

Proving Confidentiality and Its Preservation Under Compilation for Mixed-Sensitivity Concurrent Programs

Author:
Sison, Robert

Publication Date:
2020

DOI:
https://doi.org/10.26190/5fab5c0a76454

License:
https://creativecommons.org/licenses/by-nc-nd/3.0/au/
Link to license to see what you are allowed to do with this resource.

Downloaded from http://hdl.handle.net/1959.4/70452 in https://unsworks.unsw.edu.au on 2024-03-28

Proving Confidentiality and Its Preservation Under Compilation for Mixed-Sensitivity Concurrent Programs

Robert Sison

Submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy



UNSW
SYDNEY

School of Computer Science and Engineering
Faculty of Engineering

October 2020

UNSW SYDNEY | Australia's Global University

Thesis/Dissertation Sheet

Surname/Family Name	:	**Sison**
Given Name/s	:	**Robert Abella**
Abbreviation for degree as given in the University calendar	:	**PhD**
Faculty	:	**Engineering**
School	:	**Computer Science and Engineering**
Thesis Title	:	**Proving Confidentiality and Its Preservation Under Compilation for Mixed-Sensitivity Concurrent Programs**

Abstract 350 words maximum:

Here, I pose the thesis that proving noninterference and its preservation by a compiler is feasible for mixed-sensitivity concurrent programs. 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. Prior work therefore presented formal methods for proving and preserving the strictest kind of confidentiality property, noninterference, for mixed-sensitivity concurrent programs: a term I coin to describe those programs that might reuse memory shared between their threads to hold data of different sensitivity levels at different times. Although these methods addressed challenges in formalising the value-dependent coordination of such mixed-sensitivity reuse under the impact of concurrency, their practicality remained unclear: Could they be used to prove noninterference for any nontrivial mixed-sensitivity concurrent program in its entirety? Furthermore, could any compiler be verified to preserve the needed guarantees to the compiled code?

To support this claim, I prove for the first time both (1) noninterference for a nontrivial mixed-sensitivity concurrent program, modelling a real-world use case, and (2) its preservation by a compiler down to an assembly-level model. This main result rests on two major contributions. First, I demonstrate how programming-language designers can make reasoning on each thread sufficient to prove noninterference for such programs, by supplying synchronisation primitives (here, mutex locks for a generic imperative language) and proving they maintain as invariant the necessary requirements. Second, I demonstrate how compiler developers can make confidentiality-preserving refinement a feasible target for verification, by using a decomposition principle to prove that a compiler (here, from that imperative language to a generic RISC-style assembly language) establishes it for mixed-sensitivity concurrent programs. Thus, per-thread reasoning proves noninterference for the case study, and the verified compiler preserves it to assembly automatically. All my results are formalised and proved in the Isabelle/HOL interactive proof assistant.

My work paves the way for more fully featured programming languages and their compilers, in replicating these results, to raise the typical level of assurance readily offered by developers of multithreaded software responsible for data of multiple sensitivity levels.

Declaration relating to disposition of project thesis/dissertation

I hereby grant to the University of New South Wales or its agents a non-exclusive licence to archive and to make available (including to members of the public) my thesis or dissertation in whole or in part in the University libraries in all forms of media, now or here after known. I acknowledge that I retain all intellectual property rights which subsist in my thesis or dissertation, such as copyright and patent rights, subject to applicable law. I also retain the right to use all or part of my thesis or dissertation in future works (such as articles or books).

…… Signature

…… Date

The University recognises that there may be exceptional circumstances requiring restrictions on copying or conditions on use. Requests for restriction for a period of up to 2 years can be made when submitting the final copies of your thesis to the UNSW Library. Requests for a longer period of restriction may be considered in exceptional circumstances and require the approval of the Dean of Graduate Research.

ORIGINALITY STATEMENT

'I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.'

Signed …….....

Date …….....

COPYRIGHT STATEMENT

'I hereby grant the University of New South Wales or its agents a non-exclusive licence to archive and to make available (including to members of the public) my thesis or dissertation in whole or part in the University libraries in all forms of media, now or here after known. I acknowledge that I retain all intellectual property rights which subsist in my thesis or dissertation, such as copyright and patent rights, subject to applicable law. I also retain the right to use all or part of my thesis or dissertation in future works (such as articles or books).'

'For any substantial portions of copyright material used in this thesis, written permission for use has been obtained, or the copyright material is removed from the final public version of the thesis.'

Signed …….....

Date …….....

AUTHENTICITY STATEMENT

'I certify that the Library deposit digital copy is a direct equivalent of the final officially approved version of my thesis.'

Signed …….....

Date …….....

INCLUSION OF PUBLICATIONS STATEMENT

UNSW is supportive of candidates publishing their research results during their candidature as detailed in the UNSW Thesis Examination Procedure.

Publications can be used in your thesis in lieu of a Chapter provided:

- You contributed **greater than 50%** of the content in the publication and are the "primary author", i.e. you were responsible primarily for the planning, execution and preparation of the work for publication.
- You have approval to include the publication in their thesis in lieu of a Chapter from your Supervisor and Postgraduate Coordinator.
- The publication is not subject to any obligations or contractual agreements with a third party that would constrain its inclusion in the thesis.

Some of the work described in my thesis has been published and it has been documented in the relevant Chapters with acknowledgement.

A short statement on where this work appears in the thesis and how this work is acknowledged within chapter/s:

The formal preliminaries in Chapter 3 expand on a similar presentation I gave in a conference paper of which I was the primary author, published at Interactive Theorem Proving (ITP 2019). This is acknowledged at the beginning of Chapter 3.

The contributions of Chapter 4 led to a conference paper to which I contributed, published at European Symposium on Security and Privacy (EuroS&P 2018), which also first presented the case study of Chapter 6.

The contributions of Chapter 5 were first presented by me at the Workshop on Principles of Secure Compilation (PriSC 2018), then subsequently presented in more detail by me in the ITP 2019 conference paper.

For further discussion of these publications, their relationship with this thesis, and all acknowledgments of the work of others, please see the "Overview of contributions" section in Chapter 1, and the "Publications and acknowledgements" section in each of Chapters 4, 5, and 6.

CANDIDATE'S DECLARATION

I declare that I have complied with the Thesis Examination Procedure.

Candidate's Name	Signature	Date (dd/mm/yy)

Abstract

Here, I pose the thesis that **proving noninterference and its preservation by a compiler is feasible for mixed-sensitivity concurrent programs**. 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. Prior work therefore presented formal methods for proving and preserving the strictest kind of confidentiality property, *noninterference*, for *mixed-sensitivity concurrent programs*: a term I coin to describe those programs that might reuse memory shared between their threads to hold data of different sensitivity levels at different times. Although these methods addressed challenges in formalising the *value-dependent* coordination of such *mixed-sensitivity reuse* under the impact of *concurrency*, their practicality remained unclear: Could they be used to prove noninterference for any nontrivial mixed-sensitivity concurrent program in its entirety? Furthermore, could any compiler be verified to preserve the needed guarantees to the compiled code?

To support this claim, I prove for the first time both (1) noninterference for a nontrivial mixed-sensitivity concurrent program, modelling a real-world use case, and (2) its preservation by a compiler down to an assembly-level model. This main result rests on two major contributions. First, I demonstrate how programming-language designers can make reasoning on each thread sufficient to prove noninterference for such programs, by supplying synchronisation primitives (here, mutex locks for a generic imperative language) and proving they maintain as invariant the necessary requirements. Second, I demonstrate how compiler developers can make *confidentiality-preserving refinement* a feasible target for verification, by using a decomposition principle to prove that a compiler (here, from that imperative language to a generic RISC-style assembly language) establishes it for mixed-sensitivity concurrent programs. Thus, per-thread reasoning proves noninterference for the case study, and the verified compiler preserves it to assembly automatically. All my results are formalised and proved in the Isabelle/HOL interactive proof assistant.

My work paves the way for more fully featured programming languages and their compilers, in replicating these results, to raise the typical level of assurance readily offered by developers of multithreaded software responsible for data of multiple sensitivity levels.

Acknowledgements

Foremost, I thank my supervisors Carroll Morgan, Toby Murray, and Kai Engelhardt, who taught me how to be a researcher in this field.

For cultivating an excellent working and learning environment, I would like to thank the Trustworthy Systems group at CSIRO's Data61 under the leadership of June Andronick, and all the regular attendees of its reading groups.

For memorable conversations and other inputs and feedback on my work that shaped both its direction over the years of my candidature, and its presentation in this dissertation, I thank also Gerwin Klein, Christine Rizkallah, Edward Pierzchalski, Ramana Kumar, Matthew Brecknell, Johannes Åman Pohjola, Qian Ge, and Gernot Heiser. For their careful evaluation of its merits and shortcomings, I thank Gilles Barthe and David Sands.

Furthermore, I thank Alexander Legg and Anna Lyons for their mentoring and advice, delivered variously over coffee and in the midst of a metal concert.

Finally, I thank my partner Leo and my extended family for their ongoing support, and I dedicate this work to the memory of my grandfather, an architect.

This research was funded by an Australian Government Research Training Program (RTP) Scholarship, and a CSIRO Data61 Research Project Award.

Publications

The contributions of Chapter 4 led to the following conference paper, which also first presented the case study of Chapter 6:

- [68] Toby Murray, Robert Sison, and Kai Engelhardt. COVERN: A logic for compositional verification of information flow control. In *European Symposium on Security and Privacy*, pages 16–30, London, United Kingdom, April 2018. IEEE.

The contributions of Chapter 5 led to a workshop presentation and a conference paper, both of which I presented:

- [83] Robert Sison. Per-thread compositional compilation for confidentiality-preserving concurrent programs. In *2nd Workshop on Principles of Secure Compilation*, Los Angeles, January 2018. Cătălin Hriţcu.

- [85] Robert Sison and Toby Murray. Verifying that a compiler preserves concurrent value-dependent information-flow security. In *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141, pages 27:1–27:19, Portland, USA, September 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

For further discussion of these publications and their relationship with this thesis, please see Section 1.3 ("**Overview of contributions**"), and Sections 4.5, 5.6, and 6.7 ("**Publications and acknowledgements**").

List of Figures

1.1 Example of conversion of an internal timing leak into a storage leak 4
1.2 Example of mixed-sensitivity reuse 6

3.1 Excerpts from a CVDNI-preserving refinement example 34
3.2 Definition and graphical depiction of refinement preservation obligation . . 35
3.3 Graphical depictions of decomposed refinement preservation obligations . . 37

4.1 Annotation versus enforcement of assumptions on shared memory access . . 40

6.1 Functional schematics for Cross Domain Desktop Compositor hardware . . 82
6.2 Functional schematic of seL4 component architecture for CDDC HID switch 83
6.3 Examples of external device interactions by the CDDC HID switch model . 85
6.4 Examples of the SWITCH component interfacing with the compositor device 87
6.5 Remote procedure call abstraction between SWITCH, OVERLAY components 88
6.6 Examples of the SWITCH component interacting with the input-event buffer. 90

7.1 Examples of secret-dependent control flow admitted by typing rule IFH . . 98
7.2 Example program admitted by typing rule IFH with nested branching . . . 108

A.1 Introduction rule for case `if_expr` of refinement relation $\mathcal{R}_{\mathsf{wr}}$ 131
A.2 Excerpt of wr-compiler implementation: case for **if**-conditionals 131

Contents

Abstract iii
Acknowledgements v
Publications vii
List of Figures ix

1 Confidentiality in the face of scale 1
1.1 Mixed-sensitivity concurrency and its challenges . . . 2
1.1.1 Concurrency: Scaling with the number of workers . . . 2
1.1.2 Mixed-sensitivity reuse: Scaling with the number of customers . . . 5
1.1.3 Compositionality: Scaling with the size of the software project . . . 6
1.2 Approaches taken by this thesis . . . 7
1.2.1 Assume–guarantee contracts, on worker–resource access . . . 7
1.2.2 Value-dependent classification, for customer–resource dedication . . . 8
1.2.3 Confidentiality-preserving refinement, for compiler verification . . . 8
1.2.4 Machine-checked formalisation and proof . . . 8
1.3 Overview of contributions . . . 8
1.3.1 Completion of a program verification method . . . 9
1.3.2 Demonstration of a compiler verification method . . . 9
1.3.3 Demonstration of program verification, preserved by a compiler . . . 10
1.3.4 Program verification extension for secret-dependent control flow . . . 10

2 Review of related formal methods 11
2.1 Defining noninterference . . . 11
2.2 Program verification: Proving noninterference . . . 12
2.2.1 With better precision, through flow sensitivity . . . 12
2.2.2 With mixed-sensitivity reuse, through value dependence . . . 13
2.2.3 With concurrency, done compositionally . . . 14
2.2.4 With both mixed-sensitivity reuse and concurrency . . . 19
2.2.5 With secret-dependent control flow, despite concurrency . . . 20
2.3 Compiler verification: Proving property preservation . . . 22
2.3.1 For single-trace properties, like safety and liveness . . . 22
2.3.2 For multiple-trace hyperproperties, like noninterference . . . 23
2.3.3 For noninterference with concurrency . . . 24

3 Formal preliminaries from background work **27**
3.1 Concurrent value-dependent noninterference (CVDNI) . . . 28
3.1.1 Basic elements . . . 28
3.1.2 Parameters for initial conditions and extra requirements . . . 31
3.1.3 Compositionality requirements . . . 32
3.2 CVDNI-preserving refinement . . . 33
3.2.1 Original notion with cube-shaped diagram . . . 34
3.2.2 Decomposition principle . . . 36
3.3 Summary . . . 38

4 Compositional noninterference for a `While` language with mutex locks **39**
4.1 Source language: `While` with mutex locks . . . 41
4.1.1 Locking discipline and its semantics . . . 42
4.1.2 Restrictions on locking disciplines . . . 44
4.2 Local mode compliance check . . . 45
4.2.1 New rules for mutex locks . . . 46
4.2.2 Proof of soundness . . . 47
4.3 Global modes compatibility . . . 48
4.3.1 Proof of invariance . . . 48
4.3.2 Initial conditions . . . 51
4.4 Security type system . . . 52
4.4.1 New rules for mutex locks . . . 53
4.4.2 Proof of soundness . . . 55
4.5 Publications and acknowledgements . . . 60
4.6 Summary . . . 60

5 Noninterference-preserving compiler for mixed-sensitivity concurrent `While` programs **63**
5.1 Target language: `RISC` with mutex locks . . . 64
5.2 Preserving race-free expression evaluation . . . 65
5.2.1 Requirements on inputs to the wr-compiler . . . 66
5.2.2 Proof that all tracked register contents are stable . . . 68
5.3 Preserving a ban on secret-dependent control flow . . . 68
5.4 Use of the decomposition principle . . . 69
5.4.1 Refinement relation $\mathcal{R}_{\mathsf{wr}}$ and its invariants . . . 70
5.4.2 Refinement pacing function $\mathsf{abs\text{-}steps}_{\mathsf{wr}}$ . . . 71
5.4.3 Concrete coupling invariant $\mathcal{I}_{\mathsf{wr}}$ . . . 74
5.4.4 Proof of CVDNI-preserving refinement . . . 74
5.5 Proof of compositional noninterference preservation . . . 76
5.6 Publications and acknowledgements . . . 79
5.7 Summary . . . 80

6 Case study: Cross Domain Desktop Compositor input handler 81
6.1 Overview of the case study . . . 82
6.2 External device interactions . . . 84
6.2.1 Attacker model (at the output devices) . . . 85
6.2.2 Trusted user model (at the input devices) . . . 85
6.3 Internal device and inter-component interactions . . . 86
6.3.1 Use of the compositor device (by Overlay driver, Switch) . . . 86
6.3.2 Remote procedure calls (between Switch and Overlay driver) . . . 87
6.3.3 Input-event buffer (between Input driver and Switch) . . . 88
6.4 Proof of confidentiality for the `While` model . . . 90
6.5 Automation and user intervention . . . 92
6.6 Confidentiality-preserving compilation to `RISC` model . . . 93
6.7 Publications and acknowledgements . . . 95
6.8 Summary . . . 96

7 Security typing extension for secret-dependent control flow 97
7.1 Intuitions behind the new security typing rule IfH . . . 98
7.2 Updates to `While`-language security type system . . . 99
7.3 Updates to bisimulation construction . . . 101
7.4 Proof of soundness of the new typing rule . . . 103
7.4.1 The new bisimulation construction is still a security witness . . . 103
7.4.2 The construction still captures well-typed program execution . . . 106
7.5 Summary . . . 107

8 Conclusion 109
8.1 What my work showed . . . 109
8.1.1 Assume–guarantee makes it feasible to prove that synchronisation primitives make CVDNI proofs compositional . . . 109
8.1.2 Decomposition principles make it more feasible to prove that compilers satisfy CVDNI-preserving refinement . . . 110
8.1.3 Security type systems can prove CVDNI automatically for programs with secret-dependent control flow . . . 110
8.2 Why it matters: Scaling up with scarce resources . . . 110
8.2.1 For multithreaded system software that implements sharing . . . 111
8.2.2 Support by programming languages and their compilers . . . 111
8.3 What might come next? . . . 112
8.3.1 Languages with indirect-addressing commands . . . 112
8.3.2 More compile-time optimisations on shared memory . . . 113
8.3.3 Compiler preservation of general assume–guarantee reasoning . . . 113
8.3.4 Further support for secret-dependent control flow . . . 113
8.4 Closing remarks . . . 114

Bibliography **115**

A Extra details about the Covern wr-compiler **127**
A.1 Label allocation and sequential composability 127
A.2 Register allocation scheme model 127
A.3 Informal descriptions of cases of refinement relation $\mathcal{R}_{\mathsf{wr}}$ 128
A.3.1 Base cases 128
A.3.2 Inductive cases 128

B Extra details about the IfH security typing rule **133**

Chapter 1

Confidentiality in the face of scale

This thesis is about techniques for proving two things:

1. **The *total absence of information flow* between a trusted and untrusted party in a given program**, including **concurrent programs** that **reuse shared memory** to scale up their capacity to service different parties at different times.

   For such programs, this thesis in particular concerns proofs eliminating the flow of information between two sets of memory locations—*sensitive sources* and *untrusted sinks*—designated by the user of the techniques as belonging to each party. Information coming from the former set of locations will also be described as *sensitive*, and be called *secrets*. The *attacker model* throughout this thesis will be an entity with the power to read at any time from the untrusted sinks.

   This kind of strict prohibition of information flow is called *noninterference* [32] in the literature. Unless specified otherwise, this thesis will also use the broader terms *confidentiality*, *information-flow security*, and *security* to mean noninterference.

2. **That a compiler *preserves this total absence of information flow* in such programs when translating them to other languages.**

To be clear, this thesis is *not* about placing upper bounds on some small but necessary amount of information leakage between a trusted and untrusted party. If I have the job of "password checker", the techniques studied here will inevitably find leaks, for example, between the piece of paper from my boss with today's secret password, and the answer of "accepted" or "rejected" I give to the stranger at the window who tried to guess it. Even if it would take the stranger thousands of guesses, the techniques in this thesis will flag even this tiniest of leaks in a single one of my "yes" or "no" answers as unsafe. I must defer to the authority of other, leakage-bounding techniques like *quantitative information flow* [87] or *differential privacy* [4], for determining whether the rate of leakage inherent in the application domain is acceptable—such a task lies outside the scope of this thesis.

This thesis is more about whether I inadvertently leave the password on the coffee table in the break room where we host discussions with external clients who visit the building—or inexplicably blurt out the password at the cafeteria when I'm ordering lunch.

These kinds of extremely obviously wrong behaviours regarding confidentiality may sound easy to avoid. In fact, it sounds so easy that when a computer program about which we have full details does the right thing, surely we should be able to *prove* that it holds, in a way that arises *from the form* of that program. Indeed, techniques already exist that make it straightforward to *prove formally* that some kinds of computer programs do not do the obvious wrong thing, like leaving the password on the proverbial coffee table.

This thesis, however, will be about those programs for which the reasoning of these techniques breaks down—where such an apparently easy task is not, in fact, so easy.

1.1 Mixed-sensitivity concurrency and its challenges

The specific focus of this thesis is on how these two problems—(1) the *program verification* problem of proving total absence of information flow, and (2) the *compiler verification* problem of proving its preservation by a compiler—become more challenging when a program has both of the following characteristics:

- *Concurrency* (Section 1.1.1): specifically, concurrency of access by different threads of execution, to memory locations shared between those threads.
- *Mixed-sensitivity reuse* (Section 1.1.2): a term I coin to describe a reuse of memory locations to hold information of differing levels of sensitivity, at different times.

These characteristics are answers to fundamental problems of scale. They arise, respectively, from the need to divide work in computer systems that handle information, and 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, and whose responsibility is to implement that sharing; proving that it does not permit the information of one customer to flow to another is the objective of this thesis.

As answers to problems of scale, these characteristics pose an additional challenge: *Compositionality* (Section 1.1.3) demands that a proof about a program be composable from proofs about its parts, that are each not overly sensitive to changes to the other parts. This thesis focuses on compositionality of proofs about program threads—in this way, the proof effort can scale with the development effort as split into *concurrent components*.

In addressing these challenges, this thesis demonstrates that it is feasible to prove noninterference and its preservation under compilation for *mixed-sensitivity concurrent programs*, a term I coin for those whose threads make mixed-sensitivity reuse of the memory they share. Prior to my doctoral studies, no formal techniques had been exercised to prove noninterference, or its preservation by a compiler, for such a program in its entirety.

1.1.1 Concurrency: Scaling with the number of workers

Concurrency poses two new challenges when proving noninterference properties and their preservation by a compiler: (1) managing the proof impact of inter-thread interactions,

and (2) preventing the conversion of internal timing leaks into storage leaks.

Inter-thread interactions, as *interference* and *intermediate value leakage*

It is well known that the possibility of interactions between the threads of a concurrent program—as coworkers that need to cooperate using a shared pool of resources—leads to the number of possible configurations growing exponentially in the number of threads.

The main impact on proofs of confidentiality for such programs is that the usual shortcuts that are available to analyses of the behaviour of a *sequential program* (a single thread of execution assumed to be working in complete isolation) no longer apply:

- Analyses of confidentiality for sequential programs will typically assume that secrets (including *any intermediate values* derived from secrets) can be stored temporarily at locations that might only become visible to the attacker at the end of the program.

  However, in a concurrent program I cannot assume that secrets stored at such a location will not be read by another worker, if that location is in a shared space.

- As Section 1.1.2 will explain: **When coordinating mixed-sensitivity reuse, a program's functional-correctness issues can now become security issues**.

  Analyses of the functionality of sequential programs will typically assume that values written to a location will still be there later, unless that one thread of execution writes to the location again. This allows such analyses to be *flow sensitive* [19], meaning they can take into account what happens before and after a given point in the program—i.e. be *sensitive* to that program point's position in *the control flow*.

  However, as a worker thread in a concurrent program, I cannot assume that something will be where I left it, because another worker thread might have replaced it with something else. This inherent *instability* of the contents of the shared space, due to *interference* by other threads in the program, thus (if not mitigated) poses an obstacle to flow sensitive analysis of a concurrent program's functionality.

These issues pose the challenge of finding ways to reason about the **coordination of workers' access** to certain resources at certain times. With a means of formalising this coordination, (1) confidentiality proofs for concurrent programs can be structured to recognise when the coordination rules out instances of instability due to interference, or inadvertent leaks due to other workers' reading of secret intermediate values; and (2) verified compilers can be proved formally to preserve any such methods of coordination.

Internal timing leaks become *storage leaks*, without a specialised scheduler

Concurrent settings with shared space are notorious for converting internal timing leaks into storage leaks. This section will explain the problem, and the difficulties arising from preventing it without overly constraining programs, or specialising the *scheduler* to assist—that is, the part of the execution environment that chooses how threads are interleaved.

```
Thread A  1: if (h) {          Thread B  6: if (s) {
          2:   v := TRUE;                 7:   x := TRUE;
             } else {                     } else {
          3:   v := FALSE;                8:   x := FALSE;
          4:   skip;                      }
             }
          5: s := TRUE;
```

Figure 1.1: Example of conversion of an internal timing leak into a storage leak.
Here, `h` contains a secret, `s` is shared between both threads, and `x` is an untrusted sink.

Informally, *storage leaks* (discernible by a difference in value) are often distinguished from *timing leaks* (discernible by a difference in the timing of identical changes in value) because an attacker that can easily detect a storage leak (e.g. the final value of `v` in Thread A in Figure 1.1) may have no way of measuring the timing of changes to storage reliably enough to exploit a timing leak (e.g. the timing of Thread A's assignment of `TRUE` to `s`).

In a concurrent program, however, a timing leak must be treated as seriously as any storage leak. Consider the (pseudocode) example program in Figure 1.1, with `s = FALSE` initially: If the scheduler chooses to switch execution to Thread B after executing three of the numbered commands of Thread A, this program produces a storage leak of the exact (boolean) value of `h` to the variable `x`, which stems from the timing leak of `h` determining whether the assignment to `s` has happened yet. This example illustrates two principles:

1. The notion of time that matters is *scheduler relative*: counted by the distinct points at which the scheduler might interleave a thread's execution with that of others. The literature typically calls this *internal time* (and *internal timing leaks*); I will sometimes use the term *scheduler-relative time*, to emphasise the role of the scheduler.

