate configurations at different program locations, meaning (if imposed on a strong-low-bisim-mm) that a branch on a sensitive value could not have occurred at some point in the past. Furthermore if it is at a conditional branching point, the expression e determining which branch will be taken evaluates to the same boolean value for both configurations' memories. By imposing this condition on a relation that already ensures Low-equivalent memory modulo modes, this effectively disallows any branching on values that are High-classified or otherwise hidden to other threads. Then, for such programs:

▶ **Lemma 15.** $\mathsf{strong\text{-}low\text{-}bisim\text{-}mm}\ \mathcal{B}\ \wedge\ \mathsf{no\text{-}high\text{-}branching}\ \mathcal{B}\ \Longrightarrow$
$\mathsf{decomp\text{-}refinement\text{-}safe}\ \mathcal{B}\ \mathcal{R}_{\mathsf{wr}}\ \mathcal{I}_{\mathsf{wr}}\ \mathsf{abs\text{-}steps}_{\mathsf{wr}}\ \wedge$
$\mathsf{secure\text{-}refinement\text{-}decomp}\ \mathcal{B}\ \mathcal{R}_{\mathsf{wr}}\ \mathcal{I}_{\mathsf{wr}}\ \mathsf{abs\text{-}steps}_{\mathsf{wr}}$

From this it follows immediately via Theorem 9 that $\mathcal{R}_{\mathsf{wr}}$ with the help of $\mathcal{I}_{\mathsf{wr}}$ describes a CVDNI-preserving refinement for non-High-branching `While` programs:

▶ **Corollary 16** ($\mathcal{R}_{\mathsf{wr}}$ is a CVDNI-preserving refinement for non-High-branching programs).

$$\mathsf{strong\text{-}low\text{-}bisim\text{-}mm}\ \mathcal{B}\ \wedge\ \mathsf{no\text{-}high\text{-}branching}\ \mathcal{B}\ \Longrightarrow\ \mathsf{secure\text{-}refinement}\ \mathcal{B}\ \mathcal{R}_{\mathsf{wr}}\ \mathcal{I}_{\mathsf{wr}}$$

Finally, we prove that successful compilation produces a `RISC` program related by $\mathcal{R}_{\mathsf{wr}}$ to its input `While` program, when started with corresponding and reasonable initial configurations:

▶ **Theorem 17** (Successful compilations are refinements in $\mathcal{R}_{\mathsf{wr}}$).

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

6 Case study: the wr-compiler in action

To test the theory, we instantiated it and applied the wr-compiler to a `While`-language model of the Cross Domain Desktop Compositor [4] (CDDC), a non-trivial concurrent program enforcing information-flow control on input classified value-dependently on other input—this facilitates a trusted user's interaction with multiple desktop machines of differing clearance.

The CDDC model to which we applied the compiler is a 2-thread program that was a precursor to the 3-thread model that was verified using the Covern program logic [14]. Each of the `While` threads of the CDDC program (together about 150 lines of `While`) we proved satisfy the compositional security property com-secure (Definition 2), using a precursor to the Covern logic that yields CVDNI-witness bisimulations that are non-High-branching.

The resulting compiler is *executable* in Isabelle, meaning that compile-cmd can be executed on the `While` program text for each of the two threads to obtain their `RISC` compilations (together totalling about 250 `RISC` instructions) using the Isabelle tactic **eval**.

The secure compilation theorems (of Section 5.5.4), together with strong-low-bisim-mm preservation and compositionality for com-secure (Theorems 5.1, 3.1 of [16], mentioned earlier in Sections 2.2, 2.1 respectively) then allow us to derive that the compiled program is secure when its threads are run concurrently. (For full details, see the Isabelle/HOL formalisation.)

To our knowledge this is the first proof of source-level information-flow security being carried by a verified compiler to an assembly-level model of a non-trivial concurrent program.

7 Related work

A number of works, like ours, focus on compilation that preserves a noninterference property. Tedesco et al. [21] 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 [19]. However, whereas our CVDNI property accounts for *controlled interference* by other threads, via mode states, theirs instead quantifies over all possible interference by the environment with the memory contents. While doing so simplifies their task of proving that their security property is preserved under compilation—since it need not require that the compiler preserves the contents of memory—it makes their security property unable to capture notions like value-dependent noninterference, which ours can. In contrast, our wr-compiler must obey the requirement imposed by our notion of secure refinement that memory contents are preserved, and when adapting Tedesco et al.'s expression compiler, we found and fixed a bug (acknowledged in private correspondence) whereby registers in use would be incorrectly reallocated. This bug caused expressions like $v + (v + 1)$ to be compiled incorrectly to programs yielding the value of $(v + 1) + (v + 1)$ instead, causing a violation of memory contents preservation.