2. Internal timing leaks are propagated to storage by *race conditions*; here, a race between Thread A's write and Thread B's read of `s` determines the final value of `x`.

Thus, early approaches included: (1) having programs avoid race conditions by disallowing all asynchronous communication via shared memory [101], including that which would be needed to implement synchronisation primitives; or, (2) specialising the scheduler to control the notion of time to avoid leaky schedules [9, 10]. For instance, if the scheduler here only ever chose to switch to Thread B after two (or four) of the numbered commands of Thread A, it could ensure that the final value of `x` is unconditionally `FALSE` (resp. `TRUE`).

Unfortunately, it is not always feasible to impose special requirements like these on the scheduler. Furthermore, race-prone (asynchronous) interactions on shared memory between threads may be inherent to the level of abstraction at which the program is to be analysed—for example, at the operating-system level, or including the implementation of synchronisation primitives. This obliges us to verify that each program thread satisfies a *timing-sensitive* notion of noninterference: specifically, one that rejects the program if it has any internal timing leaks (relative to the scheduler) observable via race-prone

shared memory. This timing sensitivity has follow-on implications both for **compiler verification**, and for verifying **programs with secret-dependent control flow**:

- Compilers (hardware aside! [31]) tend to optimise as aggressively as possible to make the program run as fast as possible. Consequently, the soundness of any source-level program verification of timing-sensitive noninterference rests on imposing more nuanced compiler verification requirements (regarding timing), than any typically imposed on a production-quality compiler for a mainstream programming language.

  However, practical approaches to verifying that a compiler preserves timing-sensitive notions of noninterference are only recently beginning to be explored [11, 12]. Thus, this thesis adds to that body of work, by exploring the satisfiability of requirements specialised to mixed-sensitivity concurrent programs [67, 66].

- As timing leaks stem from dependence of timing behaviour on secrets, a common approach (followed by the first few contributions of this thesis) is to avoid precise source-level reasoning about time by disallowing secret-dependent control flow—for example, disallowing `if (h)` conditionals to prevent any timing leaks from `h`.

  However, a strength of timing-sensitive notions of noninterference proposed recently for mixed-sensitivity concurrent programs [67, 65] is to be compatible with such precise source-level reasoning about the time taken by such control flow paths. Thus, this thesis presents (as a final contribution) an example of such reasoning.

1.1.2 Mixed-sensitivity reuse: Scaling with the number of customers

As previously mentioned, the program verification problem also becomes more challenging when (imagining that we are threads in a concurrent program) my coworkers and I want to reuse any shared space between us to serve two or more mutually distrusting parties.

Under these circumstances, we cannot use a shortcut that makes proving confidentiality easy: calling a workspace permanently tainted if it is used for a given customer A, and then never using it to serve customer B, and vice versa. This is because if we were to use this shortcut, we would need two permanently dedicated workspaces: one for customer A, and the other for customer B. Then, if we served sixty thousand customers, we would need sixty thousand workspaces—this is clearly inefficient, if there is nothing stopping us from flushing the content of one customer before reusing the workspace for another. Furthermore, if we do not know how many customers we are liable to have, we would not even know how many workspaces to set aside. Note that, in a nutshell, it was this impetus for flexible reuse of shared resources—to prevent inefficiency and over- or under-provisioning—that in large part led to the rise to supremacy of cloud architectures.

Rather, any analysis we develop must account more generally for the possibility that a workspace might be used for customer A at one moment, and for customer B the next. The only conceivable way to do this predictably is to have the dedication of that workspace be coordinated by values handled or calculated by the program itself (e.g. as in Figure 1.2).

```
if (current_customer = 0) {
  customer_A := workspace;
} else {
  customer_B := workspace;
}
```

Figure 1.2: Example of mixed-sensitivity reuse—here, of the `workspace`, depending on the value of `current_customer`.

The security property must therefore be *value dependent* [103, 51, 64] to account for the dynamically changing dedication of the workspace, meaning the analysis must be flow sensitive (Section 1.1.1) enough to track the values on which that property is dependent.

As foreshadowed in Section 1.1.1, however, such value-dependent coordination and its flow-sensitive analysis may be complicated by concurrency. Consider the (pseudocode) example program in Figure 1.2: If `workspace` contains customer B's data, but another thread overwrites `current_customer` with 0, the program will incorrectly deliver the data to customer A. It is in this sense that, as mentioned in Section 1.1.1, functional-correctness issues can become security issues; thus, **program functionality coordinating mixed-sensitivity reuse needs to be protected from concurrency**, with values calculated needing to be analysed just as seriously as their sensitivity levels. Consequently, mitigations against concurrency's interference with a program thread's functionality (see Section 1.2.1) will become critical to protect also the confidentiality of such programs.

In short, the challenges so far can be summed up as follows: A program with mixed-sensitivity reuse inevitably features value-dependent coordination of that reuse, therefore requiring a flow-sensitive security analysis to track the changes in those values; however, as just mentioned in Section 1.1.1, concurrency interferes with that flow-sensitive analysis.

Then on top of that, we would ideally be able to support verification of a wider variety of software architectures, that might allow a given worker thread to service n customers, or where one customer is serviced (perhaps for m different purposes) by m worker threads. Developing techniques for verifying such programs poses the challenge of finding ways to reason about how worker threads coordinate the dedication of resources to customers, without necessarily tying together workers with customers in a 1:1 relationship. This means managing how formalisms for coordinating customer–resource dedication (see Section 1.2.2) interact with those for coordinating worker–resource access (see Section 1.2.1).

1.1.3 Compositionality: Scaling with the size of the software project

Finally, the proof effort for any techniques that this thesis develops (for handling concurrency and mixed-sensitivity reuse) needs to scale with the size of the software project, in terms of the number of distinctly developed software components it comprises.

Particularly, the program verification effort should scale *compositionally*, meaning **proof effort on each software component is possible without (yet, or possibly**

ever) knowing the internal details of all the other software components in the system. Furthermore, the compiler verification theorem should ensure that the compiler preserves the compositionality of whatever confidentiality property it is preserving.

This thesis will focus on the development of techniques that scale and compose proof effort for an arbitrary number of **worker threads as software components**, assuming that the code run by each thread forms a natural area of responsibility or ongoing maintenance in a long-term software development for a concurrent program. When considering worker threads as software components, this adds to the concerns of scalability raised in Section 1.1.1, because while a proof technique may vastly reduce the complexity of an analysis, it may still not be compositional—a notable example is Owicki and Gries [74].

In contrast however, this thesis will simplify to the typical basic case of servicing two customers, assuming that verifying for an arbitrary number of customers amounts to repeated applications of the basic case. We leave to future work any issues of scale in verifying for an arbitrary number of customers (e.g. as in Lourenço and Caires [51]).

1.2 Approaches taken by this thesis

In Section 1.1, I described how the problems of scale addressed by mixed-sensitivity concurrent programs beg for methods of coordination—between worker threads, and towards their cooperative dedication of resources to customers serviced—and how they must give rise to program- and compiler-verification efforts that scale compositionally in the number of threads. Here I will talk about the approaches chosen for dealing with those two challenges of coordination (described in Sections 1.2.1 and 1.2.2 respectively), and for proving that a compiler preserves noninterference for mixed-sensitivity concurrent programs (Section 1.2.3). Finally I describe briefly the methodology for experimenting with the improvements and the application of all of these approaches (Section 1.2.4).

1.2.1 Assume–guarantee contracts, on worker–resource access

This thesis continues a line of experimentation on the adaptation to noninterference verification [54, 64, 67] of *assume–guarantee* [54] (another name for *rely–guarantee* [41]), a method for achieving compositional analysis for concurrent programs.

In these confidentiality-focused adaptations of assume–guarantee, inter-thread interference is limited by the use of *assumptions* (also known as *rely conditions*) that other worker threads will not write to certain areas of the shared space. Inadvertent leakage of intermediate values is handled using a more confidentiality-centric kind of assumption (as pioneered by Mantel et al. [54]), that other workers will not read from certain areas.

However, as is standard for assume–guarantee, the greater ease of per-thread analysis (due to assumptions) is paid for by proof obligations to establish certain *guarantees* (corresponding to those assumptions). The first focus of this thesis is on a method for proving *all* of these contractual obligations on resource access by threads—these were only partially proved by any known instantiation [33, 65] of the preceding theories [54, 67].

1.2.2 Value-dependent classification, for customer–resource dedication

Static classification of locations corresponds to the inflexible scenario illustrated in Section 1.1.2, in which workspaces tainted by data belonging to a given customer must be classified as belonging to that customer for all time. In contrast, mixed-sensitivity reuse requires dynamic changes to the classification of locations.

This thesis continues experimentation in the literature [64, 67] on confidentiality properties that support *value-dependent classification* schemes, in which the classification of a location can change dynamically in a manner *dependent on values* currently held in other locations. This implies a recognition of which of the variables in the program—termed the *control variables* [64]—are significant for *controlling the classification* of other variables. (In Figure 1.2, `current_customer` would likely control the classification of `workspace`.)

For control variables to function as an effective means of coordinating customer–resource dedication, these approaches require the instantiator of the theory to demonstrate the satisfaction of certain requirements in the treatment of these variables—for example, the requirement that sensitive information belonging to one customer be flushed from a shared location, before it is reclassified to belong to another customer.

1.2.3 Confidentiality-preserving refinement, for compiler verification

Tying the above concerns together, this thesis gauges the applicability to compiler verification of notions of *confidentiality-preserving refinement* [66, 85], that specifically preserve a timing-sensitive, assume–guarantee-supporting notion of noninterference with value-dependent classification policies [67, 65].

This seeks to mirror the usual approaches of large-scale compiler verification efforts like CompCert [48] and CakeML [47] of using various notions of refinement to prove preservation of program semantics—but in this case, satisfying a notion of refinement expressly designed to preserve security while tackling the difficulties raised in Section 1.1.

1.2.4 Machine-checked formalisation and proof

All theories described in this thesis are simplified presentations of theories formalised [84] in Isabelle/HOL (Isabelle's higher-order logic) [71], with development of the theories having proceeded using the Isabelle interactive proof assistant as a development environment.

The Isabelle/HOL theories for this thesis [84] build directly on top of the previous Isabelle formalisations [65, 66] of Murray et al. [67]—work I assisted prior to this thesis—which in turn had extended the Isabelle-based formalisation [33] of Mantel et al. [54].

1.3 Overview of contributions

This thesis evaluates the feasibility of formal methods (1) for verifying noninterference—total absence of information flow between certain designated locations—for a mixed-sensitivity concurrent software program, and (2) for having that noninterference verifi-

cation proof effort be carried down to a target language semantics by a one-time compiler verification proof effort. Consequently, compiling the threads of such a program yields a concurrent program that also satisfies noninterference.

An extension of the program verification method is also presented, allowing some forms of secret-dependent control flow (as a foundation for future extensions of the compiler).

1.3.1 Completion of a program verification method

Chapter 4 presents the first contribution of this thesis: a novel set of per-thread techniques that is sufficient to prove noninterference for mixed-sensitivity concurrent programs. They show programming-language designers that assume–guarantee makes it feasible to prove that their synchronisation primitives make these proofs compositional.

Specifically I extend, for a generic imperative language, a prior set of proof techniques by Murray et al. [67, 65] whose compositionality was conditional on some unproved requirements. I add mutex lock-based synchronisation primitives to that language, and prove that their semantics maintains the needed compositionality requirements as an invariant of program execution. I also add type-checking rules for the new primitives accordingly, and prove the updated rule sets remain sound. The requirements proved are also from Murray et al. [67, 65], and I present them as formal preliminaries in Chapter 3.

The work of Chapter 4 led to a program logic (with additional features falling outside the scope of this thesis; see Section 2.2.4) that establishes security for mixed-sensitivity concurrent programs using mutex locks, presented in the following paper:

- [68] Toby Murray, Robert Sison, and Kai Engelhardt. COVERN: A logic for compositional verification of information flow control. In *European Symposium on Security and Privacy*, pages 16–30, London, United Kingdom, April 2018. IEEE.

1.3.2 Demonstration of a compiler verification method

Chapter 5 then presents the second contribution of this thesis: the first compiler proved to preserve proofs of noninterference for mixed-sensitivity concurrent programs. This shows compiler developers that decomposition principles make it more feasible to prove that compilers satisfy noninterference-preserving notions of refinement.

Specifically I implement an Isabelle/HOL function that compiles programs from the language of Chapter 4 to a generic RISC-style assembly language, adapted from a compilation scheme originally intended for fault-resilience [95]. I then prove that this compiler satisfies a confidentiality-preserving notion of refinement from Murray et al. [67, 66], using a decomposition principle (falling outside the scope of this thesis) published alongside the compiler [85]. I present these background notions as formal preliminaries in Chapter 3.

The work of Chapter 5 appeared in the following paper:

- [85] Robert Sison and Toby Murray. Verifying that a compiler preserves concurrent value-dependent information-flow security. In *10th International Conference on*

Interactive Theorem Proving (ITP 2019), volume 141, pages 27:1–27:19, Portland, USA, September 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

1.3.3 Demonstration of program verification, preserved by a compiler

Chapter 6 presents the third and central contribution of this thesis, resting on those of Chapters 4 and 5: the first significant mixed-sensitivity concurrent program proved to satisfy a noninterference property, preserved by a compiler down to an assembly-level model. This validates the central claim of this thesis, that proving confidentiality and its preservation by a compiler is feasible for such programs.

Specifically I implement, in the language of Chapter 4, a model of an actual non-trivial mixed-sensitivity concurrent program serving a real-world use case: the software-componentised input-handling regime for the *Cross Domain Desktop Compositor* [13] (CDDC). I then exercise the proof techniques of Chapter 4 on each thread to yield a proof of noninterference for the model. Finally, I show that it is straightforward to use the proofs about the compiler of Chapter 5 to obtain noninterference automatically for a RISC-style assembly language model of the program, output by the compiler.

A demonstration of all these methods on a two-component variant of the CDDC input-handling model appeared in the paper Sison and Murray [85] (noted by Section 1.3.2) that presented the contributions of Chapter 5. As case study, this thesis presents a three-component model that is more faithful to the CDDC input handler's original C-language implementation than the two-component model verified and compiled by Sison and Murray [85]. This three-component model previously appeared in the paper Murray et al. [68] (noted by Section 1.3.1), except that the version in this thesis checks certain functional properties at runtime, instead of discharging them using the general assume–guarantee verification support added by the Covern logic [68].

1.3.4 Program verification extension for secret-dependent control flow

Although secret-dependent control flow is compatible with the underlying theory of Murray et al. [67] on which this thesis rests, the contributions of Chapters 4 and 5 as applied in Chapter 6 disallow it—a practice common in the literature when proving confidentiality.

To address this, Chapter 7 presents a fourth and final contribution: the first instance of syntax-directed reasoning about secret-dependent control flow for mixed-sensitivity concurrent programs. This shows programming-language designers that security type systems are capable of such reasoning.

Specifically I add to the security type system of Chapter 4 a rule that admits limited forms of conditional branching on secrets, and prove that it remains sound. In short, the rule ensures the absence of any timing leaks that are convertible to storage leaks, assuming a scheduler making decisions according to the same notion of time.

This final contribution is intended to facilitate future work on extending the compiler of Chapter 5 to allow noninterference-preserving compilation of such programs.

Chapter 2

Review of related formal methods

As explained in Chapter 1, this thesis is about proving particular forms of a family of confidentiality properties referred to broadly as *noninterference* [32], and its preservation by a compiler, that are designed to cater to mixed-sensitivity concurrent programs.

After introducing the basic forms of noninterference properties and what distinguishes them from the more typical safety properties (Section 2.1), this chapter will present a literature review divided into each of the two problems introduced in Chapter 1—program verification (Section 2.2) and compiler verification (Section 2.3)—gradually narrowing in on the aspects of those problems most relevant to this thesis. A formal presentation of preliminaries from background work on these topics will be given separately, in Chapter 3.

2.1 Defining noninterference

The term "noninterference", as used in the field of security verification, was coined by Goguen and Meseguer [32] to mean: "what the first group (of users) does... has no effect on what the second group of users can see." The literature typically refers to this first and second group respectively as the *high* and *low* user or classification.

This thesis will mostly focus on definitions of noninterference that assert that any two initial states of some program that "look the same" to that low-classified group of users must execute with semantics that also "look the same" to that group. Overloading $\simeq$ to denote some formal notion of *(low-)observational equivalence* both (1) between states and (2) between behaviours of the program (to a low user), and using $[\![\sigma]\!]$ to denote its behaviour when executed with initial state σ, these definitions broadly take the form:

$$\forall \sigma_1\ \sigma_2.\ \sigma_1 \simeq \sigma_2 \longrightarrow [\![\sigma_1]\!] \simeq [\![\sigma_2]\!]$$

Noninterference properties are *2-safety* [96] *hyperproperties* [20]: Two executions are necessary to refute noninterference, whereas *safety* properties can be refuted by a single execution. A counterexample pair of executions to noninterference, when formalised as just shown, indicates a *leakage* of information in the form of some difference in the state visible to low user, resulting from differences in some part of the state that cannot be seen

by them—such "hidden" differences could have only resulted from the high user. Formally:

$$\exists \sigma_1\ \sigma_2.\ \sigma_1 \simeq \sigma_2 \wedge [\![\sigma_1]\!] \not\simeq [\![\sigma_2]\!]$$

Information leakage was first formalised in this manner in 1977 by Cohen [22], who termed that leakage a *strong dependency*, and formalised confidentiality in terms of its negation—some 5 years before Goguen and Meseguer [32]. Defining confidentiality in much the same form, seemingly independently of Cohen [22] in the mid-90s, Volpano et al. [99] described it as "noninterference-like", and the name has stuck ever since.

The style of noninterference definition described so far can be contrasted to *purge*-based definitions, which more closely follow Goguen and Meseguer [32]: These instead compare the behaviour of a program before and after applying a "purge" function that removes secrets from an input to the program. Overloading $\simeq$ to denote low-observational equivalence between input strings, and $[\![w]\!]$ to denote the program's behaviour when executed on input string w, Goguen and Meseguer's noninterference took the form:

$$\forall w.\ [\![w]\!] \simeq [\![\mathsf{purge}(w)]\!]$$

Note that this is just as much a 2-safety hyperproperty; it is refuted by a counterexample pair of executions of the program that have different observable semantics before and after purging secrets from the input string—that is, $\exists w.\ [\![w]\!] \not\simeq [\![\mathsf{purge}(w)]\!]$.

2.2 Program verification: Proving noninterference

This section will review the literature on improving the precision of analyses (Section 2.2.1) seeking to establish noninterference, and on having these analyses support programs with mixed-sensitivity reuse (Section 2.2.2) and concurrency, in a manner that is compositional across the concurrently running threads (Section 2.2.3). (The formal preliminaries provided in Chapter 3 will then centre on a particular form of noninterference from the literature that combines all three concerns.) After a brief discussion of works contemporary with and subsequent to the work in this thesis that also reconcile mixed-sensitivity reuse and concurrency compositionally (Section 2.2.4), the final part of this literature survey on program verification will focus on related works' decision of whether to allow or disallow secret-dependent control flow (Section 2.2.5). (That topic will be the focus of Chapter 7.)

2.2.1 With better precision, through flow sensitivity

Although Jones and Lipton [39] proved that it is impossible to construct a fully automated sound method of information-flow policy enforcement that is also *complete*—does not reject any secure programs—research in this area has steadily improved the *precision* of methods, so that they accept more and more secure programs. These approaches thereby seek to retain greater scope for automation and pose less practical difficulties than complete

methods like relational Hoare logics [14, 15] and self-composition [7, 96]. (Needless to say, proving *soundness*—rejection of all insecure programs—is broadly treated as compulsory.)

This thesis will focus on *security type systems* for achieving *language-based security*, an idea first proposed and demonstrated by Volpano et al. [99]—at the time, as a reformulation of a security analysis method (described by Denning and Denning [25]) to make it amenable to a soundness proof. In their security type system, each variable in the system was given a *security type* that corresponds exactly to a classification (there called "security class") that does not vary over time. In this way, a violation (e.g. assignment $l := h$, from a sensitive source h of type High to an untrusted sink of type Low) can be identified instantly just by comparing the security types of the variables involved in a given command.

A body of work surveyed by Hunt and Sands [36] has focused on gaining precision through greater *flow sensitivity*—in short, the ability to base security judgements on things that happen before or after a given point in the program (i.e. sensitivity to position in the control flow [19]). Consider for example the program $l := h; l := 0$ against an attacker who can only observe the final value of each variable (thanks to Hunt and Sands [36]). A flow-insensitive analysis (like that of Volpano et al. [99]) would reject the assignment $l := h$ as insecure—in effect, asserting *perfect recall* [34] of l's entire history, despite the attacker only observing l's final value. To allow the $l := h$ assignment in these situations, flow-sensitive security type systems "allow the type of a variable to float" [36] by having typing environments track the sensitivity of the data in each variable (e.g. $\Gamma :: \mathit{Var} \Rightarrow \{\mathsf{High}, \mathsf{Low}\}$) as the analysis progresses through the control flow of the program. They then only enforce at the end of the program that untrusted sinks (like l in our example) no longer contain secrets (e.g. for some final typing environment Γ', that $\Gamma'\ l \neq \mathsf{High}$).

Clearly this kind of reasoning, on account of being sensitive to state, implicitly and broadly assumes both (1) the *stability* of that state—that is, the absence of changes to the values in the locations involved—and (2) that no other parties are able to read from any relevant locations at intermediate points of the program. It is important to note that while such assumptions are perhaps reasonable for sequential programs, they will come under threat upon introducing concurrency (in Section 2.2.3). This caveat will apply to any other improvements in precision that rely on tracking or making use of extra information about the state of the program—such as the "context sensitivity" (using knowledge about the calling context) and "object sensitivity" (using knowledge about the object hosting the current method, in an object-oriented environment) offered by the JOANA tool [89].

2.2.2 With mixed-sensitivity reuse, through value dependence

We now turn to the emergence of confidentiality proof techniques precise enough to allow reuse of locations for data of differing sensitivity levels at different times. This requires judgements of security to be *value dependent*, which in turn means analyses must be flow sensitive (discussed Section 2.2.1) enough to track any runtime changes to those values. (Recalling the example of Figure 1.2, to prevent assigning the data in `workspace` to the

wrong customer, the assignment would likely be judged secure or not depending on the value of `current_customer`, which therefore must be tracked by a flow-sensitive analysis.)

Zheng and Myers [102, 103] were the first to present a formal soundness proof for a security type system with judgements dependent on values that can *change* at runtime (not merely differ between runs, as achieved by Tse and Zdancewic [97] shortly beforehand). Subsequent formally proved-sound security type systems with value dependence include those for the Fine [92] and F* [93] languages, and the dependent information-flow types of Lourenço and Caires [51]—these each differed in how they map values to sensitivity levels. Costanzo and Shao [23] showed furthermore that proving a value-dependent notion of security is just as possible using a program logic that tracks assertions about state, in the usual fashion for separation logics [72, 77] and other Floyd–Hoare-like logics [29, 35].

To analyse mixed-sensitivity reuse of shared memory in concurrent programs, Murray [64] coined the term *control variables* to describe those variables in memory, on whose values the *value-dependent classification* of other variables depends. Security typechecking for the resulting *concurrent value-dependent noninterference* property in Murray et al. [67] (which extended the security type system of Mantel et al. [54], and which I will extend further in Chapter 4) involved tracking abstractions of both:

- dynamic classifications based on memory (as a set of predicates over variables that, if all true, imply the variable contains Low-sensitivity data), and
- the current state of all memory (as a set of predicates over variables that are all "currently known" to be true).

This was in contrast to tracking only static classifications of data held by each variable (as done by Mantel et al. [54], Hunt and Sands [36] in the manner described in Section 2.2.1).

It is important here to note that as these analyses rely on tracking some abstraction of state, this once again relies on the stability of that state—which (as pointed out in Section 2.2.1) will come under threat from concurrency in Section 2.2.3. The difference here from Section 2.2.1 (and Mantel et al. [54]) is that, as judgements now depend on real values, the security type systems are now obliged to track some abstraction of the actual values computed by the program, and not just their security levels. (Recalling the example of Figure 1.2 as discussed in Section 1.1.2, functional-correctness issues that impact on the coordination of mixed-sensitivity reuse are thereby security critical.) For instance, Murray et al. [67] allows tracking of values in any shared variable (not just control variables), as they might in future influence the value of a control variable (and thereby any security judgements dependent on them). Section 2.2.3 will elaborate on how analyses formalise assumptions of stability needed for flow sensitivity in the face of concurrency.