Barthe et al. [1] consider the problem of preserving *cryptographic constant-time policies* under compilation. These are a class of noninterference properties similar to CVDNI in its explicit consideration for capturing timing-sensitivity. Barthe et al. consider a number of common categories of compile-time optimisations, and mechanise proofs in Coq that such optimisations preserve various constant-time security properties. The scope of optimisations they consider is wider than those performed by our wr-compiler. Specifically, in our While language, all variables are shared. This severely limits the scope of optimisations to those that the compiler can perform when it knows that a shared variable is stable because it has been locked. At present, our wr-compiler avoids redundant loads during expression compilation, but other optimisations like loop hoisting and constant folding we are yet to implement. Their

proof technique, *constant-time simulation* (or *CT-simulation*) for preserving constant-time policies under refinement was developed independently to our original cube-shaped secure refinement definition [16]. Like ours, theirs is also a cube-shaped obligation and makes use of a pacing function analogous in their setting to our *abs-steps*. Unlike our work here, Barthe et al. do not give a general method for decomposing their cube-shaped simulation diagrams.

Neither of the works cited above consider per-thread compositional compilation of concurrent, shared memory programs, nor value-dependent noninterference policies, which are the focus of our theory and compiler.

Patrignani et al. [17] prove that *trace-preserving compilation* preserves k-safety hyperproperties [5], including noninterference properties. However, it disallows compilers from eliminating any entries from traces. Thus for traces that need to reflect the passage of time as observable to other threads, as needed for security properties like our CVDNI com-secure to be compositional for concurrent programs, it is too restrictive to allow any optimising compilation that changes the pace of the program. This would exclude optimisations carried out by our compiler (which permits changes to pacing regulated by *abs-steps*) and studied by the two other works on timing-sensitive security-preserving compilation mentioned above.

Other work has aimed at preserving noninterference of multithreaded programs by compilation. Specifically, Barthe et al. [3] do so by extending a prior (security) *type-preserving* compilation approach [2]. This work differs from ours in two crucial ways. Firstly, unlike CVDNI, their noninterference property is termination-insensitive and timing-insensitive. Thus, their approach for preventing internal timing leaks introduced by compilation requires that the scheduler disallows certain interleavings between threads. Secondly, their security property like many mentioned above is not value-dependent. Nevertheless, they establish their security property for a compiled program using a type-preservation argument derived from a proof of (big-step) semantics preservation for their compiler. Here we also lean on a form of semantics preservation, in our case preservation of small-step semantics (specifically memory contents). This is necessary for us to avoid imposing non-standard requirements on the scheduler, as well as to preserve value-dependent security under compilation.

Finally, there has been much recent work on large-scale verified compilation [10, 9] some of which has also treated compilation of shared-memory concurrent programs [11] including taking weak-memory consistency into account [18]. Our work here does not consider the effects of weak-memory models. However, it 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, to exclude unsafe access to shared variables, and conversely to know when it is safe to allow optimisations that would typically be excluded (see Section 5.4).

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 the CDDC [4], a non-trivial concurrent program.

This work serves to demonstrate that verified security-preserving compilation for 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 2. We hope that this work paves the way for future large-scale verified security-preserving compilation efforts.