2.2.3 With concurrency, done compositionally

This section will review the literature on how to handle the impact of concurrency on proving noninterference, which implies a trade-off between the responsibility of the scheduler, and that of the threads it is scheduling. In short, this literature will show that for

choices of scheduler that allow race conditions, security analysis for each thread is obliged (1) to prevent timing leaks relative to the notion of *internal time* defined by the scheduler, and (2) to quantify over the potential impact of scheduling decisions on shared memory. I then elaborate on the approach in the literature followed by this thesis, of using assume–guarantee reasoning (introduced in Section 1.2.1) as a compositional way to contractualise the race-freedom of the inter-thread interactions that are permitted by such a scheduler.

A contract with the scheduler

As was illustrated by the example of Figure 1.1, a key goal is to prevent internal timing channels being converted into storage channels (via race-prone access to shared memory); the choice of scheduler matters because internal time is scheduler relative.

In their seminal work on language-based noninterference in concurrent programs, Smith and Volpano [88] assumed an *angelically* [38] (or Floyd-style [28]) *nondeterministic* scheduler that always (if possible) chooses an interleaving that avoids any information leak. However, Smith and Volpano recognised that their noninterference property would not be preserved when refining away that nondeterminism to choose a particular scheduler—an instance of *refinement paradox* (a term attributed to Jacob [37] by Morgan [62]).

Subsequent works therefore aimed to prove noninterference for concurrent programs when executed with particular classes of schedulers. While some targeted common classes of scheduler implementation (such as uniform probabilistic [98, 100] and round robin [78]), others sought more "scheduler-independent" verification techniques. Such techniques aim to permit wider classes of schedulers, that are instead characterised primarily by security-relevant requirements ranging from active participation to complete distrust.

On the "active participation" end of the spectrum, Barthe et al. [9, 10] presented an approach that is sound assuming a class of *security-aware* scheduler, which is aware (via program annotations) of whether a thread is on a secret-dependent control flow path, and disallows it to be interrupted by any thread that might race with its subsequent effects on race-prone memory. With the scheduler thereby actively controlling the notion of time to hide timing leaks, the security analysis for each thread need not be timing sensitive.

On the "complete distrust" end of the spectrum, Zdancewic and Myers [101] instead required threads never to write–write or read–write race on shared state. As this removes any means of propagating a timing leak to storage, the scheduler's behaviour and notion of time become largely irrelevant to the security analysis for each thread. However, this also implicitly rules out any asynchronous communication via shared memory, thus restricting analysis to programs that use only message passing-based synchronisation primitives.

In between are approaches that do not impose such an active role on the scheduler as Barthe et al. [10], but do allow race conditions; these include the "strong low-bisimulation"-based approaches [53, 54, 67] pioneered by Sabelfeld and Sands [80], that are followed by this thesis. This makes it possible to model the kind of inherently race-prone asynchronous communication underlying the implementation of synchronisation mechanisms in shared

memory. These works have in common that they impose the following kinds of scheduler-aware requirements on the program threads' security analyses:

1. The program threads are now responsible for not propagating any timing leaks to storage via race conditions; this largely comes down to the treatment of threads that have secret-dependent control flow. While some approaches ban such threads from writing to certain race-prone parts of memory after secret-dependent control flow (e.g. Mantel and Sudbrock [53], Stefan et al. [90]), other approaches instead ensure the alternative control-flow paths always take the same time (e.g. Sabelfeld and Sands [80], Mantel et al. [54], Murray et al. [67])—the latter must be sensitive to the notion of internal time as defined by the granularity of scheduling decisions.

   The various approaches to secret-dependent control flow in the literature on noninterference for concurrent programs will be the focus of Section 2.2.5.

2. Furthermore, for the security analyses of program threads to be sensitive to the impact of other threads' interference with the shared memory, these analyses may need to account for the possible choices made by the scheduler. Such works require these analyses to exhibit some variant of *strong low-bisimulation* [80] to witness noninterference: Whereas *low-bisimulations* (first applied to proving noninterference by Focardi et al. [30]) enforce observational equivalence as seen by a "low" user, *strong* low-bisimulations (as adapted for concurrency by Sabelfeld and Sands [80]) quantify over all possible scheduling decisions at every step of the program.

   The resulting problems of scale are addressed in the literature with **contracts between the threads**, which is the focus of the following section.

Finally, the scheduler is typically required (explicitly or implicitly) to satisfy a noninterference property showing that its scheduling decisions (or their probability distribution [80, 53]) for leak-prone threads are not dependent on secret information. For example, Sabelfeld and Sands [80] define schedulers as functions from non-sensitive information only, whereas Mantel and Sudbrock [53] formalised a purge-based noninterference property defining a class of "robust schedulers" that allow secret information to taint only decisions about non–leak-prone threads. Karbyshev et al. [42] have also allowed scheduling decisions to become tainted by secret information, but instead ensure (via typechecking) that the program has no race-prone effects on attacker-visible resources[1] until it has reset the scheduler state using a designated primitive. Even if not defined formally (as in Mantel et al. [54], Murray et al. [67]), an assumption that scheduling decisions (or environmental noise, in the case of Tedesco et al. [95]) do not inject any secrets is implied wherever there is a use of strong low-bisimulations that only ever demand a comparison between the *same* scheduling decisions (resp. environmental noise [95]) for observationally equivalent states.

[1]The type system of Karbyshev et al. [42] knows which of these effects are race prone by tracking the transfer between threads of fractional permissions [16] over those resources; as this transfer of ownership is between threads and not sensitivity levels, this should not be confused with mixed-sensitivity reuse.

Contracts between the threads

Approaches to concurrent program noninterference based on strong low-bisimulation (as just described), for all their strengths, commit the program analysis to quantifying over all possible scheduling decisions **and their resulting inter-thread interactions**.

Finding a way to reason over inter-thread interactions that is also *compositional* across an arbitrary number of *threads as software components* (as motivated in Section 1.1.3) is a famously nontrivial problem. For example, the classic Owicki–Gries technique [74] (albeit for analysis of program functionality, not confidentiality) reduced the effort from exponential to quadratic in the number of threads. However it is not compositional because it requires knowledge of the code for all of the threads in the system; thus, the analysis of any one thread is fragile to changes made to the code of other threads.

The approach followed by this thesis to managing the inter-thread interactions, which is explicitly geared towards compositionality, is the *rely–guarantee reasoning* of Jones [40] as adapted by Mantel et al. [54] (there called *assume–guarantee*) to apply to confidentiality. Recall (from Sections 1.1.1, 2.2.1, and 2.2.2) that the concurrency of access to shared state has two effects on the precision of formal confidentiality analyses:

1. **It shatters the assumption of stability on which most reasoning rests.** Reasoning that is sensitive to state implicitly depends on the stability of that state—that is, the absence of changes to the values in the locations involved.

   This first concern is not restricted to analyses of information-flow security, but (as pointed out in Section 2.2.2) such stability is needed for flow-sensitive analysis, which is critical to discharge value-dependent notions of security for mixed-sensitivity reuse.

2. **It also shatters any assumptions that intermediate values cannot be read.** Temporarily housing secrets in locations relies on no other parties being able to read those locations at intermediate points of the program.

   This second concern is much more specific to information-flow security.

Both assumptions (often made implicitly when analysing sequential programs) are in essence violated by actions that can be taken by other threads whose executions are interleaved with the one under analysis, when they share access to common state.

The general intuition for this adaptation of rely–guarantee to concurrency is that it **contractualises threads' freedom from each of these two effects**. To this end, Mantel et al. [54] proposed and demonstrated restricting the interleavings that must be considered during program analysis of each thread, by using explicit annotations of these assumptions in the form of two *access modes* mirroring the two concerns named above:

1. "**No write** x" is just a standard rely–guarantee condition [40], specifying that the contents of location x are unchanged by a step of interference $\sigma \rightsquigarrow \sigma'$ between two states. Its specification broadly takes the form, paraphrased from Grewe et al. [33]:

$$\sigma \rightsquigarrow \sigma' \longrightarrow \sigma.x = \sigma'.x$$

This serves in a roughly similar capacity to proving "interference freedom"[2] in the Owicki–Gries method [74]—that is, that any assertions made at a program point are not falsified by statements in the code of the other threads.

2. "**No read** x" asserts that any step of interference $\sigma \rightsquigarrow \sigma'$ taken by another thread is independent of the contents of location x. In that sense, it is itself a noninterference property that treats the original contents of x as confidential, with respect to the entirety of the destination state treated as an untrusted sink.

   Grewe et al. [33] specified this in a form equivalent to the following: Rewriting x with an arbitrary value v either has no effect on the destination, or its only effect is to change the value of x to v in the destination. Paraphrased here from Isabelle/HOL:

$$\sigma \rightsquigarrow \sigma' \longrightarrow \forall v.\ (\sigma[x \mapsto v] \rightsquigarrow \sigma') \vee (\sigma[x \mapsto v] \rightsquigarrow \sigma'[x \mapsto v])$$

   For example, an assignment $x := 3$ satisfies "no read x" (via the first disjunct, as it overwrites any v with 3); so does the assignment $y := z$ (via the second disjunct).

   Unlike the "no write" mode, the "no read" mode has no direct analogue in Owicki–Gries' method [74] or standard rely–guarantee [40]. As expected for a noninterference property, it is a 2-safety hyperproperty: A counterexample cannot merely exhibit some single interference step $\sigma \rightsquigarrow \sigma'$, but must rather exhibit the pair of that step with a violating interference step: $\sigma[x \mapsto v] \rightsquigarrow \sigma''$, where $\sigma''.x \neq v \wedge \sigma''.x \neq \sigma'.x$. Thus it cannot be encoded with a standard rely-guarantee condition, which can only make assertions about the state before or after a single given step of interference.

Mantel et al. [54] then demanded, as the witness to noninterference for each thread, a strong low-bisimulation *modulo modes*: These are not required to quantify over interference that would disobey any access modes (annotated as) active at a given program point.

In allowing security analysis to be relative to such assumptions, Mantel et al. [54] imposed two side conditions needed for their compositionality: a (non-compositional) **global** one showing that each thread's assumptions are always guaranteed by all the other threads, and a **local** one showing that each thread always complies with its own guarantees. Mantel et al. [54] provided type systems for the local side condition and security property, but proposed that the non-compositional global side condition be met by a non-compositional *may happen in parallel* analysis (e.g. Masticola and Ryder [56]). Mantel et al. [55] subsequently proposed a solution using a reachability analysis making use of dynamic pushdown networks, and Askarov et al. [5] a solution using dynamic monitoring.

Murray [64] adapted the above principles further to support mixed-sensitivity reuse of shared memory; however, Murray et al. [67] (like Mantel et al. [54]) developed proof techniques only for discharging the local obligations. Discharging the global side condition would complete a method of verifying concurrent programs with mixed-sensitivity reuse; therefore, the first contribution of this thesis (in Chapter 4) is to provide an instantiation of

[2] No direct relation to the noninterference concept of Goguen and Meseguer [32].

Murray et al. [67] wherein the global side condition is proved as an invariant of a language's mutex locking primitives, whose semantics are based on shared memory interactions.[3]

2.2.4 With both mixed-sensitivity reuse and concurrency

As I noted in Section 2.2.3, Murray [64] and Murray et al. [67] added support for programs with mixed-sensitivity reuse to the approach of Mantel et al. [54] to proving noninterference for concurrent programs; it is these works' approach which I follow and extend in this thesis, so as to prove noninterference for mixed-sensitivity concurrent programs.

The work described in this thesis is both contemporary with and succeeded by a number of related works that, likewise, concern program verification techniques for proving noninterference properties compositionally for mixed-sensitivity concurrent programs.

Firstly, as I noted in Section 1.3.1, the CoVern logic of Murray et al. [68] extends both the underlying theory of Murray et al. [67] and the work of Chapter 4 in this thesis. These extensions add support for proving *shared data invariants*, about variables protected by the mutex locks in the `While` language of Chapter 4, that may be necessary to prove noninterference for a given concurrent program. This allows the removal of runtime checks of such shared data invariants of the kind made by the case study I present in Chapter 6.

Subsequently, SecCSL (Security Concurrent Separation Logic) by Ernst and Murray [27] presented an evolution of the ideas in the CoVern logic, instead applying more directly the natural strengths of concurrent separation logics [17, 73] to enable compositional noninterference proofs for a wider range of mixed-sensitivity concurrent programs:

1. Separation logics typically already track the values at memory locations—a feature of both the security type system of Murray et al. [67] extended in Chapter 4, and the CoVern logic—which (recall from Section 2.2.2) is necessary to prove the value-dependent security properties needed for programs with mixed-sensitivity reuse.

2. Unlike CoVern and Murray et al. [67], however, and like other separation logics, SecCSL tracks the locations themselves as pointer arithmetic expressions, making it possible to support programs with indirect addressing via pointers and arrays.

Also related is Veronica by Schoepe [81], Schoepe et al. [82], a program logic which, in addition to supporting compositional noninterference proofs for mixed-sensitivity concurrent programs, also adds support for "delimited release"-style declassification policies [79]. In doing so, this work is remarkable for reconciling several features of proof precision; another such feature is a proof rule that allows programs with secret-dependent conditional branching, under restrictions similar to those I present in Chapter 7 in my extension to the security type system of Murray et al. [67] and Chapter 4. Finally, they present a new approach to noninterference proof compositionality, based on "decoupling" functional correctness reasoning from security reasoning: In contrast to mandating the use

[3] By way of preliminaries, Chapter 3 (specifically Section 3.1) will present formally the requirements posed by Murray et al. [67] that need to be met by such an instantiation.

of assume–guarantee contracts or separation logic assertions, VERONICA instead allows the use of Owicki–Gries-style annotations [74] to express the security-critical expectations of functional correctness held by each thread of others; VERONICA's logical judgements then establish the security property assuming these functional correctness assertions hold, so that they can then follow from some completely security-agnostic compositional analysis of that functional correctness. They then provide two such analysis implementation backends in their Isabelle/HOL formalisation: one based on the Owicki–Gries method as implemented by Nieto [69], Nieto and Esparza [70], and the other based on rely–guarantee.

2.2.5 With secret-dependent control flow, despite concurrency

Techniques for proving noninterference in the literature have been obliged to treat secret-dependent control flow carefully: Differing observable effects of the alternative control flow paths constitute an *implicit flow* of the secret, even if there is never any direct assignment (i.e. *explicit flow*) of a secret value to an observable location. As implicit flows can include timing leaks—when the timing (but not the value) of some observable effect differs—special care must be taken whenever there is concurrency of access to shared locations, which (as illustrated in Section 1.1.1) can convert such timing leaks into storage leaks.

In Section 2.2.3 I reviewed various ways of balancing the responsibility between the scheduler and the threads for preventing timing leaks—we now shift focus to how various such choices in the literature affect the treatment of secret-dependent control flow. The first few approaches I will present allow secret-dependent control flow, while avoiding the need for a timing-sensitive security analysis; we then move on to works whose choices make a formal treatment of time unavoidable, if secret-dependent control flow is to be allowed.

Allowing secret-dependent control flow while avoiding timing-sensitive analysis

Zdancewic and Myers [101] and Barthe et al. [10] (discussed in Section 2.2.3) both allow looping (via recursion [101], resp. **while**-loops [10]) and **if**-conditional branching to be secret dependent. Notably, both works avoid any timing-sensitive analysis, despite permitting threads to have observable effects after executing secret-dependent control flow: As Zdancewic and Myers [101] disallows any race conditions involving writes to shared memory, the precise timing of any thread's interaction with the shared memory is irrelevant; in contrast, recall that Barthe et al. [10] has the scheduler in effect adjust the notion of time on-the-fly, to ensure that the time taken regardless of control flow path (judging from the timing of observable effects afterwards) is indistinguishable to other threads.

Other approaches in the literature avoid security analysis having to be timing sensitive by restricting the kinds of observable effects a thread can have after any instance of secret-dependent control flow. This approach is taken by Mantel and Sudbrock [53] (as enforced by a security type system), Karbyshev et al. [42] (also by a security type system, but only for observable effects that it knows would be prone to race conditions), and Stefan et al. [90] (as enforced by an execution monitor), all of which allow secret-dependent **if**-

conditionals and **while**-looping. The approach they take ensures that any writes to an observable location cannot have their timing (relative to other such writes) be dependent on any secret; meanwhile, race-prone writes to shared locations not considered observable by the security property (perhaps dedicated to the transmission of secrets) are permitted.

When timing-sensitive analysis is unavoidable, it is relative to the scheduler

In contrast to the situations just described, analyses may wish to allow threads to have observable effects after secret-dependent control flow, without banning observable race conditions or having the scheduler actively manipulate the notion of time. In this case, there is no other choice but to have the security analysis be sensitive to the "time" taken by those paths of control flow. Recalling from Section 2.2.3 that internal time is scheduler relative, any such approach (to preventing storage leaks by eliminating internal timing leaks so precisely) will depend on the notion of time as defined by the choice of scheduler.

Works by Agat [2], Sabelfeld and Sands [80], Köpf and Mantel [46], and Mantel et al. [54] presented timing-sensitive security typing rules for secret-dependent **if**-conditionals (see also Chapter 7 of this thesis, and the VERONICA logic [81, 82] for an example contemporary with this thesis). These works required (and in some cases [2, 80, 46] transformed) the alternative control flow paths to take the same amount of "time", as measured in the evaluation steps of the programming (or assembly) language's semantics.

The remainder of this section will explain that, when used for threads of concurrent programs, such analyses will have to assume that scheduling is either (1) based on real time on a simple hardware platform, with evaluation steps taking a fixed amount of real time; or, (2) based on some other notion of time like the number of instructions executed [91], provided this can be measured in a reliable manner. Otherwise, banning secret-dependent control flow (e.g. Molnar et al. [61]) may be the most appropriate solution.[4]

First, if the scheduler makes decisions based on *real* time, then internal time *is* real time; to analyse the running time of control flow paths, the program verifier must then have reliable and accurate information about the real timing behaviour of the hardware platform. For simple platforms, real timing behaviour might be predictable enough to model accurately with a deterministic semantics—such as in Dewald et al. [26] for AVR.

However, typical (superscalar) processing architectures present complex microarchitectural features, exhibiting undocumented (often exploitable [31, 45]) nondeterministic real timing behaviour. Thus, banning secret-dependent control flow outright—preventing *all* implicit flows, including timing leaks—is fairly common; variants of noninterference based on the *program counter security* (or *pc-security*) notion of Molnar et al. [61] enforce this by explicitly treating control flow (as captured by the program counter) as an observable. Note also that, although the security property of Murray et al. [67] used by this thesis (see Chapter 3) is sensitive to time as modelled by evaluation steps, the security type system presented in that paper (as well as this thesis' contributions of Chapters 4 through

[4]That is, if the need for a timing-sensitive analysis cannot be avoided by ruling out race conditions between threads or using a security-aware scheduler, as described under the previous subheading.

6 that are based on it) also effectively bans secret-dependent control flow, by imposing a pc-security–like extra requirement on top of the original security property.

Finally, to prevent the internal timing leaks that result from schedulers that make decisions based on (unpredictable) real time, Stefan et al. [91] proposed *instruction-based scheduling*, which instead makes decisions based on time as counted by the number of hardware instructions "retired" (after being executed). As implementations of such schedulers have in practice been found to suffer from nondeterminism in the retired instruction counts (retrieved using hardware performance counters), timing analyses based on instruction-based scheduling would need to rely on the removal of such nondeterminism; Cock et al. [21] evaluated the extent to which they can result in measurable information leaks.

2.3 Compiler verification: Proving property preservation

Apart from being about proving certain (particularly targeted) kinds of confidentiality properties, this thesis is also about proving that a compiler preserves such properties. This section will take the following approach to the literature on compiler verification: It starts (in Section 2.3.1) with large-scale verified compiler projects that have succeeded in preserving the standard (safety, liveness) properties down to low-level (assembly, bytecode) languages, then moves on (in Section 2.3.2) to efforts to extend the scope of compiler verification to preservation of hyperproperties [20] like confidentiality, and finally (in Section 2.3.3) efforts to preserve confidentiality properties in the presence of concurrency. Chapter 3 will then provide (specifically in Section 3.2) a formal presentation of the set of requirements, drawn from background work, that this thesis will demonstrate can be met by a compiler so as to preserve noninterference for mixed-sensitivity concurrent programs.

2.3.1 For single-trace properties, like safety and liveness

Research on formal compiler verification [58, 63] and its mechanisation [59] spans decades. However, the last 15 years in particular have seen several groundbreaking projects, demonstrating that it is feasible for fully fledged compilers, for practical subsets of mainstream programming languages down to assembly or bytecode, to be verified as correct **using interactive theorem proving**:

- Klein and Nipkow [43] presented Jinja, a dialect of Java intended to exhibit its core features, with a compiler that is verified (using Isabelle/HOL [71]) to preserve the semantics and well-typedness of Jinja programs down to JVM bytecode. Lochbihler [49, 50] later did the same for JinjaThreads, an multithreaded extension of Jinja.
- Leroy [48] presented CompCert, the first C compiler formally verified (here using the Coq [94] proof assistant) to preserve program semantics from source (a "verifiable subset" of C) down to hardware assembly—here PowerPC, ARM, RISC-V, and x86.

- Kumar et al. [47] then presented CakeML (a subset of Standard ML [60]) with a compiler proved (using the HOL4 [86] proof assistant) to preserve the semantics of CakeML programs down to ARM, x86, MIPS, and RISC-V assembly code.

All of the above projects proved that their compilers preserve "program semantics" in the sense of some notion of the "observable behaviour" of the source program.

In the case of JinjaThreads, CompCert, and CakeML, the preserved behaviours are traces of various "observable events" emitted by the program, such as input–output events, or memory operations. To be precise, these three projects proved that the source program's execution *simulates* that of the compiled program;[5] therefore, the compiled program is a *refinement* of the source program, because it only produces traces that the source program could have produced. A simulation relation then serves as a *witness* to that refinement.[6]

As the traces preserved by the JinjaThreads, CompCert, and CakeML compilers can be of either finite or infinite length, this automatically implies that they preserve all safety (finite-trace counterexample) and liveness (infinite-trace counterexample) properties of the source programs that are defined in terms of the kinds of events captured by the traces.

2.3.2 For multiple-trace hyperproperties, like noninterference

The previous section talked about compilers verified to preserve traces of certain "observable" behaviours, and thereby safety and liveness properties that are defined in terms of those behaviours. However, as I explained in Section 2.1, noninterference properties are 2-safety hyperproperties [96, 20], because they make assertions about pairs of execution traces—rather than about individual execution traces—for a given program. Thus, in contrast to the previous section, fully fledged source-to-assembly (or bytecode) compilers verified to preserve hyperproperties (beyond the standard 1-safety and 1-liveness properties) are only now beginning to emerge. I now narrow in on such efforts in the literature, that preserve confidentiality (hyper)properties in particular (but defer discussing adaptations to concurrency and other environmental interference to Section 2.3.3).

Beyond the seminal work of Barthe et al. [6, 8], the line of work most relevant to this thesis is that conducted (concurrently with this thesis) by Barthe et al. [12], wherein they achieved the remarkable result of proving that a modification of the CompCert C compiler [48] 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) [11] originally intended for application to the Jasmin compiler [3]. Although not targeting programs with concurrency or mixed-sensitivity reuse (as this thesis does), CT-simulation shares in common with the refinement notions used by this thesis

[5] Note that the compiler verification literature, from Leroy [48] onwards, tends to refers to this direction as "backward simulation". This is not to be confused with the "backward simulations" of concurrency verification [52] and data refinement [24, 18], where the refined program instead simulates the original, and where simulation proceeds from the end of the program back to the beginning.

[6] In the case of JinjaThreads, Lochbihler [49, 50] went even further and exhibited a bisimulation relation as witness—an even stronger result than was strictly needed to establish refinement.

(from Murray et al. [67], which we recall in Chapter 3, Figure 3.2), that it in essence rests on a simulation diagram that is cube-shaped, as it must preserve a 2-safety hyperproperty.

I submit that Barthe et al. [12] broadly validates the argument we made in Sison and Murray [85], 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" [12]. (I will present our decomposition principle from Sison and Murray [85] in Chapter 3, Figure 3.3, as a preliminary to be used in Chapter 5.)

Furthermore, like the work of this thesis, CT-simulation pursues a timing-sensitive security property, albeit for different reasons: Whereas Barthe et al. [11, 12] explicitly aims at the prevention of external timing channels, the property of Murray et al. [67] is sensitive to an internal (scheduler-relative) notion of time primarily to prevent the emergence of storage channels (as was discussed in Section 2.2.3).

As a final note: Although Patrignani and Garg [75] proved that a strict notion of *trace-preserving compilation*—in which every single entry (observable behaviour) in the trace must be preserved—is sufficient to preserve all hypersafety properties (k-safety hyperproperties [20] for all finite k), a main limitation of this definition is that it permits no reordering, addition, or removal of observable behaviours. However, compilers routinely introduce new behaviours that, for timing-sensitive notions of confidentiality, need to be considered observable. Barthe et al. [12] consider the example of compiling some program "**if** c **then** a **else** b **fi**" to "**if** $\neg c$ **then** b **else** a **fi**", where the new negation operation "$\neg$" constitutes an additional observable behaviour. The strict trace-preserving compilation defined by Patrignani and Garg [75] is not applicable to this compilation; in contrast, both CT-simulation and the notion of refinement used in this thesis allow adding such new observable operations (in this thesis, as long as they modify no shared memory).

2.3.3 For noninterference with concurrency

I have so far covered related work verifying that a number of source-to-assembly (or bytecode) compilers for dialects of mainstream programming languages preserve standard (1-safety, 1-liveness) properties, as well as a recent example of the same feat for confidentiality (a 2-safety hyperproperty) by a modification of the CompCert compiler [12].

The task of preserving confidentiality becomes much more difficult again when introducing concurrency—in this section, we review related work that demonstrates these difficulties. We divide the following discussion of concurrency's impact into that on each of the "no read" and "no write" assumptions (discussed in Section 2.2.3), which were available implicitly in reasoning about sequential programs, but are violated by concurrency.

Compilers have to worry about concurrency making things readable

Typical optimising compilers freely add and remove behaviours not considered observable to the user. However, concurrency obliges the shared state to be treated as observable at every program point at which the scheduler might invoke a context switch to another thread; this constitutes a form of scheduler-relative timing sensitivity. As already noted, this heightened timing sensitivity renders simple optimisations (like the compilation example at the end of Section 2.3.2 adding a "¬" negate operation) and basic code generation (like an expansion of $x := y$ into [**Load** r y, **Store** x r], even when only x's and y's contents are considered observable) as changes to observable behaviour, largely ruling out strict trace-preserving compilation [75] as a method of proving preservation of confidentiality.

Extending the seminal work of Barthe et al. [6, 8] on information-flow security type-preserving compilation, Barthe et al. [9, 10] added a concurrency semantics and proved confidentiality preservation for concurrent programs, again following a type-preserving compilation approach. As explained by Section 2.2.3, this particular work escaped the need for a timing sensitive notion of observable behaviour and its preservation, because it assumed the presence of a security-aware scheduler (at the compilation target level) that prevents any context switches that have the potential to leak sensitive information.

Assuming instead a scheduler with less active responsibilities regarding security (as elaborated on also in Section 2.2.3), Murray et al. [67] presented a notion of confidentiality-preserving refinement that has the required timing sensitivity, because it demands preservation of the contents of the shared state at every point of the program at which context switch may occur—in this case, between every possible execution step (see Chapter 3). As the shared state includes all control variables (for mixed-sensitivity reuse, as explained in Section 2.2.2) this furthermore preserves all value-dependent classifications.

Compilers have to worry about concurrency making things writable

Apart from introducing extra points at which behaviour must be considered observable, concurrency also introduces the threat of changes to the shared state, at each potential point in time that a context switch is possible. While mostly a concern for preservation of functional properties, recall from Section 2.2.2 that such functionality has security-critical impacts when security is value-dependent to admit mixed-sensitivity reuse.

In this respect, the closest related work to ours in compiler verification is that of Tedesco et al. [95], who presented a type-preserving compilation scheme that preserves noninterference that, like the one in this thesis, is based on strong low-bisimulation (introduced in Section 2.2.3)—there, intended for execution contexts that are prone to memory faults. (It is their compilation scheme that this thesis will later adapt in Chapter 5, to preservation of noninterference for concurrent programs with mixed-sensitivity reuse.)

Also closely related are notions of *robust property preservation*, of which Abate et al. [1] have mapped out a spectrum. These have now largely superseded earlier-explored notions of *fully abstract compilation* (surveyed by Patrignani et al. [76]). Both bodies of work

explored requirements for compilers aimed at preserving source program (hyper)properties in the presence of adversarial execution contexts at the level of the compilation target.

The refinement notion of Murray et al. [67] to be presented in Chapter 3 (on which compiler verification in Chapter 5 is based) deals with the threat of these changes by demanding the witness refinement relation be closed under all changes by other threads permitted by the currently active access modes (recall from Section 2.2.3, these record which parts of shared memory are assumed to be readable or writable by other threads).

In our publication Sison and Murray [85] of the work of Chapter 5, we conjectured that, because it preserves a noninterference property that quantifies over some environmental interference, the refinement notion of Murray et al. [67] is implied by the formal notion of *robust 2-hypersafety preservation* (R2HSP) on the Abate et al. [1] spectrum. However, because that property does not quantify over all environmental interference—only changes modulo modes (as in Mantel et al. [54])—we do not expect its refinement notion to imply any of the preservation notions on the Abate et al. [1] spectrum.

Chapter 3

Formal preliminaries from background work

This chapter will now give formal details on the theory of a particular background work, Murray et al. [67], which will be relevant as preliminaries to the formalisms presented in rest of this thesis. Its relation to the wider survey conducted in Chapter 2 is as follows:

1. Section 2.2 surveyed treatments of concurrency and mixed-sensitivity reuse in other program verification techniques for confidentiality; Murray et al. [67] sought to provide such a verification technique combining all of these concerns.

2. Section 2.3 covered some of the other work on verifying confidentiality preservation by a compiler for concurrent programs; Murray et al. [67] sought to provide such a preservation notion that is furthermore tailored to allow mixed-sensitivity reuse.

This chapter *expands on* a similar presentation of the same background work, which we gave in a recent publication—Sison and Murray [85]—on the contributions of Chapter 5.

First, Section 3.1 presents the *concurrent value-dependent noninterference* (CVDNI) notions of Murray et al. [67], whose verification and preservation is the objective of this thesis. This presentation differs from that given by Sison and Murray [85], in two ways:

1. I provide additional details from Murray et al. [67, 65] on the *compositionality requirements* needed for the "per-thread" CVDNI property, once proved for each thread, to compose into a "whole-system" property about the entire program.

   This will be particularly relevant to the program verification efforts of Chapter 4.

2. I present also some updates made to the Isabelle/HOL formalisation [65, 84] of these properties, that allow parameterisation of initial conditions and extra requirements.

   The ability to phrase such restrictions will allow the compiler verification and its case-study application to rely on them precisely in Chapters 5 and 6.

Then, Section 3.2 presents the *CVDNI-preserving refinement* notion put forward by Murray et al. [67], which is the objective of the compiler verification efforts of Chapter 5.

In addition to that original notion, here I present the other major contribution of Sison and Murray [85], considered outside the scope of this thesis' contributions: a decomposition principle used in Chapter 5 to prove that notion of refinement.

Finally, Section 3.3 summarises what was covered in the chapter.

3.1 Concurrent value-dependent noninterference (CVDNI)

First I present the basic elements in terms of which CVDNI is defined (Section 3.1.1). Then I present support for parameters on initial conditions and extra requirements added to the definitions of the per-thread and whole-system CVDNI properties, and the theorem by which the former composes into the latter (Section 3.1.2). Finally I present the compositionality requirements demanded as a side-condition to that theorem (Section 3.1.3).

3.1.1 Basic elements

CVDNI as proved for each thread is defined by Murray et al. [67] in terms of:

1. A binary *strong low-bisimulation (modulo modes)* relation $\mathcal{B}$ between program configurations, which serves as witness to CVDNI. In the style of other low-bisimulation–based noninterference definitions, it requires the program configurations it relates to agree on their "low"-observable portions, and demands that lock-step execution preserves that correspondence (explained in Section 2.2.3). Furthermore, it is rely–guarantee-style concurrency aware, following Mantel et al. [54], but modified to allow value-dependent classifications [64] for mixed-sensitivity reuse (see next point).

2. A *classification* function $\mathcal{L}$ that determines the "low"-observable portion of a program configuration, thus affecting $\mathcal{B}$'s requirements. The innovation of $\mathcal{L}$ as parameterised first by Murray [64], and then by Murray et al. [67] as reproduced here, is that $\mathcal{L}$ can depend on values in the program configuration itself, thus expressing dynamic and not just static classifications (explained by Section 2.2.2).

The following definitions are reproduced from the background section of Sison and Murray [85], which in turn simplified definitions from Section III-2b of Murray et al. [67].

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 **Assu**mes 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.[1] This facilitates compositional, rely–guarantee-style reasoning about such access [41, 54] (as explained in Section 2.2.3).

- *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 $\mathcal{L} :: Mem \Rightarrow Var \Rightarrow \{\mathsf{High}, \mathsf{Low}\}$, inducing a function $\mathcal{C}\mathsf{vars} :: Var \Rightarrow Var\ set$ that returns all the *control variables* of a given variable, on which its classifications depend. The set $\mathcal{C} = \{y \mid \exists x.\ y \in \mathcal{C}\mathsf{vars}\ x\}$ is then defined to contain all control variables in the system.

The notion of observational equivalence used for the *whole-system* noninterference property—the ultimate objective, to be proved for the entire concurrent program of threads—is value dependent: Low-classified variables are required to have the same value in both memories. Formally (as defined originally by Murray [64]):[2]

Definition 3.1 (Low-equivalent memories)**.**

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

To support compositionality for concurrent programs, however, the equivalence notion for the *per-thread* noninterference property—for compositional analysis of each thread—is tightened up to be *modulo modes* in the style of Mantel et al. [54] (as described in Section 2.2.3) but only for non-control variables ($x \notin \mathcal{C}$): These are required to have the same value only if they are assumed to be *readable* by other threads according to the mode state.[3] Defined more formally (again, originally by Murray [64]):[4]

Definition 3.2 (Readability of variable x, according to mode state *mds*)**.**

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

[1] There is, strictly speaking, nothing in this formalism to stop the sets of any two different modes from overlapping, but my instantiations of this theory for this thesis will in practice prevent any such overlaps from occurring (via restrictions on parameters in Section 4.1.2, and via invariants proved in Section 4.3).

[2] Note that the asymmetry of both of Definition 3.1 and Definition 3.3 referring only to mem_1 is resolved by requiring (see Section 4.1.2) that the user of the theory provide a classification function $\mathcal{L}$ that statically (i.e. always, regardless of the memory state) classifies all control variables as Low.

[3] Thus intuitively, the user of the theory should model permanent untrusted output sinks of the whole concurrent program, as variables for which $\mathcal{L}$ *always returns* Low, ungoverned by any synchronisation scheme that the attacker cannot be trusted to follow.

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

Definition 3.3 (Low-equivalence of memories, modulo the mode state *mds*)**.**

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

For this thesis, I will use notation $lc_1 =^{\mathsf{Low}}_{\mathsf{mds}} lc_2$ (from Sison and Murray [85]) to lift this Definition 3.3 of $=^{\mathsf{Low}}_{mds}$ to local program configurations, asserting also that lc_1 and lc_2 have the same assume–guarantee mode state. Additionally, I 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.

We now have almost enough definitions to state the per-thread compositional security property (deferred to Section 3.1.2). This property will assert the existence of a witness relation $\mathcal{B}$ for every possible observationally equivalent pair of starting configurations. This witness relation must be a *strong low-bisimulation (modulo modes)* (denoted by $\mathsf{strong\text{-}low\text{-}bisim\text{-}mm}\ \mathcal{B}$), 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. $(lc_1, lc_2) \in \mathcal{B}$) 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 $\mathsf{sym}\ \mathcal{B}$) 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. $(lc_1', lc_2') \in \mathcal{B}$) 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 $\mathsf{cg\text{-}consistent}\ \mathcal{B}$)—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 3.5). 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 Definitions 3.4, 3.5, and 3.6:

Definition 3.4 (Strong low bisimulation, modulo modes)**.**

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

Definition 3.5 (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 3.6 (Closedness under globally consistent changes)**.**

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

3.1.2 Parameters for initial conditions and extra requirements

For this thesis, I use a version of the theory of Murray et al. [67] that allows the developer two additional forms of customisation for the CVDNI security properties:

1. They may characterise which initial states of the system are to be considered valid, by providing as a parameter to the theory 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.

   This parameter was added in an update to the Isabelle/HOL formalisation [65] of Murray et al. [67] following its publication, and was necessary to enable the work covered in Chapter 4.

2. They may furthermore characterise additional 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.

   This second parameter was added to enable the work of Sison and Murray [85], which will be covered in Chapter 5.

When not specified, *INIT* and *EXTRA* default to $(\lambda_\ .\ \mathsf{True})$ in the following definitions, in which case they simplify to their original versions as presented in Murray et al. [67], Sison and Murray [85]. In such cases, I will drop the parameter from the property name.

With the additional parameters, the per-thread security property is then as follows:

Definition 3.7 (Per-thread compositional security, with $INIT, EXTRA$ requirements)**.**

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

The compositionality theorem of Murray et al. [67] is re-proved straightforwardly to hold regardless of the $INIT, EXTRA$ chosen. Subject to some "sound mode use" side

conditions (which in this thesis I will call "compositionality requirements", deferred to Section 3.1.3), it gives us that the parallel composition $cms :: (\mathit{ThreadPrivate} \times (\mathit{Mode} \Rightarrow \mathit{Var\ set}))\ \mathit{list}$ of com-secure program threads is itself a concurrent program that enforces sys-secure, a system-wide value-dependent noninterference property:

Theorem 3.8 (Compositionality of $\mathsf{com\text{-}secure}^{EXTRA}_{INIT}$)**.**

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

This whole-system property then asserts low-equality for all global configuration pairs reachable via pairwise execution *to the same schedule* (defined in the natural way; see Murray et al. [67, 65] for a precise definition). The special form of low-equality that it applies is modified from Definition 3.1, such that it only requires each Low-classified non-control variable $x \notin \mathcal{C}$ to be of equal value in both global configurations if the mode states of *all threads* consider x to be readable (Definition 3.2). 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 $cms'_1 =_{\mathsf{all\text{-}mds}} cms'_2 \triangleq (\mathsf{map\ mds}\ cms'_1 = \mathsf{map\ mds}\ cms'_2)$. Finally, I will use the syntax $cms[i]$ to denote the ith element in list cms:

Definition 3.9 (Whole-system value-dependent security, with *INIT* requirements)**.**

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

3.1.3 Compositionality requirements

We now turn to the *sound mode use* side condition of Theorem 3.8, whose fulfilment will form the focus of Chapter 4. This consists of a "local" and a "global" part:

Definition 3.10 (Sound mode use side-condition)**.**

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

Firstly, 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; here, precise definitions for reachable-lcs and doesnt-read(-or-modify) are left to the Isabelle/HOL formalisations [65, 84]:

Definition 3.11 (Local mode compliance)**.**

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

Then together, all threads must obey a *global modes compatibility* requirement. This 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 3.12 (Global modes compatibility)**.**

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

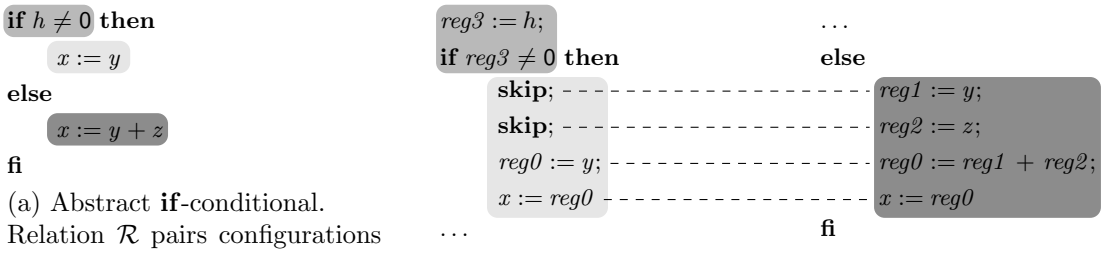
where

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

For more details and precise definitions, please refer to Section III-2(a) of Murray et al. [67], and to its Isabelle/HOL formalisation [65] or the formalisation for this thesis [84].

3.2 CVDNI-preserving refinement

Section 3.2.1 gives details on a refinement notion that is tailored to preserve the per-thread CVDNI property I presented in Section 3.1. (Both the property and its refinement notion were originally presented together by Murray et al. [67].) Subsequently, Section 3.2.2 presents a decomposition principle for proving that notion of refinement.



(a) Abstract **if**-conditional. Relation $\mathcal{R}$ pairs configurations of this program with configurations of the program in Figure 3.1b that are of the same-shaded region.

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

Figure 3.1: Excerpts from a CVDNI-preserving refinement example. Reproduced from Sison and Murray [85]—the example is originally from Murray et al. [67].

3.2.1 Original notion with cube-shaped diagram

Proof of *security-preserving refinement* (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 3.1):

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.

2. A *concrete coupling invariant* $\mathcal{I}$ that allows us to use $\mathcal{B}$ and $\mathcal{R}$ 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 3.4)—be imposed on the nominated $\mathcal{R}$ and $\mathcal{I}$, in relation to a given witness relation $\mathcal{B}$ establishing com-secure (Definition 3.7) for the abstract program. The definitions to follow are adapted from Murray et al. [67] Section V, as we presented in Sison and Murray [85]—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.

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 $\mathcal{R}$ is required to preserve the shared memory in its entirety:

Definition 3.13 (Preservation of modes and memory)**.**

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

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

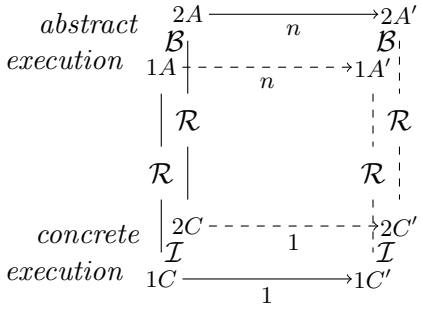


Figure 3.2: Definition and graphical depiction of refinement preservation obligation for secure-refinement (Definition 3.15). Reproduced from Sison and Murray [85]—the definition is a simplified restatement of its original formalisation in Murray et al. [67].

Regarding the closedness under changes by other threads that ensures compositionality for concurrency, on $\mathcal{I}$ we again impose cg-consistent (Definition 3.6) from Section 3.1. However, in the case of $\mathcal{R}$, 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 3.14 (Closedness of refinements under changes by others)**.**

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

The final major—and hardest—requirement for confidentiality preservation is to prove $\mathcal{R}$ and $\mathcal{I}$ 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 3.2), whose edges are configuration pairs in $\mathcal{B}$, $\mathcal{R}$, and $\mathcal{I}$. (Reducing its difficulty is the focus of the decomposition principle in Section 3.2.2.)

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

Definition 3.15 (Requirements for confidentiality-preserving secure refinement)**.**

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

The soundness theorem for confidentiality-preserving refinement by Murray et al. [67] then gives us that under the aforementioned conditions, the concrete bisimulation relation

$\mathcal{B}_{\mathsf{C}}$of $\mathcal{B}$ $\mathcal{R}$ $\mathcal{I}$ derived from strong-low-bisim-mm relations $\mathcal{B}$, refinement relation $\mathcal{R}$, and coupling invariant $\mathcal{I}$, is itself a witness strong-low-bisim-mm for the concrete program.

Definition 3.16 (Concrete bisimulation relation derived from $\mathcal{B}, \mathcal{R}$ and $\mathcal{I}$)**.**

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

Theorem 3.17 (Preservation of strong-low-bisim-mm by secure-refinement)**.**

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

3.2.2 Decomposition principle

Recent work, published in Sison and Murray [85] alongside (but outside the scope of) Chapter 5's contributions, presented an alternative way to prove secure-refinement (Definition 3.15) that obviates the need to use the cube-shaped, two-sided refinement obligation (depicted by Figure 3.2), by decomposing its concerns into:

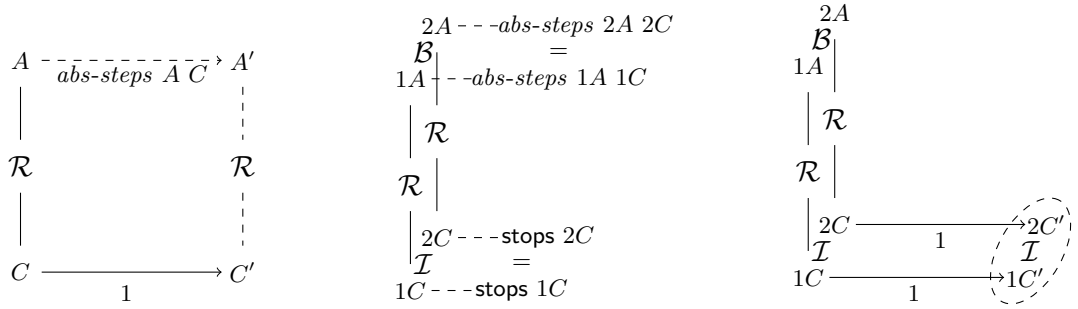
1. Proving $\mathcal{R}$ closed using a square-shaped simulation diagram (depicted by Figure 3.3a) akin to the *backward simulations* commonly used to prove semantics-preserving refinement by compilers (e.g. for CompCert [48]), and
2. A number of obligations (depicted by Figures 3.3b, 3.3c), separable from the square-shaped simulation, that prevent the introduction of *timing leaks*, *termination leaks*, and secret-dependent differences in the 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 3.3a. Deferring the separable side conditions ("decomp-refinement-safe") to afterwards, the decomposition principle is then defined formally as follows:

Definition 3.18 (Decomposed requirements for secure-refinement)**.**

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

The aforementioned side conditions on all refinement parameters, depicted by Figures 3.3b, 3.3c, are then defined formally under the predicate decomp-refinement-safe as follows:



(a) Refinement preservation for relation $\mathcal{R}$ under program execution paced by *abs-steps*. (Part of Definition 3.18.)

(b) Consistency of pacing and stopping behaviour, to prevent timing and termination leaks. (Part of Definition 3.19.)

(c) Closedness of the coupling invariant relation $\mathcal{I}$ under lockstep program execution. (Part of Definition 3.19.)

Figure 3.3: Graphical depictions of decomposed refinement preservation obligations. Reproduced from Sison and Murray [85].

Definition 3.19 (Side conditions for secure-refinement decomposition)**.**

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

The intuitive meanings of the side conditions in Definition 3.19 are:

- $\mathsf{stops}\ lc_{1C} = \mathsf{stops}\ lc_{2C}$ 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.
- $\textit{abs-steps}\ lc_{1A}\ lc_{1C} = \textit{abs-steps}\ lc_{2A}\ lc_{2C}$ 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 $\mathcal{B}$- and $\mathcal{R}$-edges in Figure 3.3c 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 $\mathcal{I}$ and prove it satisfies the condition.

Assuming the fulfilment of all of the decomposed requirements, Sison and Murray [85] provided a proof that they are a sound method for establishing secure refinement of the per-thread confidentiality property, as desired:

Theorem 3.20 (Soundness of secure-refinement-decomp)**.**

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

3.3 Summary

This chapter presented formal preliminaries from the background work (Murray et al. [67]), its Isabelle/HOL formalisation (Murray et al. [65]), and other details (from Sison and Murray [85]) considered outside the scope of, but relevant to, this thesis' contributions.

In particular, Section 3.1 presented a per-thread security property and details about its compositionality, which will be targeted by program verification (in Chapter 4). It also presented support (from Murray et al. [65]) for parameterising the property with extra restrictions, which will be relied on by the compiler verification and its case study (Chapters 5 and 6). Section 3.2 then presented a notion of security-preserving refinement for that security property, as well as a decomposition principle (from Sison and Murray [85]) for that refinement notion, which will be targeted by compiler verification (in Chapter 5).

Although I participated in all of the aforementioned publications [67, 65, 85] cited in this summary, as mentioned previously the content presented in this chapter is considered outside the scope of this thesis' contributions; the decomposition principle, however, appears in the same publication (Sison and Murray [85]) as the contributions of Chapter 5.

Chapter 4

Compositional noninterference for a `While` language with mutex locks

This chapter presents the first contribution of this thesis: a novel set of per-thread techniques that is sufficient to prove noninterference for mixed-sensitivity concurrent programs. Here I extend `While`, a generic imperative language with **while**-looping, and its proof techniques from prior work [67, 65], to support mutex lock-based synchronisation. By having the programmer explicitly specify via a parameter each lock's *footprint*—the variables it protects—I prove that the language's execution semantics maintains compositionality requirements as an invariant, and that the extended techniques remain sound.

In Section 2.2.3 (informally) and Section 3.1 (formally), I explained that Murray et al. [67] followed Mantel et al. [54] in using assume–guarantee modes to express expectations about each thread's patterns of access to shared memory, to allow verification to be compositional across each thread in the system. However, for the per-thread verification results to compose into a whole-system proof of confidentiality, the concurrent program must implement a synchronisation scheme satisfying the expectations expressed by the modes.

As these prior works [54, 67] were designed to be agnostic of the synchronisation scheme and its implementation, they exposed all compositionality requirements to the program developer. Both works gave the developer full freedom to specify their expectations as *mode annotations* in the `While` language (as in Figure 4.1a) indicating whether or not each thread accesses certain parts of the shared state at certain points in the program. This freedom, however, comes at the cost of program developers themselves having to prove that these expectations are well founded. This chapter therefore aims to show:

1. How to prove formally that *mutex locking primitives* (as in Figure 4.1b) maintain the compositionality requirements posed by Murray et al. [67], and thereby:

2. How to provide program developers a way to prove noninterference for mixed-sensitivity concurrent programs that use them, having specified each lock's footprint.