References

1. G. Barthe, B. Grégoire, and V. Laporte. Secure compilation of side-channel countermeasures: The case of cryptographic "constant-time". In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 328–343, July 2018.
2. Gilles Barthe, Tamara Rezk, and Amitabh Basu. Security types preserving compilation. *Comput. Lang. Syst. Struct.*, 33(2):35–59, July 2007. URL: `http://dx.doi.org/10.1016/j.cl.2005.05.002`.
3. Gilles Barthe, Tamara Rezk, Alejandro Russo, and Andrei Sabelfeld. Security of multithreaded programs by compilation. *ACM Trans. Inf. Syst. Secur.*, 13(3):21:1–21:32, July 2010. URL: `http://doi.acm.org/10.1145/1805974.1805977`.
4. Mark Beaumont, Jim McCarthy, and Toby Murray. The cross domain desktop compositor: Using hardware-based video compositing for a multi-level secure user interface. In *Annual Computer Security Applications Conference (ACSAC)*, pages 533–545, 2016.
5. Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, September 2010. URL: `http://dl.acm.org/citation.cfm?id=1891823.1891830`.
6. Joseph Goguen and José Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, California, USA, April 1982. IEEE Computer Society.
7. Cliff B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. D.Phil. thesis, University of Oxford, June 1981.
8. Thierry Kaufmann, Hervé Pelletier, Serge Vaudenay, and Karine Villegas. When constant-time source yields variable-time binary: Exploiting curve25519-donna built with msvc 2015. In *Cryptology and Network Security*, pages 573–582, Cham, 2016. Springer International Publishing.
9. Ramana Kumar, Magnus Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 179–191, San Diego, January 2014. ACM Press.
10. Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4):363–446, December 2009. URL: `http://dx.doi.org/10.1007/s10817-009-9155-4`.
11. Andreas Lochbihler. Mechanising a type-safe model of multithreaded java with a verified compiler. *Journal of Automated Reasoning*, 61(1):243–332, Jun 2018. URL: `https://doi.org/10.1007/s10817-018-9452-x`.
12. Luísa Lourenço and Luís Caires. Dependent information flow types. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 317–328, Mumbai, India, January 2015. ACM.
13. Heiko Mantel, David Sands, and Henning Sudbrock. Assumptions and guarantees for compositional noninterference. In *IEEE Computer Security Foundations Symposium*, pages 218–232, Cernay-la-Ville, France, June 2011. IEEE.
14. Toby Murray, Robert Sison, and Kai Engelhardt. COVERN: A logic for compositional verification of information flow control. In *European Symposium on Security and Privacy*, London, United Kingdom, April 2018. IEEE.
15. Toby Murray, Robert Sison, Edward Pierzchalski, and Christine Rizkallah. Compositional security-preserving refinement for concurrent imperative programs. *Archive of Formal Proofs*, June 2016. `http://isa-afp.org/entries/Dependent_SIFUM_Refinement.shtml`, Formal proof development.
16. Toby Murray, Robert Sison, Edward Pierzchalski, and Christine Rizkallah. Compositional verification and refinement of concurrent value-dependent noninterference. In *IEEE Computer Security Foundations Symposium*, pages 417–431, Lisbon, Portugal, June 2016.
17. Marco Patrignani and Deepak Garg. Secure Compilation and Hyperproperty Preservation. In *IEEE 30th Computer Security Foundations Symposium, CSF 2017, Santa Barbara, USA, August 21 - 25, 2017*, CSF'17, 2017.

18 Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. Bridging the gap between programming languages and hardware weak memory models. *Proc. ACM Program. Lang.*, 3(POPL):69:1–69:31, January 2019. URL: `http://doi.acm.org/10.1145/3290382`.

19 Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings of the 13th IEEE Workshop on Computer Security Foundations*, CSFW '00, pages 200–, Washington, DC, USA, 2000. IEEE Computer Society. URL: `http://dl.acm.org/citation.cfm?id=794200.795151`.

20 Mark Staples, Ross Jeffery, June Andronick, Toby Murray, Gerwin Klein, and Rafal Kolanski. Productivity for proof engineering. In *Empirical Software Engineering and Measurement*, page 15, Turin, Italy, September 2014.

21 F. Del Tedesco, D. Sands, and A. Russo. Fault-resilient non-interference. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 401–416, June 2016.

A Informal descriptions of the cases of refinement relation $\mathcal{R}_{\mathsf{wr}}$

A.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 5.3.2).
- `lock_acq`: This case relates **lock**(k) with **LockAcq** k.
- `lock_rel`: This case relates **unlock**(k) with **LockRel** k.

A.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 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 $\mathcal{R}_{\mathsf{wr}}$, in that:
  - c_1 is required to be related by $\mathcal{R}_{\mathsf{wr}}$ 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 $\mathcal{R}_{\mathsf{wr}}$ to the first instruction of P_2. This quantification ensures that $\mathcal{R}_{\mathsf{wr}}$ remains closed when execution progresses from the first program to the second program.