Mutex lock-based synchronisation relies on *locking discipline*: To access certain shared variables, each thread must hold a lock "governing" access to those variables. *Mutual*

```
skip /* control += AsmNoW */;
skip /* workspace += AsmNoRW */;
workspace := source;
if (control = 0) then
  low_sink := workspace
else
  high_sink := workspace
fi;
workspace := 0;
skip /* workspace -= AsmNoRW */
```

(a) Mode annotations supported by prior work [67]. These specify that (1) `control` be protected from write, as it determines the classification of `source`; and (2) `workspace` must, additionally, be protected from read, as it might be assigned sensitive data.

```
lock(big_lock);
workspace := source;
if (control = 0) then
  low_sink := workspace
else
  high_sink := workspace
fi;
workspace := 0;
unlock(big_lock)
```

(b) Enforcement with mutex locks. The programmer would specify that `big_lock`'s footprint in the locking discipline includes `control`, `source`, and `workspace`, via the lock interpretation parameter (see Section 4.1.1).

Figure 4.1: Annotation versus enforcement of assumptions on shared memory access, in the `While` language. Example program snippet adapted from Murray et al. [67].

exclusion of access then follows from the locking primitives ensuring at most one thread holds a given lock at a time. To this end, in Section 4.1, I replace the mode annotations in the `While` language of [67, 54] with mutex locking primitives, whose semantics (1) enforces this mutual exclusivity of lock acquisition, and (2) manipulates the assume–guarantee mode state to express the expectation of threads' adherence to the locking discipline.

Proving the requirements for compositionality (recalled as Theorem 3.8) then entails showing that each thread locally follows the locking discipline, and its global consequences:

1. The local compositionality requirement (local-mode-compliance, Definition 3.11) here entails showing that threads do not access any variables whose lock they do not hold. The developer may prove it by running a simple check on each of the threads: In Section 4.2, I present the adaptation of such a check from the prior work [67].

2. The global compositionality requirement (global-modes-compatibility, Definition 3.12) then follows from the threads' adherence to the locking discipline, in tandem with the language semantics' enforcement that mutex lock acquisition is mutually exclusive. Therefore, the language designer can prove it so the developer need not prove it: In Section 4.3, I present proof of its invariance for all concurrent programs written in `While` with mutex locks, when initialised not to have any locks held.

With these discharged, it suffices for a program developer wishing to prove noninterference for an entire mixed-sensitivity concurrent `While` program (sys-secure, Definition 3.9) to prove a noninterference property for each of its threads (com-secure, Definition 3.7), and to invoke Theorem 3.8. To this end, in Section 4.4, I update the `While`-language's security type system of Murray et al. [67] to replace mode annotations with mutex locks, and prove that it remains sound for proving the per-thread noninterference property.

In Section 4.5, I list publications that I coauthored whose work builds directly on the contributions of this chapter, and I acknowledge the work of others included in this chapter's presentation. Finally, I summarise the chapter's main implications in Section 4.6.

4.1 Source language: `While` with mutex locks

`While` with mutex locks (hereafter `While`) consists of the commands *cmd*:

$$\begin{aligned} \mathit{cmd} ::= {} & \textbf{skip} \mid \mathit{cmd}\ ;\ \mathit{cmd} \mid \textbf{if}\ \mathit{bexp}\ \textbf{then}\ \mathit{cmd}\ \textbf{else}\ \mathit{cmd}\ \textbf{fi} \mid \\ & \textbf{while}\ \mathit{bexp}\ \textbf{do}\ \mathit{cmd}\ \textbf{od} \mid v := \mathit{aexp} \mid \\ & \textbf{lock}(k) \mid \textbf{unlock}(k) \end{aligned}$$

For this version of `While` I introduce the mutex synchronisation primitives $\textbf{lock}(k)$ and $\textbf{unlock}(k)$; these wholly replace both the ad-hoc mode annotations and the $\textbf{await}(v)$ synchronisation primitive offered for `While` by Murray et al. [67]. After noting how `While` instantiates the underlying theory from Chapter 3, I will present the new primitives' execution semantics, which depends on the program developer supplying details of the locking discipline as a parameter (Section 4.1.1), subject to some restrictions (Section 4.1.2).

As in the previous work [67], `While` instantiates the concurrent value-dependent noninterference theory described in Section 3.1, leaving users free to supply as parameters the types of arithmetic and boolean expressions (resp. *aexp*, *bexp*) over a set of shared program-variable identifiers $v :: \mathit{Var}$ and a type of constant values *Val*. (For example, for Chapters 5 and 6, I will fix both *aexp* and *bexp* to a generic type of expressions $\mathit{exp} ::= n \mid v \mid \mathit{exp} \oplus \mathit{exp}$ over $v :: \mathit{Var}$, $n :: \mathit{Val}$ and binary arithmetic operators $\oplus :: \mathit{Val} \Rightarrow \mathit{Val} \Rightarrow \mathit{Val}$.)

Again in common with the previous work, 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 $c :: \mathit{cmd}$ being executed (understood as the current program location) is modelled as the thread-private state of the local configuration triple: $\langle c, \mathit{mds}, \mathit{mem} \rangle_{\mathsf{w}}$.

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

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

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

4.1.1 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-interp* :: *Lock* ⇒ (*Var set* × *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* ⇒ *Var set* to refer to its fst and snd projection.

Alongside encoding the mutex primitives' usual effect on control flow—most crucially, **lock**(k) should refuse to proceed meaningfully if the lock k is already held—I 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.

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 4.1 (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 4.2 (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 **lock**(k) is then given by two rules: LockAcq when lock k is available, and LockSpin when it is already held. For these, I use predicate ev_{Lock} :: *Val* ⇒ *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 $\mathsf{ev}_{Lock}(\mathsf{False}_{Lock}) = \mathsf{False}$.[1]

Apart from impacting the mode state as already specified (by Definition 4.1), attempting to acquire an available lock will succeed in the usual manner, setting the lock-variable

[1] 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.

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 (according to the local mode compliance check to be presented in Section 4.2). 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, I define the following helper:

Definition 4.3 (Mode state is consistent with holding a lock k)**.**

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

With that condition satisfied, the LockRel rule specifies that an $\mathbf{unlock}(k)$ will proceed successfully, to enact lock release on the memory and mode state as expected:

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

To ensure that the `While` evaluation semantics is defined for all possible configurations, the LockInvalid rule defines a stuttering evaluation step for attempts to $\mathbf{unlock}(k)$ that are an apparent violation of the locking discipline due to not having previously acquired the lock k. Program developers can rely on the local compliance check (see UnlockComply rule, in Section 4.2.1) to reject programs that misbehave in attempting to do this.

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

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. However, as the local mode compliance check will only ever admit 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.

4.1.2 Restrictions on locking disciplines

`While` with mutex locks imposes the following cleanliness conditions on locking disciplines and their interaction with the classification function $\mathcal{L}$ (and derived information like $\mathcal{C}$ and $\mathcal{C}$vars, all introduced in Section 3.1) to be supplied by program developers as parameters:

1. Locks cannot govern access to other locks.

   This reflects that in this proof framework, the access modes encoded by locking discipline will mandate that if some lock k were to govern access to lock k', then no thread may even attempt to acquire k' without first successfully acquiring k. This, however, would make k' entirely redundant with k.

   Instead this framework enforces, using the type signature of the *lock-interp* parameter (see Section 4.1.1), that locks can only govern access to program-variables. This will simplify the reasoning in Section 4.2 and Section 4.3.

2. Lock-variables k cannot be control variables, i.e.

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

   In the prior work [65], the definitions of doesnt-read(-or-modify) that determine local-mode-compliance (Definition 3.11) 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 will thus simplify the reasoning for local-mode-compliance in Section 4.2, because **lock**(k) and **unlock**(k) (which only access lock-variable k) cannot violate any guarantees active for program-variables.

3. The classification of all lock-variables k must be Low statically, i.e.

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

   This reflects the expectation that developers would have no desire to allow secrets to leak into the locking state. This would be problematic for two reasons.

   First, the control flow of any thread would immediately become secret-dependent following an attempt to acquire such a lock. While this is strictly speaking not disallowed by the security property (Definition 3.7), the proof techniques presented in this thesis prior to Chapter 7 will disallow any such secret-dependent control flow.

   Furthermore, the locking semantics chosen here modifies the mode state in tandem with the lock-memory state, and the security property (see Definition 3.4 of its witness in particular) prohibits mode state from ever becoming secret-dependent.

4. 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, i.e.

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

   The design of the security type system from Murray et al. [67], when I adapt it in Section 4.4 for locks, will require effectively that when a program locks some variable v, all of v's control variables' contents must also be stable, to ensure that v's security type is stable afterwards. This restriction simplifies matters by ensuring that variables and their control variables are always locked simultaneously.

5. No variable can be managed by more than one lock, i.e.

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

   This simplifies both the mode update semantics (given in Section 4.1.1) and the reasoning for global-modes-compatibility in Section 4.3, because acquiring or releasing a lock will not affect assumptions or guarantees governed by any other lock.

6. There are no "vacuous" locks, i.e. ones that would govern no variables:

$$\forall k.\ varsNoW\ k\ \cup\ varsNoRW\ k \neq \emptyset$$

   I argue that this is a reasonable expectation; it will serve to simplify some proofs in Section 4.3 by ensuring that lock-held-mds-correct and lock-not-held-mds-correct (resp. Definition 4.3, Definition 4.5) cannot both be true simultaneously.

7. The lock interpretation sets for any given lock k do not overlap, i.e.

$$\forall k.\ varsNoW\ k\ \cap\ varsNoRW\ k = \emptyset$$

   This restriction leads to the fulfilment of a guard to be asserted by the security typing rule for **unlock**(k) in Section 4.4, which I will then use to exclude a pathological case in the soundness proof (specifically in the proof of Lemma 4.23).

4.2 Local mode compliance check

To check local-mode-compliance (Definition 3.11) for program threads written in `While` with mutex locks, I adapted the local compliance check (there named a *type system for locally sound use of modes*) presented in the Isabelle/HOL formalisations [65, 33] of precursor works [67, 54]. This entailed adding dedicated judgement rules for the new locking primitives **lock**(k) and **unlock**(k), due to their replacing the mode annotation feature of

the `While` language as the sole means of making changes to the mode state (Section 4.2.1).[2] I then proved that the modified compliance check is still sound (Section 4.2.2).

This compliance check makes judgements of the following form:

$$\vdash mds\ \{c\}\ mds'$$

This judgement says that all the evaluation steps of the command c, when run with an initial mode state of mds, will comply with any guarantees present in the mode state *as it develops*; furthermore if it terminates, it will do so with the final mode state mds'.

The pre-existing judgement rules from Murray et al. [65] ensure that commands that might write (namely, an assignment $v := aexp$ to its destination v) or read shared program variables (namely, for the conditional *bexp* of an if-conditional or while-loop, or the right-hand side of an assignment $v := aexp$) do not violate any guarantees (resp. **GuarNoW**, **GuarNoRW**) present in the mode state. When such guarantees encode a locking discipline (as formalised in Section 4.1.1, with initial conditions to be given in Section 4.3.2), this in effect establishes that threads do not access variables whose locks they do not hold.

Apart from removing the handling of the mode annotations, the only other change to the pre-existing rules is the addition of a guard no-lock-mds, asserting a cleanliness condition that the mode state never records any assumptions or guarantees for lock-variables:

$$\mathsf{no\text{-}lock\text{-}mds}\ mds\ \triangleq\ \forall l\ m.\ \mathsf{Lock}\ l \notin mds\ m$$

This is expected to follow for well-initialised programs (see Section 4.3.2) from the restriction imposed on locking disciplines (enforced by the type signature of *lock-interp*, Section 4.1.1) that locks protect only program variables, not other locks (Section 4.1.2).

We defer to the Isabelle/HOL formalisations [84, 65] for further details as, apart from these changes, the rules are substantially unchanged since their prior publication [67].

4.2.1 New rules for mutex locks

The compliance check rules added for the new locking primitives are then as follows.

The LockComply rule reflects the mode state updates made by **lock**(k) (given by Definition 4.1), while also asserting the aforementioned no-lock-mds cleanliness condition.

$$\frac{\mathsf{no\text{-}lock\text{-}mds}\ mds \qquad mds' = mds \oplus k}{\vdash mds\ \{\mathbf{lock}(k)\}\ mds'}\ \textsc{LockComply}$$

In addition to doing the same for **unlock**(k) with respect to its mode state updates (given by Definition 4.2), the UnlockComply rule also has the responsibility of asserting that programs do not attempt to release locks that they are not holding—to be precise,

[2] As mode annotations previously had no dedicated compliance check rule, but rather affected the rules for every command to which they could be attached, this change has the slight advantage of decoupling the rules' management of changes to modes, from their checking of commands' compliance with those modes.

they must be in a mode state that is consistent with their having previously acquired the lock. To that end, it also asserts lock-held-mds-correct (Definition 4.3) as a precondition:

$$\frac{\textsf{lock-held-mds-correct}\ mds\ k \qquad \textsf{no-lock-mds}\ mds \qquad mds' = mds \ominus k}{\vdash mds\ \{\mathbf{unlock}(k)\}\ mds'}\ \textsc{UnlockComply}$$

4.2.2 Proof of soundness

The changes to the local compliance check require re-proving that it indeed establishes local-mode-compliance (Definition 3.11)—stated formally, that:

$$\frac{\vdash mds\ \{c\}\ mds'}{\forall mem.\ \textsf{local-mode-compliance}\ \langle c, mds, mem\rangle_{\mathsf{w}}}$$

Recall from Section 3.1 that this compliance requirement states formally that at no point reachable from a given starting configuration will a thread violate any of its own guarantees not to read or write a given location and its control variables:

Definition 3.11 (Local mode compliance)**.**

$$\begin{aligned}&\textsf{local-mode-compliance}\ lc \triangleq\\ &\forall c'\ mds'\ mem'.\ \langle c', mds', mem'\rangle_{\mathsf{w}} \in \textsf{reachable-lcs}\ lc \longrightarrow\\ &\quad (\forall x.\ (x \in mds'\ \mathbf{GuarNoRW} \longrightarrow \textsf{doesnt-read-or-modify}\ c'\ x) \wedge\\ &\qquad (x \in mds'\ \mathbf{GuarNoW} \longrightarrow \textsf{doesnt-modify}\ c'\ x))\end{aligned}$$

The soundness theorem I actually re-prove is one from Murray et al. [65] that allows a little more flexibility about the initial and final mode states. It says that if a command c passes the compliance check with initial mode state mds_1, then c still obeys local-mode-compliance even when started with an initial mds_2 with less modes[3] than mds_1—note that only the guarantee modes impact on compliance. Furthermore, the theorem promises that if c terminates, it will do so with a mode state mds_2' that has no more modes than the final mds_1' emitted by the compliance check. Stated formally:

Theorem 4.4 (Soundness of local compliance check)**.**

$$\frac{\vdash mds_1\ \{c\}\ mds_1' \qquad mds_2 \leq mds_1}{\begin{aligned}&\forall mem.\ \textsf{local-mode-compliance}\ \langle c, mds_2, mem\rangle_{\mathsf{w}} \wedge\\ &(\forall mds_2'\ mem'.\ \langle \textbf{\textit{stop}}, mds_2', mem'\rangle_{\mathsf{w}} \in \textsf{reachable-lcs}\ \langle c, mds_2, mem\rangle_{\mathsf{w}} \longrightarrow mds_2' \leq mds_1')\end{aligned}}$$

Proof. By induction over the structure of the compliance check.

Mode annotations were previously the only mechanism by which the mode state could be modified; this responsibility instead now rests solely with the new locking primitives.

[3]The less-or-equals relation $mds \leq mds'$ means all modes present in mds must also be in mds'.

Thus, the removal of mode annotations simplifies the proofs of many of the rule cases and helper lemmas, leaving intact that the remaining parts of the existing rules were already established in the previous work [65] to comply with any guarantee-modes present.

The cases for the new rules LockComply and UnlockComply are then straightforward because the locking primitives never read or modify the program-variables *Var*. This just leaves the possibility of violating a guarantee not to read or modify a *Lock* variable. But we know from the no-lock-mds guards that the mode state never records modes about *Lock* variables, so there are never any guarantees regarding them to comply with. □

4.3 Global modes compatibility

This section will present proof that global-modes-compatibility (Definition 3.12) holds as an invariant for concurrent `While` programs (Section 4.3.1) when initialised to have no locks held (Section 4.3.2). Consequently, it is sufficient for a developer to use the local compliance check of Section 4.2 to obtain the sound-mode-use condition (Definition 3.10) needed for per-thread security proofs to be compositional via Theorem 3.8.

Recall from Section 3.1 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 3.12 (Global modes compatibility)**.**

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

where

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

4.3.1 Proof of invariance

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 4.3) from Section 4.1.1, 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 4.5 (Mode state is consistent with *not* holding a lock k)**.**

$$\begin{aligned}&\mathsf{lock\text{-}not\text{-}held\text{-}mds\text{-}correct}\ mds\ k\ \triangleq\\ &\quad \forall x.\ (x \in varsNoW\ k\ \longrightarrow\ x \in mds\ \mathbf{GuarNoW}\ \wedge\ x \notin mds\ \mathbf{AsmNoW}) \wedge\\ &\qquad (x \in varsNoRW\ k\ \longrightarrow\ x \in mds\ \mathbf{GuarNoRW}\ \wedge\ x \notin mds\ \mathbf{AsmNoRW})\end{aligned}$$

This first management requirement for global configurations, regarding lock-managed variables, is then that for all locks, exactly one thread if and only if the lock is held by anybody has a mode state consistent with holding it, and all other threads have a mode state consistent with not holding it. Formally, with $\mathsf{mdss}\ gc \triangleq \mathsf{map}\ \mathsf{mds}\ (\mathsf{cms}\ gc)$:

Definition 4.6 (Lock-managed variable modes are compatible with memory)**.**

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

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 4.7 (Unmanaged variable modes are compatible)**.**

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

[4] Note that this not merely the negation of $\mathsf{lock\text{-}held\text{-}mds\text{-}correct}\ mds\ k$ (Definition 4.3)!

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 4.8 (No assumptions and guarantees on lock variables)**.**

$$\mathsf{no\text{-}lock\text{-}mds\text{-}gc}\ gc \triangleq \forall mds \in \mathsf{set}\ (\mathsf{mdss}\ gc).\ \mathsf{no\text{-}lock\text{-}mds}\ mds$$

This follows trivially from the restriction (in Section 4.1.2) that the lock interpretation parameter only permit locks to protect access to program variables (not other locks), and the 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 4.9 (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 4.6):

Lemma 4.10 (Single-step preservation of lock-managed-modes-mem-compatible)**.**

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

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 4.7):

Lemma 4.11 (Single-step preservation of unmanaged-var-modes-compatible)**.**

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

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

I 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 4.9), yield in a straightforward manner the desired global compatibility invariance theorem:

Theorem 4.12 (Mode management requirements ensure global compatibility)**.**

$$\frac{\begin{array}{c}\mathsf{lock\text{-}managed\text{-}modes\text{-}mem\text{-}compatible}\ gc \qquad \mathsf{unmanaged\text{-}var\text{-}modes\text{-}compatible}\ gc \\ \mathsf{no\text{-}lock\text{-}mds\text{-}gc}\ gc\end{array}}{\mathsf{global\text{-}modes\text{-}compatibility}\ gc}$$

4.3.2 Initial conditions

I 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 3.12) of the sound-mode-use side condition (Definition 3.10) of the compositionality theorem for our security property (Theorem 3.8).

I define the following predicate for initial memory:

Definition 4.13 (A requirement for initial memory that no locks are held)**.**

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

I then define an initial mode state $\mathsf{mds}_0 :: Mode \Rightarrow Var\ set$ that provides all guarantees demanded by the lock interpretation parameters $varsNoW, varsNoRW$ (described in Section 4.1.1) for all lock variables in the system, and makes no assumptions:

Definition 4.14 (Initial mode state mds_0)**.**

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

I am then able to show that these conditions are enough to satisfy the requirements I just showed (in Section 4.3) ensure global modes compatibility for `While`:

Lemma 4.15 (Initialising with $\mathsf{no\text{-}locks\text{-}held}, \mathsf{mds}_0$ ensures global modes compatibility)**.**

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

Proof. Theorem 4.12 obliges us to show that the mode management conditions (Definitions 4.6, 4.7, and 4.8) hold. This follows straightforwardly from all the relevant definitions. □

4.4 Security type system

Murray et al. [67] presented a security type system for `While`-language programs where the developer can use annotations to specify changes to the mode state directly. With the changes to the `While` language presented in Section 4.1, however, the mode state is instead managed implicitly by the new **lock**(k) and **unlock**(k) primitives, which for simplicity also replace the **await**(v) synchronisation primitive that was present in Murray et al. [67].

As `While` no longer supports mode annotations and **await**(v), here I remove the corresponding typing rules Anno and Await, and replace these with rules for the new **lock**(k) and **unlock**(k) commands (Section 4.4.1). Meanwhile the typing rules Seq, Skip, As$(1|2|\mathcal{C})$, If and While for the language features retained here, and Rewrite rule for rewriting the typing environment, are substantially the same as in Murray et al. [67]. Finally I prove that the security type system remains sound (Section 4.4.2).

As in Murray et al. [67], the security type system tracks three forms of information: typing environment Γ, stable variables $\mathcal{S}$, and predicate set P (all to be explained here). For convenience, I will refer to all three together as an *extended typing environment*. A typing judgement, attesting that command c is *well typed* with respect to a particular *initial* $(\Gamma, \mathcal{S}, P)$ and *final* $(\Gamma', \mathcal{S}', P')$ extended typing environment, then takes the form:

$$\vdash \Gamma, \mathcal{S}, P\ \{c\}\ \Gamma', \mathcal{S}', P'$$

The typing environment $\Gamma :: Var \rightharpoonup bexp\ set$ represents value-dependent security types

for each program variable[5] as a partial map to sets of predicates (boolean expressions) over program variables. The *bexp set* encoding the security type of a given variable encodes what is known about the sensitivity of the data stored in the variable, at that program point. The intuition is that to conclude that a variable of security type (predicate set) *ps* contains data of Low sensitivity when the shared memory state is *mem*, all of the predicates in *ps* must evaluate to True in that *mem*. Written formally:

$$\mathsf{preds\text{-}hold}\ ps\ mem\ \triangleq\ \forall p \in ps.\ \mathsf{ev}_{bexp}\ mem\ p$$

Otherwise if any evaluate to False (formally, $\neg$preds-hold *ps mem*), then the variable must be considered to contain data of High sensitivity.

The type system also tracks a pair $\mathcal{S} :: Var\ set \times Var\ set$, of the sets of variables assumed to be stable (not written to). The fst of this pair tracks variables having a mode state of **AsmNoW**, and the snd resp. a mode state of **AsmNoRW**.

Finally, the type system maintains a set of predicates $P :: bexp\ set$ that it currently tracks as True, which the type system uses (as an abstraction of *mem* state) to determine the sensitivity of data in variables according to their value-dependent security type.

With the help of the stable set pair $\mathcal{S}$, the security type system at all times restricts the typing environment Γ and predicate set P only to concern (or refer to) variables that are stable. (This is the reason that the typing environment is a partial map.) This role previously fell to Anno, and will now fall to the new security rules to be introduced below.

4.4.1 New rules for mutex locks

I now elaborate on the new typing rules that I add for **lock**(k) and **unlock**(k), before moving on (in Section 4.4.2) to a proof of their soundness.

Because I have replaced mode annotations with mutex locking primitives, variables now become stable (resp. return to being unstable) for a program precisely when that program acquires (resp. releases) a lock governing them according to the locking discipline.

The effect on $\mathcal{S}$ of acquiring (resp. releasing) the lock k is formalised as follows:

Definition 4.16 (Impact of acquisition, resp. release of lock k on stable set pair $\mathcal{S}$)**.**

$$\mathcal{S} \oplus k\ \triangleq\ (\mathsf{fst}\ \mathcal{S} \cup varsNoW\ k,\ \mathsf{snd}\ \mathcal{S} \cup varsNoRW\ k)$$
$$\mathcal{S} \ominus k\ \triangleq\ (\mathsf{fst}\ \mathcal{S} - varsNoW\ k,\ \mathsf{snd}\ \mathcal{S} - varsNoRW\ k)$$

In response to acquiring a lock k, the typing environment adds the variables governed by that lock k to the set of variables being tracked. The security types of variables previously tracked remain unchanged; newly tracked variables are given a default security type that corresponds to their value-dependent classification $\mathcal{L}$ transliterated as a set of

[5] Its full type in the Isabelle formalisation [84] also includes lock variables, but the restrictions laid out in Section 4.1.2, together with an extra guard (elided from this presentation) on the Lock typing rule, are effectively enough to exclude any tracking of lock variables; similarly for the stable set pair $\mathcal{S}$.

bexp predicates (written $\mathcal{L}_{bexp}$) that must all hold for the variable's classification (thus its data's sensitivity) to be Low. (An exception to the above is that control variables are excluded from this tracking, as the assignment typing rules As1, As2, As$\mathcal{C}$ [67] already ensure they are only ever assigned data of Low sensitivity.) Stated formally, acquiring lock k when in typing environment Γ results in a new typing environment defined as follows:[6]

Definition 4.17 (Impact on typing environment Γ of acquiring lock k)**.**

$$\begin{aligned}\Gamma \oplus k \;\triangleq\; \lambda x.\ &\mathsf{if}\ (x \in \mathsf{dom}\ \Gamma)\ \mathsf{then}\ \Gamma\ x \\ &\mathsf{else\ if}\ (x \in \mathit{varsNoW}\ k \cup \mathit{varsNoRW}\ k\ \wedge\ x \notin \mathcal{C})\ \mathsf{then\ Some}\ (\mathcal{L}_{bexp}\ x) \\ &\quad \mathsf{else\ None}\end{aligned}$$

Conversely, when releasing k the typing environment ceases tracking the variables governed by k (i.e. it only gives answers for variables that are still stable after the release):

Definition 4.18 (Impact on typing environment Γ of releasing lock k)**.**

$$\begin{aligned}\Gamma \ominus_{\mathcal{S}} k \;\triangleq\; \lambda x.\ &\mathsf{if}\ (x \in \mathsf{fst}\ (\mathcal{S} \ominus k) \cup \mathsf{snd}\ (\mathcal{S} \ominus k))\ \mathsf{then}\ \Gamma\ x \\ &\quad \mathsf{else\ None}\end{aligned}$$

Then as mentioned earlier, the new typing rules now bear the responsibility (taking over from the old mode annotation typing rule Anno) for restricting the predicate set to the variables that are stable. This trimming of the predicate set reflects the intuition that predicates previously thought to be True may become unreliable if their variables become unstable. For this I reuse a helper function from Murray et al. [67]:

Reproduced definition (Restriction of a predicate set P to those stable under $\mathcal{S}$)**.**

$$P \restriction \mathcal{S} \;\triangleq\; \{e \mid e \in P \wedge e \subseteq (\mathsf{fst}\ \mathcal{S} \cup \mathsf{snd}\ \mathcal{S})\}$$

Finally, both of the new typing rules will impose a check carried over from the Anno rule of Murray et al. [67] (itself adapted originally from Mantel et al. [54]) that disallows the locking primitives from lowering the tracked security type of any variable. For this I use a subtyping relation (also defined previously in Murray et al. [67]) between predicate sets $t, t' :: \mathit{bexp}\ \mathit{set}$, asserting that for shared memories mem where the set of tracked predicates P holds, the latter set t' holding entails the former set t holding too:

Reproduced definition (Subtyping relation between predicate sets)**.**

$$t \leq:_P t' \;\triangleq\; \forall \mathit{mem}.\ \mathsf{preds\text{-}hold}\ (P \cup t')\ \mathit{mem} \longrightarrow \mathsf{preds\text{-}hold}\ t\ \mathit{mem}$$

The aforementioned subtyping check, formally $\forall x.\ \Gamma\ x \leq:_{P'} \Gamma'\ x$, then ensures that under a new set of facts P', the tracked sensitivity of the data in any variable x cannot have dropped from High (in the old typing environment Γ) to Low (in the new environment Γ').

[6] Note that from here onwards, I will overload the $\oplus, \ominus$ notation seen earlier in Section 4.1.1.

The checks all described so far yield the typing rule for the **lock**(k) primitive:

$$\frac{\Gamma' = \Gamma \oplus k \qquad \mathcal{S}' = \mathcal{S} \oplus k \qquad P' = P \restriction \mathcal{S}' \qquad \forall x.\ \Gamma\ x \leq:_{P'} \Gamma'\ x}{\vdash \Gamma, \mathcal{S}, P\ \{\mathbf{lock}(k)\}\ \Gamma', \mathcal{S}', P'}\ \textsc{Lock}$$

For the **unlock**(k) primitive, the typing rule also enforces that if a lock k governing some control variable v is released, then no variable x tracked by the typing environment Γ has their type continuing to rely on v. This restriction must be enforced because if it were violated, releasing k would cause v (and thus all sensitivity-tracking security types relying on it) to become unstable. Stated precisely, we say that a variable v is part of what determines the security type $t = \Gamma\ x$ of some variable x, if it occurs in any of its boolean expressions p (this was called vars-of-type in Murray et al. [67, 65]):

Reproduced definition (Variables determining a given predicate set)**.**

$$\mathsf{vars\text{-}determining\text{-}preds}\ t\ \triangleq\ \{v' \mid \exists p \in t.\ v' \in \mathit{bexp\text{-}vars}\ p\}$$

The requirement just described is then phrased precisely as:

Definition 4.19 (Unlock maintains tracking stability)**.**

$$\begin{aligned}&\mathsf{unlock\text{-}maintains\text{-}tracking\text{-}stability}\ k\ \Gamma\ \mathcal{S} \triangleq \\ &\quad (\forall v \in (\mathit{varsNoW}\ k \cup \mathit{varsNoRW}\ k).\ v \in \mathcal{C}\ \longrightarrow \\ &\qquad (\forall x \in \mathsf{dom}\ (\Gamma \ominus_{\mathcal{S}} k).\ v \notin \mathsf{vars\text{-}determining\text{-}preds}\ (\Gamma\ x)))\end{aligned}$$

The typing rule for the **unlock**(k) primitive is then:

$$\frac{\begin{array}{c}\Gamma' = \Gamma \ominus_{\mathcal{S}} k \qquad \mathcal{S}' = \mathcal{S} \ominus k \qquad P' = P \restriction \mathcal{S}' \qquad \forall x.\ \Gamma\ x \leq:_{P'} \Gamma'\ x \\ \mathsf{unlock\text{-}maintains\text{-}tracking\text{-}stability}\ k\ \Gamma\ \mathcal{S} \qquad \mathsf{fst}\ \mathcal{S} \cap \mathsf{snd}\ \mathcal{S} = \emptyset\end{array}}{\vdash \Gamma, \mathcal{S}, P\ \{\mathbf{unlock}(k)\}\ \Gamma', \mathcal{S}', P'}\ \textsc{Unlock}$$

The final guard $\mathsf{fst}\ \mathcal{S} \cap \mathsf{snd}\ \mathcal{S} = \emptyset$ enforces a basic check that the stable sets are non-overlapping, which should follow from a similar restriction imposed on pairs returned by the *lock-interp* parameter (Section 4.1.2). This guard will allow me to exclude a pathological case (see Lemma 4.23) where a variable continues to have a security type (due to being **AsmNoW**-stable after the unlock) without assurance that its contents are free of secrets (due to having been hidden by **AsmNoRW**-stability before the unlock).

4.4.2 Proof of soundness

Having changed the set of rules in the security type system, we are obliged to re-prove that the security type system is sound—meaning that for a `While` program thread, it establishes the per-thread compositional security property com-secure (Definition 3.7).

The only substantial proof impact is on two main lemmas from Murray et al. [65]:

- The "preservation" lemma says that if a well-typed program takes a step, its destination is also well typed (here, relative to the same final extended typing environment). Such a property is commonly proved also for traditional (non-security) type systems.
- The "typed step capture" lemma says that well-typed program evaluation is captured by an artifact exhibiting the desired security property:[7] If a well-typed program takes a step, then a configuration considered equivalent by the initial typing environment must also be able to take a step, and both destinations are related by a *bisimulation construction* $\mathcal{B}_{\Gamma',\mathcal{S}',P'}$ defined relative to the final extended typing environment.

These main lemmas are both proved by induction over the structure of the type system, and therefore the impact is largely restricted to the removal of cases for Anno and Await, and the addition of the new cases for Lock and Unlock.

Proving these two main lemmas makes use of the following two pairs of helper lemmas about the impact on the extended typing environment of the new mutex locking primitives.

The first pair of helper lemmas establishes that the updates made by the Lock and Unlock rules preserve the *stability* of the types (i.e. the variables they depend on) in a *well-formed* typing environment (i.e. dependent only on control variables):

Reproduced definitions (Wellformedness of types, based on stability of their predicates. From Murray et al. [65])**.**

$$\begin{aligned}
\mathsf{preds\text{-}stable}\ \mathcal{S}\ ps &\triangleq \forall x \in \mathsf{vars\text{-}determining\text{-}preds}\ ps.\ x \in \mathsf{fst}\ \mathcal{S} \cup \mathsf{snd}\ \mathcal{S} \\
\mathsf{types\text{-}stable}\ \Gamma\ \mathcal{S} &\triangleq \forall x \in \mathsf{dom}\ \Gamma.\ \mathsf{preds\text{-}stable}\ \mathcal{S}\ (\mathsf{dom}\ \Gamma) \\
\mathsf{types\text{-}wellformed}\ \Gamma &\triangleq \forall x \in \mathsf{dom}\ \Gamma.\ \mathsf{vars\text{-}determining\text{-}preds}\ (\Gamma\ x) \subseteq \mathcal{C}
\end{aligned}$$

Lemma 4.20 (Lock acquisition updates preserve type environment stability)**.**

$$\frac{\mathsf{types\text{-}stable}\ \Gamma\ \mathcal{S} \qquad \mathsf{types\text{-}wellformed}\ \Gamma}{\mathsf{types\text{-}stable}\ (\Gamma \oplus k)\ (\mathcal{S} \oplus k)}$$

Proof. When acquiring a lock on variable x, its type is initialised to $\mathcal{L}_{bexp}\ x$, which could only be in terms of its control variables ($\mathcal{C}\mathsf{vars}\ x$). To know that they are stable, we rely on a lock-interpretation requirement (as mentioned in Section 4.1.2) that access to variables must be governed by exactly the same lock as all of their control variables.

The types of all other variables could not have been in terms of any of these (previously unstable) variables, and therefore are unaffected and remain stable. □

[7] In the sense that it links the type system with the property it is meant to establish, the "typed step capture" lemma here could be viewed as a security type system analogue of the "progress" theorem commonly proved for traditional type systems whose purpose is to prevent programs from getting stuck.

Lemma 4.21 (Lock release updates preserve type environment stability)**.**

$$\frac{\mathsf{types\text{-}stable}\ \Gamma\ \mathcal{S} \qquad \mathsf{types\text{-}wellformed}\ \Gamma \qquad \mathsf{unlock\text{-}maintains\text{-}tracking\text{-}stability}\ k\ \Gamma\ \mathcal{S}}{\mathsf{types\text{-}stable}\ (\Gamma \ominus_{\mathcal{S}} k)\ (\mathcal{S} \ominus k)}$$

Proof. The types of all variables governed by the lock k are dropped from the typing environment, so their stability is of no concern.

However, there may be other variables not governed by k whose types have moved away from their default $\mathcal{L}_{bexp}\ x$, and have come to rely on control variables other than their own ($\mathcal{C}\mathsf{vars}\ x$). The $\mathsf{unlock\text{-}maintains\text{-}tracking\text{-}stability}\ k\ \Gamma\ \mathcal{S}$ guard (Definition 4.19) explicitly protects the rest of these cases, by disallowing programs that try to unlock control variables when there are types that still rely on them. □

The second pair of helper lemmas establishes that the updates made by the Lock and Unlock rules to the typing environment preserve a security condition: Effectively, the tracked sensitivity of data should never exceed the classification of the variable containing it, unless that variable is currently assumed not to be readable. Stated formally:

Reproduced definitions (Type environment security condition, Murray et al. [65])**.**

$$\begin{aligned}\mathsf{type\text{-}max}\ t\ mem &\triangleq \mathsf{if}\ (\mathsf{preds\text{-}hold}\ t\ mem)\ \mathsf{then\ Low\ else\ High}\\ \mathsf{tyenv\text{-}sec}\ mds\ \Gamma\ mem &\triangleq \forall x \in \mathsf{dom}\ \Gamma.\ x \notin mds\ \mathbf{AsmNoRW} \longrightarrow\\ &\qquad \mathsf{type\text{-}max}\ (\Gamma\ x)\ mem \leq \mathcal{L}\ mem\ x\end{aligned}$$

Lemma 4.22 (Lock acquisition updates preserve type environment security)**.**

$$\frac{\mathsf{tyenv\text{-}sec}\ mds\ \Gamma\ mem \qquad \forall x.\ x \neq \mathsf{Lock}\ k \longrightarrow mem'\ x = mem\ x}{\mathsf{tyenv\text{-}sec}\ (mds \oplus k)\ (\Gamma \oplus k)\ mem'}$$

Proof. We are only worried about the case of variables x that after the lock acquisition are tracked as containing High data, but are still assumed to be readable with a value-dependent classification $\mathcal{L}\ mem\ x$ of Low. Using Isabelle we are able to discharge this proof with the `auto` tactic, unfolding all of the aforementioned definitions. □

Lemma 4.23 (Lock release updates preserve type environment security)**.**

$$\frac{\begin{array}{c}\mathsf{tyenv\text{-}sec}\ mds\ \Gamma\ mem \qquad \mathcal{S} = (mds\ \mathbf{AsmNoW}, mds\ \mathbf{AsmNoRW})\\ \mathsf{fst}\ \mathcal{S} \cap \mathsf{snd}\ \mathcal{S} = \emptyset \qquad \mathsf{lock\text{-}held\text{-}mds\text{-}correct}\ mds\ k\\ \forall x.\ x \neq \mathsf{Lock}\ k \longrightarrow mem'\ x = mem\ x\end{array}}{\mathsf{tyenv\text{-}sec}\ (mds \ominus k)\ (\Gamma \ominus_{\mathcal{S}} k)\ mem'}$$

Proof. Similarly to Lemma 4.22, we are only worried about variables x that continue to be tracked as stable but readable (**AsmNoW**) after the lock release.

As lock release can only possibly drop assumptions, this means x must have previously been tracked as **AsmNoW**, and furthermore the lemma's guards tell us x must have satisfied the security condition previously. But the only part of memory that may change is the value of lock k, so therefore neither of the still-tracked security type nor the value-dependent classification of x could have changed from before.

Finally, as mentioned in Section 4.4, the guard $\mathsf{fst}\ \mathcal{S} \cap \mathsf{snd}\ \mathcal{S} = \emptyset$ excludes a pathological case where x was permitted previously by the security condition to contain sensitive contents, due to being hidden by **AsmNoRW**-stability before the unlock.

Along these lines of reasoning, using various Isabelle tactics in mixed Isar/"apply"-style proof script, we are able to show that the security condition continues to hold. □

To state the two aforementioned "preservation" and "typed step capture" lemmas formally and detail the proof impact on them, I need a few more definitions from Murray et al. [65] that bundle the type environment security and stability conditions together with other wellformedness conditions, relative to a given mode state:

Reproduced definitions (Wellformedness and consistency of type environments)**.**

$$\begin{aligned}
\mathsf{tyenv\text{-}mds\text{-}consistent}\ mds\ \Gamma\ \mathcal{S}\ P\ \triangleq\ & \mathcal{S} = (mds\ \mathbf{AsmNoW}, mds\ \mathbf{AsmNoRW}) \wedge \\
& \mathsf{dom}\ \Gamma = \{x \mid x \notin \mathcal{C} \wedge x \in \mathsf{fst}\ \mathcal{S} \cup \mathsf{snd}\ \mathcal{S}\} \wedge \\
& \mathsf{preds\text{-}stable}\ \mathcal{S}\ P \\
\mathsf{tyenv\text{-}wellformed}\ mds\ \Gamma\ \mathcal{S}\ P\ \triangleq\ & \mathsf{tyenv\text{-}mds\text{-}consistent}\ mds\ \Gamma\ \mathcal{S}\ P \wedge \\
& \mathsf{types\text{-}wellformed}\ \Gamma\ \wedge\ \mathsf{types\text{-}stable}\ \Gamma\ \mathcal{S}
\end{aligned}$$

(Note that tyenv-wellformed was denoted by the abbreviation wf in Murray et al. [67].)

Then as mentioned before, the preservation lemma says that if a well-typed program takes a step, its destination is also a well-typed program relative to the same final typing environment. Furthermore, the step preserves the desired well-formedness conditions.

Lemma 4.24 (Preservation of welltypedness by program evaluation)**.**

$$\frac{\begin{array}{c}\vdash \Gamma, \mathcal{S}, P\ \{c\}\ \Gamma', \mathcal{S}', P' \\ \mathsf{tyenv\text{-}wellformed}\ mds\ \Gamma\ \mathcal{S}\ P \qquad \mathsf{preds\text{-}hold}\ P\ mem \qquad \mathsf{tyenv\text{-}sec}\ mds\ \Gamma\ mem \\ \langle c, mds, mem\rangle_{\mathsf{w}} \rightsquigarrow_{\mathsf{w}} \langle c', mds', mem'\rangle_{\mathsf{w}}\end{array}}{\begin{array}{c}\exists \Gamma''\ \mathcal{S}''\ P''.\ \vdash \Gamma'', \mathcal{S}'', P''\ \{c'\}\ \Gamma', \mathcal{S}', P'\ \wedge\ \mathsf{tyenv\text{-}wellformed}\ mds'\ \Gamma''\ \mathcal{S}''\ P''\ \wedge \\ \mathsf{preds\text{-}hold}\ P''\ mem'\ \wedge\ \mathsf{tyenv\text{-}sec}\ mds'\ \Gamma''\ mem'\end{array}}$$

Proof. By induction over the structure of the type system. The proofs of the surviving cases from Murray et al. [65] proceed largely unchanged.

The new cases for Lock and Unlock follow straightforwardly from Lemma 4.22 and Lemma 4.23 (respectively), and from a type environment wellformedness preservation lemma that makes use of Lemma 4.20 and Lemma 4.21 in a similar manner. □

I then prove that steps taken from configurations considered *low-equivalent by the typing environment* maintain a strong low-bisimulation (modulo modes) given by the construction $\mathcal{B}_{\Gamma',\mathcal{S}',P'}$. In this chapter, the construction remains largely unchanged from the previous work's Isabelle formalisation [65]—it will be revisited in more detail in Chapter 7. Meanwhile, the notion of low-equivalence is unchanged from Murray et al. [67]:

Definition 4.25 (Low-equivalence of memories according to a typing environment [67])**.**

$$mem_1 =_\Gamma mem_2 \triangleq \forall x.\ \mathsf{type\text{-}max}\ (\mathsf{to\text{-}total}\ \Gamma\ x)\ mem_1 = \mathsf{Low} \longrightarrow mem_1\ x = mem_2\ x$$

Lemma 4.26 (Well-typed program evaluation steps are captured by bisimulation $\mathcal{B}_{\Gamma',\mathcal{S}',P'}$)**.**

$$\frac{\begin{array}{c}\vdash \Gamma, \mathcal{S}, P\ \{c\}\ \Gamma', \mathcal{S}', P' \qquad mem_1 =_\Gamma mem_2 \\ \mathsf{tyenv\text{-}wellformed}\ mds\ \Gamma\ \mathcal{S}\ P \qquad \mathsf{preds\text{-}hold}\ P\ mem_1 \qquad \mathsf{tyenv\text{-}sec}\ mds\ \Gamma\ mem_1 \\ \langle c, mds, mem_1\rangle_{\mathsf{w}} \rightsquigarrow_{\mathsf{w}} \langle c_1', mds', mem_1'\rangle_{\mathsf{w}}\end{array}}{\begin{array}{c}\exists c_2'\ mem_2'.\ \langle c, mds, mem_2\rangle_{\mathsf{w}} \rightsquigarrow_{\mathsf{w}} \langle c_2', mds', mem_2'\rangle_{\mathsf{w}} \wedge \\ \langle c_1', mds', mem_1'\rangle_{\mathsf{w}}\ \mathcal{B}_{\Gamma',\mathcal{S}',P'}\ \langle c_2', mds', mem_2'\rangle_{\mathsf{w}}\end{array}}$$

Proof. By induction over the structure of the type system. As for Lemma 4.24, the proofs of the surviving cases from Murray et al. [65] proceed largely unchanged.

For the new Lock and Unlock cases, there are new sub-cases for the LockSpin (and LockInvalid) evaluation step. The remaining sub-cases, for the actual LockAcq and LockRel evaluation steps, are mostly adaptations of the proof for the old Anno rule, making use of the new lemmas (and re-proved Lemma 4.24) covered in this section. As for the Anno rule in Murray et al. [67], the subtyping guard $\forall x.\ \Gamma\ x \leq:_{P'} \Gamma'\ x$ serves to prevent the locking primitives from breaking the equivalence $=_\Gamma$ between memories, which must be maintained through to the destination pair of memories (specifically $=_{\Gamma'}$ for those) as a condition for their membership in $\mathcal{B}_{\Gamma',\mathcal{S}',P'}$. □

Then ultimately, what I re-prove is a more general form [65] of the soundness theorem from Murray et al. [67]. It permits any initial mode state *mds* that "yields stable types", meaning that if a variable is stable, then all its control variables must also be stable:

Reproduced definitions (Allowable initial mode states *mds*, and their corresponding initial typing environment Γ_{mds} and stable set pair $\mathcal{S}_{mds}$. From Murray et al. [65])**.**

$$\begin{aligned}\mathsf{yields\text{-}stable\text{-}types}\ mds \triangleq\ & \forall x \in (mds\ \mathbf{AsmNoW} \cup mds\ \mathbf{AsmNoRW}). \\ & (\forall v \in \mathcal{C}\mathsf{vars}\ x.\ v \in mds\ \mathbf{AsmNoW} \cup mds\ \mathbf{AsmNoRW}) \\ \Gamma_{mds} \triangleq\ & \lambda x.\ \mathsf{if}\ (x \notin \mathcal{C} \wedge x \in mds\ \mathbf{AsmNoW} \cup mds\ \mathbf{AsmNoRW})\ \mathsf{then} \\ & \quad \mathsf{if}\ (x \in mds\ \mathbf{AsmNoRW})\ \mathsf{then}\ \mathsf{Some}\ \{\mathsf{False}_{bexp}\} \\ & \quad \mathsf{else}\ \mathsf{Some}\ (\mathcal{L}_{bexp}\ x) \\ & \mathsf{else}\ \mathsf{None} \\ \mathcal{S}_{mds} \triangleq\ & (mds\ \mathbf{AsmNoW}, mds\ \mathbf{AsmNoRW})\end{aligned}$$

The security type system is sound in the sense that if a `While` program $c :: cmd$ passes it, with initial typing environment Γ_{mds} and stable set pair $\mathcal{S}_{mds}$ corresponding to an allowable mode state mds, then it is per-thread secure when started with that mds:

Theorem 4.27 (Soundness of the security type system updated for mutex locks)**.**

$$\frac{\vdash \Gamma_{mds}, \mathcal{S}_{mds}, \emptyset\ \{c\}\ \Gamma', \mathcal{S}', P' \qquad \mathsf{yields\text{-}stable\text{-}types}\ mds}{\mathsf{com\text{-}secure}\ (c, mem)}$$

Proof. As in Murray et al. [65], I use the strong low-bisimulation (modulo-modes) construction $\mathcal{B}_{\Gamma',\mathcal{S}',P'}$ derived from the type system, but with the re-proved Lemma 4.26. □

4.5 Publications and acknowledgements

Work building on the contributions of this chapter was first presented by me at a workshop:

- [83] Robert Sison. Per-thread compositional compilation for confidentiality-preserving concurrent programs. In *2nd Workshop on Principles of Secure Compilation*, Los Angeles, January 2018. Cătălin Hriţcu.

Subsequently, further work building directly on them has appeared in the publications:

- [68] Toby Murray, Robert Sison, and Kai Engelhardt. COVERN: A logic for compositional verification of information flow control. In *European Symposium on Security and Privacy*, pages 16–30, London, United Kingdom, April 2018. IEEE.
- [85] Robert Sison and Toby Murray. Verifying that a compiler preserves concurrent value-dependent information-flow security. In *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141, pages 27:1–27:19, Portland, USA, September 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

Some definitions in Section 4.4 survive unchanged, apart from rephrasing for presentation, from an unfinished early prototype by Edward Pierzchalski: $\mathcal{S} \oplus k$ and $\mathcal{S} \ominus k$ (Definition 4.16), and unlock-maintains-tracking-stability (Definition 4.19). These adapt analogous concepts from the ANNO typing rule of our previous work, Murray et al. [67, 65]. Pierzchalski's initial versions of the new security typing rules LOCK and UNLOCK, however, underwent changes by me in the course of my soundness proof effort (Section 4.4.2). Furthermore, the restriction in Section 4.1.2 ensuring variables and their control variables are always locked simultaneously, and the final versions of Isabelle proofs for two subsidiary lemmas in Section 4.4.2 (Lemmas 4.20 and 4.21), are thanks to Toby Murray.