  It asserts that P_1 and P_2 are joinable (Section B.2), 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 $pc >$ length P_1 into a `RISC` program $P_1@P_2$, assuming the inductive hypothesis that c is related by $\mathcal{R}_{\mathsf{wr}}$ with the offset $pc -$ 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 $\mathcal{R}_{\mathsf{wr}}$.
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 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 $\mathcal{R}_{\mathsf{wr}}$ to its compiled `RISC` counterparts when started with initialisation states judged valid by compiled-cmd-config-consistent.
- `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 with the wr-compiler.
  It relies on c_1 being related by $\mathcal{R}_{\mathsf{wr}}$ 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 epilogue step forms (see Section 5.5.2).
  It is inductive in that it asserts closedness of $\mathcal{R}_{\mathsf{wr}}$ 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)'s initial intermediate step to **if** e **then** (c ; **while** e **do** c) **else stop**, and its expression evaluation part, with the expression evaluation and conditional jump of the `RISC` program that **while** e **do** c was compiled to by compile-cmd.
  It relies on c being related by $\mathcal{R}_{\mathsf{wr}}$ 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 reachable from c ; **while** e **do** c to the loop body part of the `RISC` program compiled from **while** e **do** c.
  It relies on c_I being related by $\mathcal{R}_{\mathsf{wr}}$ to its compiled `RISC` counterpart at the appropriate program counter offset.
  It also carries around the same reliance on c being related by $\mathcal{R}_{\mathsf{wr}}$ 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 $\mathcal{R}_{\mathsf{wr}}$ 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 $\mathcal{R}_{\mathsf{wr}}$ to its compiled `RISC` counterpart for all initialisation states judged valid by compiled-cmd-config-consistent.

B More details of some definitions and proofs

B.1 Proofs about the decomposition principle

▶ **Theorem 18** (Soundness of secure-refinement-decomp).

secure-refinement-decomp $\mathcal{B}$ $\mathcal{R}$ $\mathcal{I}$ *abs-steps* $\implies$ secure-refinement $\mathcal{B}$ $\mathcal{R}$ $\mathcal{I}$

Proof. The only obligation for secure-refinement (Definition 6) not obtained immediately from secure-refinement-decomp (Definition 7) is the cube-shaped coupling-inv-pres (Figure 1).

The front face of the cube is just ordinary square-shaped refinement preservation (depicted in Figure 2a), given to us by secure-refinement-decomp. This gives us that a single concrete step from lc_{1C} is simulated by n abstract steps lc_{1A}, 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 lc_{2A} related by $\mathcal{B}$ to lc_{1A} are simulated by some concrete step from lc_{2C} related by $\mathcal{R}$ to lc_{2A} and by $\mathcal{I}$ to lc_{1C}.

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 lc_{2A}, simulating a single concrete step from lc_{2C}, could only be the very same n abstract steps from lc_{2A} that we were required to consider. This allows us to use once again the square-shaped refinement preservation (Figure 2a) given to us by secure-refinement-decomp.

Consistency of refinement pacing and stopping behaviour (depicted in Figure 2b) given by decomp-refinement-safe (Definition 8) 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 lc_{2C} to drive the matching simulation step.

Finally, the remainder of decomp-refinement-safe (depicted in Figure 2c) discharges the requirement of closedness and modes-equality maintenance of $\mathcal{I}$ under lockstep execution, demanded by the bottom face of the cube. ◀

B.2 Label allocation and sequential composability

For allocating natural numbers to use as labels for `RISC` instructions the wr-compiler ensures freshness merely by using the highest number reached so far on a "next label" counter (nl in the invocation example (1)), incrementing the counter before passing it along to subsequent calls, and outputting the next available unused label on return (as nl' in the example).

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.

We prove a lemma that says that two `RISC` programs that were compiled by the wr-compiler *consecutively*—in the sense that the relevant outputs from the first call are fed directly into the second call—are joinable.

B.3 More detail on compile-cmd-input-reqs and the wr-compiler proofs

The first two requirements to the predicate compile-cmd-input-reqs $C\ l\ nl\ c$ were given in Section 5.4. 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 Section B.2 for more information on label allocation).