4.6 Summary

This chapter has presented a novel set of per-thread proof techniques that is sufficient to verify noninterference for mixed-sensitivity concurrent programs. In doing so, it demonstrated how to use assume–guarantee to prove that mutex lock-based synchronisation

primitives, supplied with a way of explicitly specifying their footprints, yield a fully compositional proof method—that is, that those primitives maintain as invariant that assumptions on threads' access to shared state are always met by corresponding guarantees.

To this end, this chapter proved that, with lock footprints specified by the programmer, the semantics of the `While` language [67, 65] with mutex locks added (in Section 4.1) itself ensures ongoing compatibility between the guarantees and assumptions (in Section 4.3). Having discharged this one non-compositional (global) compositionality requirement from the background work, this chapter then soundly extends for mutex lock support (1) a check of local compliance with guarantees promised by locking discipline (in Section 4.2), and (2) a security type system for proving noninterference for each thread (in Section 4.4).

Thus, a developer wishing to prove confidentiality for a mixed-sensitivity concurrent program written in `While` with mutex locks is obliged only to provide details of the program's locking discipline, and to exercise the checks on each of the threads considered independently. Chapter 6 will demonstrate the success of this approach on a case study.

Chapter 5

Noninterference-preserving compiler for mixed-sensitivity concurrent `While` programs

As a second contribution, this chapter presents the Covern wr-compiler: the first compiler proved to preserve proofs of noninterference for mixed-sensitivity concurrent programs. Implemented by me as an Isabelle function [84] adapting a compilation scheme originally for fault-resilient noninterference [95], the wr-compiler compiles programs from the `While` language of Chapter 4, to a generic RISC-style assembly language with mutex locks. Here, by using assume–guarantee modes [54, 67] and a decomposition principle [66, 85] to prove it introduces (resp.) no race conditions or timing leaks, I prove that it satisfies the needed notion of refinement to preserve noninterference despite mixed-sensitivity concurrency.

This contribution demonstrates the *applicability to compiler verification* of the CVDNI-preserving refinement notion posed by Murray et al. [67], *as made feasible by* a decomposition principle for proving it [66, 85]; I presented both notions in Section 3.2 as preliminaries. Here the decomposition principle (Figure 3.3) is crucial because, in separating the concern of preventing new timing leaks, it avoids directly having to prove the cube-shaped refinement diagram (Figure 3.2) arising from its need to preserve a 2-safety property.

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 in terms of memory contents,[1] I 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

[1] For instance, CompCert [48] preserves *operations* on such memory locations, but its proofs of refinement lack the direct reasoning about their contents needed to capture interference by *other threads* changing the values of control variables, and thereby changing what the security property considers observable.

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

In preserving CVDNI, the wr-compiler preserves security proofs that are produced by my program verification techniques of Chapter 4. Such an application of the wr-compiler to a case study program verified using these techniques will be demonstrated in Chapter 6.

This chapter will now proceed as follows. Section 5.1 will briefly introduce `RISC` with mutex locks, the target language of the Covern wr-compiler. Then Section 5.2 will focus on the wr-compiler's particular adaptations to concurrent value-dependent noninterference (beyond the fault-resilient noninterference targeted by the original compilation scheme of Tedesco et al. [95]), in the form of static checks and invariants that ensure and maintain the absence of race conditions on lock-protected shared variables. Section 5.3 formalises a ban, preserved by the wr-compiler, on secret-dependent control flow. Section 5.4 then presents formal proof, using the decomposition principle published alongside this chapter's contributions in Sison and Murray [85], that the wr-compiler implements CVDNI-preserving refinement. Section 5.5 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.

The chapter concludes with acknowledgement of publications on this work and how they relate to this chapter's presentation in Section 5.6, and a summary in Section 5.7.

5.1 Target language: `RISC` with mutex locks

The wr-compiler targets `RISC` with mutex locks (hereafter `RISC`), a generic RISC-style assembly language (based on the `RISC` architecture targeted by Tedesco et al. [95]) but for the addition of lock-based synchronisation operations **LockAcq** and **LockRel**:

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

The only new types here are register identifiers $r :: \mathit{Reg}$, and labels $l :: \mathit{Lab}$. A `RISC` program text is then a list of `RISC` instructions I, each optionally associated with a label. In contrast, the types of the constant values $n :: \mathit{Val}$, binary arithmetic operators $\oplus :: \mathit{Val} \Rightarrow \mathit{Val} \Rightarrow \mathit{Val}$, shared program variables $v :: \mathit{Var}$, and shared lock variables $k :: \mathit{Lock}$ are fixed to be the same as those for the source `While` language being compiled.

The new **LockAcq** k and **LockRel** k operations are then given the same operational semantics on the shared memory and mode state as the **lock**(k) and **unlock**(k) primitives from `While` (Section 4.1). Thus, the wr-compiler is expected to have knowledge of the locking discipline supplied by the program developer for the `While` program being compiled,

so as to be able to ensure that the `RISC` program it produces follows the same discipline.

As for `While` in Section 4.1, I instantiate the CVDNI theory [67] (recalled in Section 3.1) for `RISC` assuming that the underlying concurrency model (e.g. OS, scheduler etc.) prevents one thread from reading the program text of another. Here for `RISC`, I furthermore assume that the context switching mechanism ensures effectively that no thread can read or interfere with the contents of the registers when active for another thread. This includes a distinguished *program counter* register, which captures the current thread's program location as an index into its `RISC` program text. Based on these assumptions, I 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: $\langle((pc, P), regs), mds, mem\rangle_{\mathsf{r}}$. The subscript $_{\mathsf{r}}$ distinguishes the configuration triple as being for a `RISC` program (as opposed to $_{\mathsf{w}}$ for `While` configurations).

Apart from this notational adaptation to the configuration triple format for CVDNI proofs, the `RISC` language's evaluation semantics follows that of the `RISC` target architecture of Tedesco et al. [95]. For the new **LockAcq** k and **LockRel** k operations, the program counter is incremented by the `RISC` equivalents for the LockAcq and LockRel evaluation rules for the `While` language, and left unchanged by those for LockSpin and LockInvalid (Section 4.1.1). There is no evaluation rule that changes the program text.

The direct-addressing **Load** and **Store** instructions, and conditional **Jz** (jump if zero), are adequate for `RISC` to implement all features of `While` present in Chapter 4.

5.2 Preserving race-free expression evaluation

Recall from Section 3.2 that CVDNI-preserving refinement [67] 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 (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.

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* $\Phi :: RegRec \triangleq Reg \rightharpoonup exp$. This draws inspiration from that used by the compilation scheme of Tedesco et al. [95] (originally of type $Reg \rightharpoonup Var$) to avoid generating unnecessary **Load** instructions to registers that already contain a variable; in addition, here I extend it to track entire expressions on shared variables.
- An *assumption record* $\mathcal{S} :: AsmRec \triangleq (Var\ set \times Var\ set)$ that, like the security type system of Chapter 4, tracks which variables at a given point in the source `While` program are "stable" due to having an **AsmNoW** or **AsmNoRW** assumption.

The wr-compiler's main function compile-cmd then outputs every register–assumption record pair (or *compilation record*) $C = (\Phi, \mathcal{S}) :: CompRec \triangleq RegRec \times AsmRec$ associated with the program state *before* execution of each instruction in the output `RISC` program.[2] A typical invocation to compile some $c :: cmd$ takes an *initial compilation record* C, and returns the *CompRec*-annotated `RISC` program $PCs :: (I \times CompRec)\ list$ (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)**.**

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

The remainder of this section will focus on formal properties of the compilation records output alongside each `RISC` text: Section 5.2.1 will elaborate on checks enforced on input programs with the help of *AsmRec*s, and Section 5.2.2 will present a resulting property that *RegRec*s track stable expressions, needed to prove security preservation (in Section 5.4).

Remaining details (e.g. l, l', nl, nl' for label allocation) will be relegated to Appendix A. I 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 5.2.1), or if expression depth exceeds the limit assumed by the register allocation scheme model (elided to Appendix A.2); also, (2) relative to the label allocation scheme (elided to Appendix A.1) I 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.

5.2.1 Requirements on inputs to the wr-compiler

I define a shared variable v to be recorded as assumed *stable* if it and all its control variables (i.e. $\mathcal{C}\mathsf{vars}\ v$) 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 $\mathcal{S}$)**.**

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

For register record entries to be of any help in ensuring consistency of `While` and `RISC` expression evaluation, I exclude expression evaluation on race-prone variables by lifting the

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

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)**.**

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

I then implement a collection of stability-checks :: $cmd \times CompRec \Rightarrow bool$ (called no-unstable-exprs in Sison and Murray [85]) 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 requirement is the main one I described at the beginning of Section 5.2: The program must not refer to expressions on any unstable variables.

  Note that this means that even a simple assignment $x := y$, of a single variable to another, must be lock-protected; this is stricter than what is enforced by the local compliance check of Section 4.2, because it is a means of requiring the developer of the `While`-language program to be explicit in ensuring that an assignment from an otherwise-unstable variable will still execute atomically in the `RISC` output program. In this thesis I will call locks introduced solely for this purpose "read-atomicity" locks, a practice I will exercise in Chapter 6.

- The remaining requirements are ones that the wr-compiler expects to be enforced by the security type system and local compliance check of Chapter 4:

  - If the program assigns *to* an unstable variable, that variable 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 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.
  - Any **while**-loops in the program must restore the original set of locks held on loop entry (again, as captured by the assumption record) on loop termination.

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. (If any of these requirements are violated, compile-cmd rejects the program with $failed =$ True.) 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):

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

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

5.2.2 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)**.**

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

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, I 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, I 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. □

5.3 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 applied by the security type system of Chapter 4 (inherited from Murray et al. [67]), and—as noted in Section 2.2.5—commonly applied as a means to prevent all implicit flows, including timing leaks; note that I will revisit this in Chapter 7. 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 $\mathcal{I}_{\textsf{wr}}$ (see Section 5.4.3).

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 3.1.2) 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. Stated formally:[3]

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

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

Then, in Section 5.5, I 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, the *pc-security* notion of Molnar et al. [61] (noted in Section 2.2.5), but also explicitly equating the program text:

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

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

5.4 Use of the decomposition principle

Having covered the most relevant aspects of the wr-compiler's implementation, I now present the refinement relation $\mathcal{R}_{\mathsf{wr}}$ (in Section 5.4.1), pacing function $\mathsf{abs\text{-}steps}_{\mathsf{wr}}$ (in Section 5.4.2), and concrete coupling invariant $\mathcal{I}_{\mathsf{wr}}$ (in Section 5.4.3), parameters I use to apply the decomposition principle I presented in Chapter 3 to prove (in Section 5.4.4) that successful compilations are legitimised by secure-refinement (Definition 3.15)—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 3.19) for a targeted class of input `While`-language programs—ones with no secret-dependent control flow—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 I avoid a direct proof of the cube-shaped refinement diagram (Figure 3.2) of Murray et al. [67]—which would have involved reasoning about both $\mathcal{R}_{\mathsf{wr}}$ and $\mathcal{I}_{\mathsf{wr}}$ at once—and instead prove (with the assistance of $\mathsf{abs\text{-}steps}_{\mathsf{wr}}$) a square-shaped refinement diagram for $\mathcal{R}_{\mathsf{wr}}$ (Figure 3.3a) more typically found in compiler verification.

[3] Here, the helper function leftmost-cmd gives the leftmost in a sequence of ;-separated `While`-language commands.

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

In this section I introduce the refinement relation $\mathcal{R}_{\mathsf{wr}}$ that characterises compilation of programs from `While` to `RISC` using the wr-compiler, and prove it satisfies the two properties demanded of $\mathcal{R}_{\mathsf{wr}}$ (alone) by formal secure-refinement (Definition 3.15):

1. Preservation of modes and all contents of shared memory (preserves-modes-mem, Definition 3.13), and

2. Closedness under changes by other threads (closed-others, Definition 3.14).

An actual proof of refinement (using the square-shaped diagram of Figure 3.3a) for $\mathcal{R}_{\mathsf{wr}}$ will be deferred to Section 5.4.2, which introduces the $\mathsf{abs\text{-}steps}_{\mathsf{wr}}$ function pacing it.

Just like the earlier example of a secure refinement relation (in Figure 3.1), the refinement relation $\mathcal{R}_{\mathsf{wr}}$ pairs abstract (here, `While`-language) with concrete (here, `RISC`-language) program configurations. For example, the `if_expr` case of $\mathcal{R}_{\mathsf{wr}}$ 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 A.1, and a relevant excerpt of the compile-cmd implementation provided in Figure A.2 for comparison, both on page 131 of Appendix A. An informal description of all the cases of $\mathcal{R}_{\mathsf{wr}}$, their purpose, and the invariants they maintain, is also relegated to Appendix A.)

I define almost all the cases of $\mathcal{R}_{\mathsf{wr}}$ to assert at least one successful run of compile-cmd (where *failed* = False). I then define a guard that I impose to restrict the scope of $\mathcal{R}_{\mathsf{wr}}$ 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 $\mathcal{S}$ *mds* $\triangleq$ $\mathcal{S} = (mds\ \mathbf{AsmNoW},\ mds\ \mathbf{AsmNoRW})$

    (Consistency between an assumption record and a mode state)

Apart from using compiled-cmd-config-consistent to restrict the scope of $\mathcal{R}_{\mathsf{wr}}$ in this manner, I will also impose it in Section 5.4.4 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 $\mathcal{R}_{\mathsf{wr}}$ also tend to assert regrec-stable (Definition 5.2), which I already proved holds for all compilation records produced by the wr-compiler (Lemma 5.4).

Finally, whenever a case of $\mathcal{R}_{\mathsf{wr}}$ 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 $\mathcal{R}_{\mathsf{wr}}$ thus specified, I can now prove the two requirements for secure-refinement that pertain to $\mathcal{R}_{\mathsf{wr}}$ alone: preserves-modes-mem (Definition 3.13), and closed-others (Definition 3.14). In short, preserves-modes-mem is largely enforced by the definition of $\mathcal{R}_{\mathsf{wr}}$, but closed-others relies in part on $\mathcal{R}_{\mathsf{wr}}$ only ever talking about stable register records:

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

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

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

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

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

Changes by others (Definition 3.14) 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 $\mathcal{R}_{\mathsf{wr}}$ guards mention shared memory. □

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

In this section I nominate a pacing function, $\mathsf{abs\text{-}steps}_{\mathsf{wr}}$, 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 3.3a (part of Definition 3.18), I then prove that the $\mathcal{R}_{\mathsf{wr}}$ relation I introduced in Section 5.4.1 is a refinement when "paced" by $\mathsf{abs\text{-}steps}_{\mathsf{wr}}$ in this manner.

Here I 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 3.3b (part of Definition 3.19)—which we will be obliged to prove in

Section 5.4.4. To this end, abs-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, abs-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, abs-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, 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, by secure-refinement-decomp (Definition 3.18), to prove refinement preservation (Figure 3.3a). To this end, I 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, I in fact prove it separately for regrec-mem-consistent and asmrec-mds-consistent, both times by induction on the structure of the While program c.

In each case, I 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 I 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 **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 ($\mathcal{R}_{\mathsf{wr}}$ is a refinement paced by $\mathsf{abs\text{-}steps}_{\mathsf{wr}}$)**.**

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

Proof. By induction on the structure of $\mathcal{R}_{\mathsf{wr}}$. (Refer to Appendix A, Section A.3, for an informal description of all cases of $\mathcal{R}_{\mathsf{wr}}$.)

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, I use Lemma 5.10 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 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 I use Lemma 5.11 to obtain that the result is an accurate calculation of the expression, and this time I 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 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 $\mathcal{R}_{\mathsf{wr}}$ 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. □

5.4.3 Concrete coupling invariant $\mathcal{I}_{\mathsf{wr}}$

The next element needed is the concrete coupling invariant $\mathcal{I}_{\mathsf{wr}}$. Recall from Section 5.3 that the no-high-branching requirement (Definition 5.5) ensures that input `While` programs have no secret-dependent control flow; here I choose $\mathcal{I}_{\mathsf{wr}}$ to ensure that the wr-compiler has not introduced any *new* secret-dependent control flow in the output `RISC` program.

I define $\mathcal{I}_{\mathsf{wr}}$ 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 $\mathcal{I}_{\mathsf{wr}}$ for compiled programs)**.**

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

From this definition, pc-security (Definition 5.6) is clearly immediate for any concrete bisimulation $\mathcal{B}_{\mathsf{C}}$ of $\mathcal{B}\ \mathcal{R}\ \mathcal{I}_{\mathsf{wr}}$ (Definition 3.16) derived using $\mathcal{I}_{\mathsf{wr}}$.

5.4.4 Proof of CVDNI-preserving refinement

With $\mathcal{R}_{\mathsf{wr}}$, $\mathsf{abs\text{-}steps}_{\mathsf{wr}}$, and $\mathcal{I}_{\mathsf{wr}}$ nominated, we are ready to prove confidentiality-preserving refinement using the decomposition principle secure-refinement-decomp (Definition 3.18).

To this end, I now prove the suitability of these three parameters, for `While` programs that do not branch on High-sensitivity values (as I specified earlier, in Section 5.3):

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

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

Proof. Unfolding Definition 3.19 gives us the following obligations. (See also Figure 3.3.)

For consistent stopping behaviour, I 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 (respectively) 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, I use $\mathsf{no\text{-}high\text{-}branching}$ for $\mathcal{B}$ with memory preservation via $\mathcal{R}_{\mathsf{wr}}$ (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 (Section 4.1.2), I 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 5.15 ($\mathcal{R}_{\mathsf{wr}}, \mathsf{abs\text{-}steps}_{\mathsf{wr}}, \mathcal{I}_{\mathsf{wr}}$ meet decomposed $\mathsf{secure\text{-}refinement}$ requirements)**.**

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

Proof. Unfolding Definition 3.18, the obligations pertaining only to $\mathcal{R}_{\mathsf{wr}}$ and $\mathsf{abs\text{-}steps}_{\mathsf{wr}}$ are discharged by Lemma 5.12, Lemma 5.9, and Lemma 5.8. Pertaining to $\mathcal{I}_{\mathsf{wr}}$: Clearly $\mathcal{I}_{\mathsf{wr}}$ is symmetric, and furthermore it is $\mathsf{cg\text{-}consistent}$ (Definition 3.6) 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 (regarding $\mathsf{decomp\text{-}refinement\text{-}safe}$) is discharged by Lemma 5.14. □

From this it follows immediately via Theorem 3.20 that $\mathcal{R}_{\mathsf{wr}}$ with the help of $\mathcal{I}_{\mathsf{wr}}$ describes a confidentiality-preserving refinement for non-High-branching `While` programs:

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

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

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

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

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

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

The compiler input and initial configuration conditions I impose allow me 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 $\mathcal{R}_{\mathsf{wr}}$ (described further in Appendix A, Section A.3) that I designed for them, respectively. □

5.5 Proof of compositional noninterference preservation

Going beyond the level of detail of our presentation in Sison and Murray [85], I now present the final few steps to obtain preservation of (1) compositional security properties down to `RISC` level on a per-thread basis, and (2) whole-system security for concurrent compositions of those `RISC` threads when obtained via compilation by the wr-compiler.

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

Lemma 5.18 (Programs witnessed by $\mathcal{B}$ are captured by $\mathcal{B}_{\mathsf{C}}\mathsf{of}\ \mathcal{B}\ \mathcal{R}_{\mathsf{wr}}\ \mathcal{I}_{\mathsf{wr}}$ once compiled)**.**

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

Proof. Straightforward from the definition of $\mathcal{B}_{\mathsf{C}}\mathsf{of}$ (Definition 3.16), using Theorem 5.17 to show membership of $\mathcal{R}_{\mathsf{wr}}$, and the definition of strong-low-bisim-mm (Definition 3.4) to show that the memories are low-equal modulo modes, as required by $\mathcal{B}_{\mathsf{C}}\mathsf{of}$. Finally, membership of $\mathcal{I}_{\mathsf{wr}}$ (Definition 5.13) follows from the fact that the paired configurations are at the same location (program counter 0) of the same program P. □

I initialise the compiler with an empty C_0 :: *CompRec* that knows nothing about the register contents, and assumes no variables to be stable:

Definition 5.19 (Empty compilation record C_0)**.**

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

With these definitions we have the desired consistency result:

Lemma 5.20 (Initial $\mathsf{C}_0, \mathsf{mds}_0$ are consistent with no-locks-held)**.**

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

Proof. This is straightforward by unfolding Definitions 5.7, 4.13, 4.14, and 5.19, also relying on the cleanliness conditions on locking disciplines specified in Section 4.1.2. □

We are ready to state the security preservation result formally. As I explained in Section 5.3, 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 4.3.2), I will impose no-locks-held (Definition 4.13) as an initial condition. Therefore, the security preservation theorem I choose to prove here demands the input `While` program be $\mathsf{com\text{-}secure}^{\mathsf{no\text{-}high\text{-}branching}}_{\mathsf{no\text{-}locks\text{-}held}}$ (Definition 3.7, with additional requirements as specified). Given this, it then promises that the output `RISC` program is $\mathsf{com\text{-}secure}^{\mathsf{pc\text{-}security}}_{\mathsf{no\text{-}locks\text{-}held}}$:

Theorem 5.21 (Preservation of per-thread confidentiality by the wr-compiler)**.**

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

Proof. We are given by $\mathsf{com\text{-}secure}^{\mathsf{no\text{-}high\text{-}branching}}_{\mathsf{no\text{-}locks\text{-}held}}$ (Definition 3.7) that for low-equal starting configurations (modulo modes) of c with no locks held, there exists some witness $\mathcal{B}$ satisfying both strong-low-bisim-mm and no-high-branching.

From this and Lemma 5.18 we have that the output program's corresponding execution is captured by a `RISC` semantics-level relation $\mathcal{B}_{\mathsf{C}}\mathsf{of}\ \mathcal{B}\ \mathcal{R}_{\mathsf{wr}}\ \mathcal{I}_{\mathsf{wr}}$ derived from this $\mathcal{B}$, with Lemma 5.20 discharging the compiled-cmd-config-consistent requirements.

Corollary 5.16 then gives us that $\mathsf{secure\text{-}refinement}\ \mathcal{B}\ \mathcal{R}_{\mathsf{wr}}\ \mathcal{I}_{\mathsf{wr}}$ holds, and from this and $\mathsf{strong\text{-}low\text{-}bisim\text{-}mm}\ \mathcal{B}$ using Theorem 3.17 we have $\mathsf{strong\text{-}low\text{-}bisim\text{-}mm}\ (\mathcal{B}_{\mathsf{C}}\mathsf{of}\ \mathcal{B}\ \mathcal{R}_{\mathsf{wr}}\ \mathcal{I}_{\mathsf{wr}})$. 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 3.7; as Section 5.4.3 noted, pc-security (Definition 5.6) is immediate from the definition of $\mathcal{I}_{\mathsf{wr}}$. □

To prove a whole-system security result at the `RISC` level for the compiled program, we must also prove sound-mode-use (Definition 3.10). To that end, I prove a local and global result for `RISC` programs output by the wr-compiler when given a secure `While` program.

The local compliance result follows from a property of the refinement relation, $\mathcal{R}_{\mathsf{wr}}$:

Lemma 5.22 (Each step from a `RISC` configuration in $\mathcal{R}_{\mathsf{wr}}$ respects its own guarantees)**.**

$$\frac{(\langle c, mds, mem\rangle_{\mathsf{w}}, \langle((pc, P), regs), mds, mem\rangle_{\mathsf{r}}) \in \mathcal{R}_{\mathsf{wr}}}{\begin{array}{c}\forall x.\ (x \in mds\ \mathbf{GuarNoRW} \longrightarrow \mathsf{doesnt\text{-}read\text{-}or\text{-}modify}\ ((pc, P), regs)\ x) \wedge \\ (x \in mds\ \mathbf{GuarNoW} \longrightarrow \mathsf{doesnt\text{-}modify}\ ((pc, P), regs)\ x)\end{array}}$$

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

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 $\mathcal{R}_{\mathsf{wr}}$), we are obliged to show that the `RISC` instruction paired to it by $\mathcal{R}_{\mathsf{wr}}$ similarly respects the guarantee modes implied by the locking discipline (as specified in Section 4.1.1). I do so with a mixed Isar/"apply"-style proof that exercises the `RISC` semantics for all the relevant cases, making much use of lemmas about control flow under sequential composition (mentioned in Section 5.2—see also Appendix A). □

Lemma 5.23 (Refinements in $\mathcal{R}_{\mathsf{wr}}$ ensure local mode compliance)**.**

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

Proof. Unfolding Definition 3.11, we must show that what was proved by Lemma 5.22 holds for every `RISC` configuration reachable from $\langle((pc, P), regs), mds, mem\rangle_{\mathsf{r}}$.

First, I prove a lemma that establishes that every such reachable `RISC` configuration is also paired by $\mathcal{R}_{\mathsf{wr}}$ to some `While` configuration. Specifically, I prove that $\mathcal{R}_{\mathsf{wr}}$ 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 5.4.2).