▶ **Definition 19** (Requirements on inputs to compile-cmd).

$$\mathsf{compile\text{-}cmd\text{-}input\text{-}reqs}\ C\ l\ nl\ c\ \equiv\ c \neq \textbf{\textit{stop}} \wedge$$
$$(\forall x.\ l = \mathsf{Some}\ x \longrightarrow x < nl)\ \wedge\ \mathsf{no\text{-}unstable\text{-}exprs}\ c\ C\ \wedge\ \mathsf{regrec\text{-}stable}\ C$$

These input conditions give us enough information to prove that every instruction of a *CompRec*-annotated `RISC` program output by a successful run of compile-cmd is annotated by a stable register record, and that the output *CompRec*'s register record is also stable:

▶ **Lemma 20** (Successful compilations output only stable register records)**.**

$(PCs, l', nl', C', failed) =$ compile-cmd $C\ l\ nl\ c\ \wedge$ compile-cmd-input-reqs $C\ l\ nl\ c\ \wedge$
$failed =$ False $\implies (\forall pc <$ length $PCs.$ regrec-stable (map snd PCs ! $pc)) \wedge$ 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 expressions that appear in register records must be stable because they are always only ever parts of expressions on variables that were stable in the input program when read into registers, and furthermore, when compiling an **unlock**() the wr-compiler will always flush all register records making reference to any variables that the **unlock**() makes unstable. ◀

Before proceeding, we name the parts of compiled-cmd-config-consistent more explicitly:

▶ **Definition 21** (Configuration consistency requirements for compiled commands)**.**

compiled-cmd-config-consistent $C\ regs\ mds\ mem\ \equiv$
regrec-mem-consistent (regrec C) $regs\ mem\ \wedge$ asmrec-mds-consistent (asmrec C) mds

▶ **Definition 22** (Consistency between a register record, register bank, and shared memory)**.**

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

▶ **Definition 23** (Consistency between an assumption record and a mode state)**.**

asmrec-mds-consistent $\mathcal{S}\ mds\ \equiv\ \mathcal{S} = (mds\ \mathbf{AsmNoW},\ mds\ \mathbf{AsmNoRW})$

▶ **Lemma 24** ($\mathcal{R}_{\mathsf{wr}}$ preserves modes and memory)**.** preserves-modes-mem $\mathcal{R}_{\mathsf{wr}}$

Proof. By induction on the structure of $\mathcal{R}_{\mathsf{wr}}$. For all cases of $(lc_w, lc_r) \in \mathcal{R}_{\mathsf{wr}}$, $lc_w =^{\mathsf{mem}}_{\mathsf{mds}} lc_r$ is either asserted directly by the guards or obtainable from the inductive hypothesis. ◀

▶ **Lemma 25** ($\mathcal{R}_{\mathsf{wr}}$ is closed under changes by others)**.** closed-others $\mathcal{R}_{\mathsf{wr}}$

Proof. By induction on the structure of $\mathcal{R}_{\mathsf{wr}}$. Changes by others (Definition 4) 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 compile-cmd only creates regrec-stable records (referring to no writable variables). No other $\mathcal{R}_{\mathsf{wr}}$ guards mention shared memory. ◀

▶ **Lemma 26** (Successfully compiled programs maintain config consistency requirements)**.**

$(PCs, l', nl', C', failed) =$ compile-cmd $C\ l\ nl\ c\ \wedge$ compile-cmd-input-reqs $C\ l\ nl\ c\ \wedge$
$failed =$ False $\wedge\ pc <$ length $PCs\ \wedge\ P =$ map fst $PCs\ \wedge\ Cs =$ map snd $PCs\ \wedge$
compiled-cmd-config-consistent $(Cs$! $pc)\ regs\ mds\ mem\ \wedge$
$\langle((pc, P), regs), mds, mem\rangle_{\mathsf{r}} \rightsquigarrow_{\mathsf{r}} \langle((pc', P), regs'), mds', mem'\rangle_{\mathsf{r}}) \implies$
compiled-cmd-config-consistent (`if` $pc' <$ length P `then` $(Cs$! $pc')$ `else` $C')\ regs'\ mds'\ mem'$

Proof. We in fact prove it separately for regrec-mem-consistent and asmrec-mds-consistent, in both cases by induction on the structure of the `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, and so for this we must prove a sub-lemma 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, **while** e **do** c', $v := e$. Additionally, **unlock**() flushes register record entries mentioning variables that are to become unstable, and **while** e **do** c' 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**() and **unlock**(). The other cases for c do not touch the mode state. ◀

▶ **Lemma 27** (Correctness of the expression compiler).

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

Proof. By induction on the structure of expressions e, using the simplification rules for the implementation of compile-expr, and also relying on the assumptions of correctness of the register allocation scheme supplied by the instantiator of the theory (see Section 5.3.1). ◀

▶ **Lemma 28** ($\mathcal{R}_{\mathsf{wr}}$ is a refinement paced by $\mathsf{abs\text{-}steps}_{\mathsf{wr}}$).

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

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

The base case `stop` is immediate, because 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 of these cases, we use Lemma 26 to obtain the preservation of the guards demanded by the $\mathcal{R}_{\mathsf{wr}}$ 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 27) in order to ensure that once the expression evaluation result 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 27 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 $\mathcal{R}_{\mathsf{wr}}$ 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 26 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 $\mathcal{R}_{\mathsf{wr}}$ are always enough to satisfy the guards demanded by the possible destination cases. Like in the proof of Lemma 26, the trickiness mostly comes from accounting for all the possible cases of control flow (ruling out spurious destinations) that need to be considered. ◀

▶ **Lemma 29.** $\mathsf{strong\text{-}low\text{-}bisim\text{-}mm}\ \mathcal{B} \wedge \mathsf{no\text{-}high\text{-}branching}\ \mathcal{B} \implies$
$\mathsf{decomp\text{-}refinement\text{-}safe}\ \mathcal{B}\ \mathcal{R}_{\mathsf{wr}}\ \mathcal{I}_{\mathsf{wr}}\ \mathsf{abs\text{-}steps}_{\mathsf{wr}}$

Proof. Definition 8 gives us the following obligations.

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 $\mathcal{I}_{\mathsf{wr}}$ 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 $\mathsf{no\text{-}high\text{-}branching}$ and $\mathcal{I}_{\mathsf{wr}}$ forces them (resp.) to be equal for the local configurations under consideration.

For closedness of $\mathcal{I}_{\mathsf{wr}}$ under lockstep execution, the only non-straightforward cases to consider are conditional branching, and the locking primitives. For conditional branching, we use $\mathsf{no\text{-}high\text{-}branching}$ for $\mathcal{B}$ with memory preservation via $\mathcal{R}_{\mathsf{wr}}$ (Lemma 11) 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 mode state equality maintenance under lockstep execution. As all lock memory is classified Low (see Section 5.3.2), we use $\mathsf{strong\text{-}low\text{-}bisim\text{-}mm}$ for $\mathcal{B}$ with memory preservation via $\mathcal{R}_{\mathsf{wr}}$ to ensure the `RISC` configurations behave consistently. ◀

▶ **Lemma 30.** $\mathsf{strong\text{-}low\text{-}bisim\text{-}mm}\ \mathcal{B} \wedge \mathsf{no\text{-}high\text{-}branching}\ \mathcal{B} \implies$
$\mathsf{secure\text{-}refinement\text{-}decomp}\ \mathcal{B}\ \mathcal{R}_{\mathsf{wr}}\ \mathcal{I}_{\mathsf{wr}}\ \mathsf{abs\text{-}steps}_{\mathsf{wr}}$

Proof. Referring to Definition 7, the obligations pertaining only to $\mathcal{R}_{\mathsf{wr}}$ and $\mathsf{abs\text{-}steps}_{\mathsf{wr}}$ are discharged by Lemma 14, Lemma 12, and Lemma 11. Pertaining to $\mathcal{I}_{\mathsf{wr}}$: clearly $\mathcal{I}_{\mathsf{wr}}$ is symmetric, and furthermore it is $\mathsf{cg\text{-}consistent}$ (Definition 3) because the actions over which $\mathcal{I}_{\mathsf{wr}}$ must be closed modify only the shared memory, and $\mathcal{I}_{\mathsf{wr}}$ places only restrictions on the program text and current location. The final obligation is discharged by Lemma 29. ◀

▶ **Theorem 31** ($\mathsf{Successful\ compilations\ are\ refinements\ in}\ \mathcal{R}_{\mathsf{wr}}$)**.**

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

Proof. By induction on the structure of `While`. The compiler input and initial configuration conditions we impose allow us to have each of **skip**, cmd ; cmd, **if** exp **then** cmd **else** cmd, **while** exp **do** cmd, $v := exp$, **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 $\mathcal{R}_{\mathsf{wr}}$ that were designed for them respectively. ◀