2. Between each evaluation step, arbitrary changes are allowed to occur to the memory locations judged by the mode state to be writable (Definition 3.5).

Because all such `RISC` configurations reachable from the initial one are in $\mathcal{R}_{\mathsf{wr}}$, it then follows from Lemma 5.22 that they respect their own guarantees, as required. □

Lemma 5.24 (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. This follows from Theorem 5.17, Lemma 5.20, and Lemma 5.23. □

Then I prove invariance of global modes compatibility (as in Section 4.3) for compiled `RISC` programs, due to `RISC`'s identical semantics to `While` regarding locking and modes:

Lemma 5.25 (Initialising `RISC` with no-locks-held, 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. I firstly prove versions of Lemma 4.10, Lemma 4.11, and Theorem 4.12 for `RISC`, following exactly the same reasoning as I did in Section 4.3 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 4.15 did for `While`. □

We now have enough information to derive a whole-system security result, for concurrent `RISC` programs obtained by running the wr-compiler on secure `While` programs:

Theorem 5.26 (Secure threads compiled by the wr-compiler form a secure system)**.**

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

Proof. By Theorem 3.8 and unfolding Definition 3.10, 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 5.21, Lemma 5.24, and Lemma 5.25. □

5.6 Publications and acknowledgements

The work described in this chapter was first presented by me in the workshop talk already noted in Section 4.5:

- [83] Robert Sison. Per-thread compositional compilation for confidentiality-preserving concurrent programs. In *2nd Workshop on Principles of Secure Compilation*, Los Angeles, January 2018. Cătălin Hriţcu.

It subsequently appeared in the publication (also noted in Section 4.5):

- [85] Robert Sison and Toby Murray. Verifying that a compiler preserves concurrent value-dependent information-flow security. In *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141, pages 27:1–27:19, Portland, USA, September 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

This chapter recounts a selection of the content of this publication, brings details from its extended version (`arXiv:1907.00713 [cs.LO]`) in-line with what was published, and expands on it with further details to clarify its relationship with the work of the rest of the thesis. Notably, whereas Sison and Murray [85] stops at the compiler's fulfilment

of the refinement notion, Section 5.5 expands the presentation here to formal proofs of per-thread security property preservation (via the refinement) and whole-system security property preservation (via compositionality of the per-thread property) by the compiler.

5.7 Summary

This chapter has presented the second major contribution of this thesis: the Covern wr-compiler from `While` (of Chapter 4) to `RISC` (of Section 5.1), which is the first compiler proved to preserve proofs of noninterference for mixed-sensitivity concurrent programs.

In demonstrating such a proof (in Sections 5.4 and 5.5), this chapter attests to the applicability to compiler verification of the CVDNI-preserving refinement notion for security posed by Murray et al. [67], as made feasible by a decomposition principle for proving it (published alongside this chapter's contributions, in Sison and Murray [85]). It also shows that a valid approach to implementing such a compiler (presented in Section 5.2) is to use assume–guarantee mode state (1) to apply strict checks on the input source, and (2) to maintain invariants internally, both so as to maintain the absence of race conditions.

Thus, a developer who has used the methods of Chapter 4 to prove confidentiality, for a mixed-sensitivity concurrent `While` program with no secret-dependent control flow (as specified in Section 5.3), can rely on the compiler to preserve that confidentiality to the output `RISC` program. Chapter 6 (specifically Section 6.6) will demonstrate the success of this compiler verification-based approach to security preservation, on a case study.

Chapter 6

Case study: Cross Domain Desktop Compositor input handler

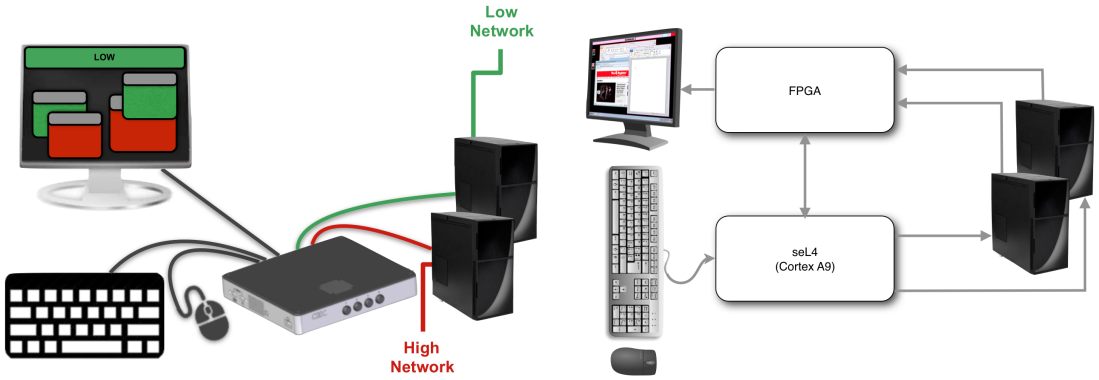
This chapter presents the third and central contribution of this thesis, and the main case study for the contributions of Chapters 4 and 5: the first mixed-sensitivity concurrent program proved to satisfy a noninterference property, preserved by a compiler down to an assembly-level model. This contribution validates the central claim of this thesis, that proving confidentiality and its preservation by a compiler is feasible for such programs.

The Cross Domain Desktop Compositor (CDDC) of Beaumont et al. [13] is a desktop device that gives trusted users the option of replacing multiple monitor, keyboard, and mouse setups with a single multi-level secure user interface (via a single monitor, keyboard, and mouse, as depicted in Figure 6.1a) when using several desktop computers simultaneously. Here I present as case study a software 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 thesis' main focus:

- Concurrency—here, between *software components* whose execution is interleaved (by the seL4 operating-system microkernel [44]), 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.

By exercising the *program verification* techniques of Chapter 4—with assume–guarantee compatibility proved as an invariant of the language—on a `While` model of this case study, I show that these techniques constitute a practical, per-component compositional approach to obtaining a security proof, that handles both of these characteristics successfully.

Then, by exercising the *compiler verification* result of Chapter 5—a compiler that preserves information-flow security—on this `While` model, I show this compiler verification-based approach to be feasible for obtaining the preservation of proved security results, straightforwardly and for little extra effort, down to a `RISC` model of the program.



(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. [13]) using an FPGA.

Figure 6.1: Functional schematics for Cross Domain Desktop Compositor hardware. Reproduced from Murray et al. [68].

The chapter will proceed as follows. Following an overview (in Section 6.1) of the main characteristics of the case study, we will examine certain aspects of the `While` model in detail: firstly (in Section 6.2) the device interfaces where the attacker and trusted user are modelled, then (in Section 6.3) the internal device and inter-component interactions via shared memory. Discussion of these aspects will focus on the impacts of the case study's characteristics on (1) the classification of the locations concerned, and (2) the use of mutex locks to govern access to those locations. Section 6.4 then presents the formal security properties I proved subsequently about this `While` model, and how I obtained them using the program verification techniques that I presented in Chapter 4; Section 6.5 touches on the extent to which applying this (mostly automated) process required user intervention. Finally, Section 6.6 presents the formal preservation of security properties down to a `RISC` model, obtained from running the verified wr-compiler of Chapter 5 on the `While` model. I conclude with a list of relevant publications (Section 6.7) and a summary (Section 6.8).

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 6.1b). 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 6.1a).

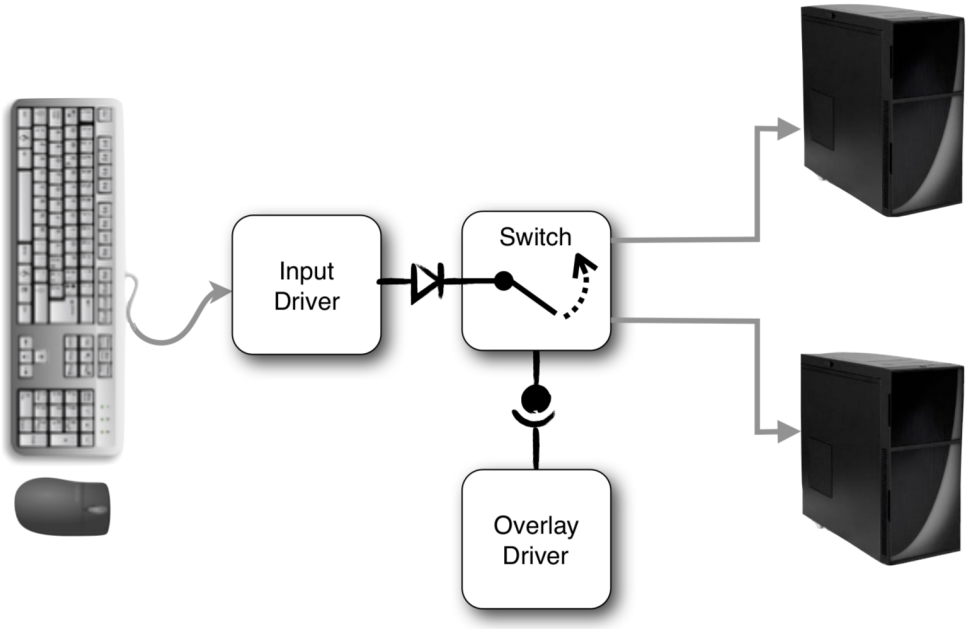


Figure 6.2: Functional schematic of seL4 component architecture for CDDC HID switch. Reproduced from Murray et al. [68].

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-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.

Shared-variable concurrency

The software implementation (replacing the original FPGA-based implementation [13]) 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 operating-system microkernel [44].

For this thesis, I 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 6.2) 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).
- The Switch is responsible for inspecting all input events on the buffer (from the Input driver), determining (partly by querying the Overlay driver) if any constitute

a user-directed change to the destination of subsequent events, and if so, updating the compositor device to display that change. Finally, it is responsible for delivering all events to their destination computer via the appropriate *output-device* interface.

- 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.

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 will be used to model all synchronisation and restriction of concurrent access by the components to those variables.

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 memory locations, rather than from those of distinct device sets for each sensitivity level.

Furthermore, the HID switch propagates all input data (regardless of sensitivity) through a single set of memory locations (the input-event buffer, and SWITCH-internal variables), 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 will be assigned value-dependent classifications, reflecting the trusted user's expectation of the sensitivity level of the data they contain.

6.2 External device interactions

As described in Section 6.1, the information-flow security goal for the CDDC's HID switch is—intuitively—to prevent input data going to the "wrong" output-device interface, from the perspective of the trusted user generating those inputs.

This goal impacts mostly on the classifications of locations modelling (1) the input-device interfaces, where I have explicitly modelled the trusted user's perspective and behaviour; and (2) the output-device interfaces, where I have modelled the attacker.

Meanwhile, all of the shared variables used to model device interfaces are left unprotected by mutex locks for most of the time. As the analysis assumes that environmental havoc occurs to all write-unprotected shared variables between evaluation steps, this effectively models (near-)arbitrary changes by an environment.[1] The trusted user and the attacker merely form part of that environment, with the trust in the user encoded in the classification of the variables concerned—as opposed to any restrictions on their contents.

[1] "Near"-arbitrary because, as explained in Chapter 3, the security property does assume that the havoc will not write High-sensitivity data to any control variables or non–read-protected Low-classified locations.

```
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 6.3: Examples of external device interactions by the CDDC HID switch model—here, for the keyboard events.

6.2.1 Attacker model (at the output devices)

I 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.[2] One computer is designated as belonging to the High security domain, and is the only legitimate destination for High-sensitivity input events. The other computer is designated as belonging to the Low security domain.

The output-device interfaces are modelled as shared variables, abstracting the hardware and connections that the SWITCH component uses to forward events to the destination computers (as depicted in Figure 6.3b). These are classified statically: one High, the other Low. The attacker is then considered to be an entity that can read at any time from the output-device interface variable that is classified Low.

Note also that the analysis is in fact robust to a more powerful attacker that may write to these output-device interfaces.[3] (A reason this does not make our analysis harder is that none of the CDDC components modelled ever reads from them.)

6.2.2 Trusted user model (at the input devices)

The model trusts the user to type sensitive information into the keyboard only when they see that the compositor device is indicating that it is safe to do so—that is, when it is indicating that it is the High domain whose computer is *active*, meaning it is currently the computer that is set to receive all keyboard events.

I model here explicitly that the user's perception and actions are entirely faithful to

[2] The rationale for such simplifications for this thesis is that, 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.

[3] They are still assumed not to write High-sensitivity data of their own to the Low-classified output sink.

what is indicated by the compositor, following a similar assumption by Beaumont et al. [13]. I do this by using a shared variable named `indicated_domain` to model the security domain that the compositor is indicating as active, and then having the INPUT driver draw keyboard events from a shared variable (as depicted in Figure 6.3a) that has value-dependent classification (where `DOM_HIGH` is a designated constant):

$$\begin{cases} \mathsf{High}, & \text{if } \texttt{indicated_domain} = \texttt{DOM_HIGH} \\ \mathsf{Low}, & \text{otherwise} \end{cases}$$

Furthermore, the model trusts the user not to encode sensitive information into the mouse input in any way. I will explain soon (in Section 6.3) that this is because mouse events (unlike keyboard events) in this program have a potential ability to influence the future value of control variables, which in turn are not ever permitted to receive any High-sensitivity data. Thus, the INPUT driver always draws mouse events from a statically Low-classified shared variable, regardless of the current value of `indicated_domain`.

6.3 Internal device and inter-component interactions

Recall from Section 6.2 that I modelled the attacker as an entity that can read from a particular statically Low-classified location—this was so that the security analysis would flag immediately any possibilities that High-sensitivity data would arrive in that location.

I will also statically classify as Low any internal locations where we do not ever expect any High-sensitivity information to arrive—the intent is that our security analysis will then flag immediately, to us as the program verifiers, any such violations of our expectations. Because mouse events are considered to be of Low sensitivity, this will include the compositor device interfaces (Section 6.3.1), and everything used for and by the OVERLAY driver (Section 6.3.2); these parts are only ever used to process mouse data.

Finally, locations subject to mixed-sensitivity reuse will be assigned value-dependent classifications. This primarily concerns the input-event buffer between the INPUT driver and SWITCH component (Section 6.3.3), and some SWITCH-internal local variables, all of which are used to hold input events regardless of their sensitivity level and eventual destination. This achieves that the security analysis will flag an arrival of High-sensitivity data in a location, *precisely when* their arrival would violate an expectation that only Low-sensitivity data should be there *at that time*.

6.3.1 Use of the compositor device (by OVERLAY driver, SWITCH)

The CDDC's HID switch—acting through its OVERLAY-driver and SWITCH components—makes use of a number of interfaces of the compositor device, all of which are modelled here as shared variables (approximating the usual representation of hardware interfaces as device registers mapped into a designated portion of the memory space).

```
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 6.4: Examples of the SWITCH component interfacing with the compositor device.

As mentioned in Section 6.2, I model compositor hardware state—specifically, the domain it is currently indicating as active—using a shared variable named `indicated_domain`. This `indicated_domain` variable also models directly the interface by which the SWITCH enacts a *domain switch*—that is, instructs the compositor hardware to change which domain it indicates as active. In the model, SWITCH does so by writing the desired domain identifier to the variable (as in Figure 6.4b). As a control variable whose value determines the classification of other variables, the `indicated_domain` is statically classified Low.

To leave the precise behaviour of the compositor device unspecified (as with the input and output devices) but *trusted not to inject secrets*, its interfaces are statically classified Low. As the strong low-bisimulation–based analysis is robust to arbitrary environmental behaviour that does not inject secrets into Low locations, I use this to model such behaviour at these interfaces whenever they are left unlocked. This is sufficient to model all interactions with the compositor device (an example of which is given in Figure 6.4a), as these operations are expected to deal only with Low-sensitivity data. Beyond the standard assumption not to inject secrets, any further requirements on the device's behaviour (and by extension, on the rest of the environment acting on it) need to be checked at runtime.

6.3.2 Remote procedure calls (between SWITCH and OVERLAY driver)

The SWITCH component queries the OVERLAY driver, which functions as a layer of abstraction to part of the compositor device interface concerned with visual overlay elements.

These queries only ever concern whether mouse coordinates lie within certain parts of the overlay. From the perspective of the security analysis, it matters only that the mouse event data (including the current position of the mouse cursor) is only ever expected to be of Low sensitivity. Thus, all shared variables implementing the model of this RPC mechanism are statically classified Low, so that any arrival of High data is caught by the analysis. This is needed because answers returned by the RPC query will influence the SWITCH's decision of whether to initiate a domain switch; recall from Section 6.3.1, as this amounts to a change in the value of the input-event buffer's control variable `indicated_domain` (as

```
lock(rpc_lock);
rpc_arg := current_event_data;
rpc_call := 1;
unlock(rpc_lock);
done_rpc := 0;
while (done_rpc != 0) do
  lock(rpc_lock);
  if (rpc_call != 0) then
    overlay_result := rpc_ret;
    done_rpc := 1
  else
    skip
  fi;
  unlock(rpc_lock)
od
```

(a) Switch making an RPC.

```
while (1) do
  lock(rpc_lock);
  if (rpc_call != 0) then
    /* ...
       Elided: Assign to rpc_ret
       DOM_(LOW|HIGH|OVERLAY|INVALID)
       depending on rpc_arg's value.
       ... */ ;
    rpc_call := 0
  else
    skip
  fi;
  unlock(rpc_lock)
od
```

(b) Overlay driver RPC server loop.

Figure 6.5: Remote procedure call abstraction between Switch, Overlay components.

depicted in Figure 6.4b), it cannot ever be allowed to be secret-dependent.

For the RPC mechanism itself, I model a rough abstraction in `While` using shared variables and a mutex named `rpc_lock`, but its precise details (provided in Figure 6.5) are largely not security critical. What matters more is that the security analysis, with all of these locations classified Low, ensures that High-classified data never arrives in these locations, so that they may never influence the value of any control variable down the line.

6.3.3 Input-event buffer (between Input driver and Switch)

The buffer that propagates events between Input driver and Switch is the only location in the HID switch that is subject both to mixed-sensitivity reuse, and to concurrent access by multiple threads. Consequently, value-dependent classifications and mutex locks are used simultaneously in this part of the model. I address each of these aspects in turn.

Value-dependent classification

I 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. This single-place buffer is split into a data portion and a control portion, each modelled by a shared variable, and classified as follows:

- The control portion—a variable named `input_event_type`—identifies whether the data portion describes a keyboard event or a mouse event. As one of the control variables for the data portion (see next point), it must be statically classified Low.
- The data portion—a variable named `input_event_data`, which may contain such data as mouse coordinates, or keystroke identifiers—is value-dependently classified. This classification depends both on (1) whether it pertains to a keyboard event, and also (2) if so, whether the keyboard event was typed in by the user when the

`indicated_domain` was High. Stated precisely, its classification is:

$$\begin{cases} \mathsf{High}, & \text{if } \texttt{input_event_type} = \texttt{KEYBOARD} \ \wedge \ \texttt{indicated_domain} = \texttt{DOM_HIGH} \\ \mathsf{Low}, & \text{otherwise} \end{cases}$$

where `KEYBOARD` and `DOM_HIGH` are the self-explanatory designated constants.

Note that the input-event buffer's classification as stated above will always assert mouse (as non–`KEYBOARD`-type) event data to be of Low sensitivity. This is important because the SWITCH component enacts domain switch in response to mouse events—for example, clicking on compositor-rendered buttons in the overlay, or clicking on a window belonging to a domain that is not active. As domain switch entails a change to the control variable `indicated_domain` (as was depicted in Figure 6.4b), the security type system will reject any program that allows such changes to be secret-dependent.

Mutex lock-enforced critical sections

Working with value-dependently classified locations subject to concurrency implies a need to establish a critical section that stabilises not only their contents, but also their classification (by stabilising the contents of their control variables). In the `While` model, I establish such critical sections using mutex locks.

For the input-event buffer in particular, I use a lock named `input_event_lock`; furthermore, I model two behaviours of the seL4-based SWITCH component's implementation regarding the buffer, that together allow minimising the size of the critical sections during which it must be locked in the model:

- I model the seL4-based SWITCH component's copying of the event from the shared input-event buffer into its own local variables. In the `While` model, SWITCH-local variables for the data and control portion are named `current_event_data` and `current_event_type` respectively; the copying is depicted in Figure 6.6b.

- Likewise, I model the seL4-based SWITCH component's maintenance of its own authoritative view of the currently active domain—in a variable named `active_domain`, to which it refers instead of ever querying the compositor device's `indicated_domain`.

Consequently, only small critical sections (locking the input-event buffer) are ever needed, as opposed to a very large one that would cover the entire duration of the SWITCH component's duties for each incoming event: determining its classification, checking (and enacting) if it initiates domain switch, and forwarding it to its destination computer.

The SWITCH component's internal `active_domain` must remain authoritative with respect to what is composited by the CDDC into the display. Thus in the `While` model, the SWITCH initialises `indicated_domain` to match the initial value of `active_domain` (as depicted in Figure 6.6a), and checks at runtime that `active_domain` = `indicated_domain` when copying data from the buffer to its own private variables (as depicted in Figure 6.6b).

```
/* 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 6.6: Examples of the SWITCH component interacting with the input-event buffer.

Finally, any local variables intended for private use by each component are governed by a dedicated lock acquired at initialisation time (`switch_private_lock` in Figure 6.6a for SWITCH), and held permanently. This models the memory protection mechanisms of seL4 that enforce ownership of non-shared memory configured to be private to each component.

6.4 Proof of confidentiality for the `While` model

Having now presented the `While`-language model for the CDDC's HID switch, this section will give a formal exposition of the verification of its security property. In short, for the concurrent program of all three software components (INPUT, SWITCH, and OVERLAY), I prove the whole-system security property (sys-secure, Definition 3.9) from Murray et al. [65] as presented in Chapter 3, as instantiated to specify that no locks are held initially.

My approach will be to use the security type system of Section 4.4 to establish the per-thread security property for each component, and then use the compositionality theorem (Theorem 3.8) to derive the whole system security property from the per-thread ones.

So that we can use the approach I gave in Section 4.3.2 to obtain the global modes compatibility part of the sound-mode-use side-condition (Definition 3.10), I will specify no-locks-held (Definition 4.13) as the *INIT* requirement on memory, and use the initial mode state mds_0 (Definition 4.14) for all of the components in the system. Instantiated in this way, the security compositionality theorem (Theorem 3.8) is then restated as:

Theorem 6.1 (Compositionality of $\mathsf{com\text{-}secure}_{\mathsf{no\text{-}locks\text{-}held}}$)**.**

$$\frac{\begin{array}{c}\forall (tps, mds) \in \mathsf{set}\ cms.\ \mathsf{com\text{-}secure}_{\mathsf{no\text{-}locks\text{-}held}}\ (tps, mds) \\ \forall mem.\ \mathsf{no\text{-}locks\text{-}held}\ mem \longrightarrow \mathsf{sound\text{-}mode\text{-}use}\ (cms, mem)\end{array}}{\mathsf{sys\text{-}secure}_{\mathsf{no\text{-}locks\text{-}held}}\ cms}$$

This no-locks-held predicate and mds_0 are both defined relative to a lock interpretation parameter that I supply (as required by Section 4.1.1) 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. Recall from Section 6.3 these were:
  - `input_event_lock`, for the input-event buffer between INPUT and SWITCH, the value-dependently classified keyboard source (modelling user behaviour), and their common control variable `indicated_domain`.
  - `rpc_lock`, for the OVERLAY component's RPC mechanism used by SWITCH.
- There are also locks granting the SWITCH and INPUT components exclusive read–write access to a set of "private" variables each, for internal use. As depicted in Figure 6.6a for SWITCH, 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 5.2.1, and depicted in Figures 6.3a and 6.4a. 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 for the `While` model's confidentiality result proved in this section, but rather for the preservation of that result (via small-step semantic preservation) down to the `RISC` model by the wr-compiler (this will be presented later, in Section 6.6).

As we need to impose no-locks-held as an *INIT* parameter in order to obtain global modes compatibility, the per-thread security property we need is $\mathsf{com\text{-}secure}_{\mathsf{no\text{-}locks\text{-}held}}$ (Definition 3.7, with $INIT \triangleq$ no-locks-held and no extra requirements on the bisimulation).

First I point out that if a program is secure without imposing any initial conditions, then it remains secure if we impose any, arbitrarily (i.e. any *INIT* parameter)—note this holds *regardless* of any *EXTRA* requirements imposed on the bisimulation witness:

Lemma 6.2 (Add *INIT* requirements to com-secure for free)**.**

$$\mathsf{com\text{-}secure}^{EXTRA}\ (c, mds) \implies \mathsf{com\text{-}secure}^{EXTRA}_{INIT}$$

Proof. This result is trivial from the definition of com-secure (Definition 3.7). □

Following from this, it suffices to use the security type system of Section 4.4 to show that each of the programs are $\mathsf{com\text{-}secure}_{\mathsf{no\text{-}locks\text{-}held}}$ when started with mode state mds_0. (From here I will use the identifiers OVERLAY, INPUT, and SWITCH to denote the `While` commands for each of the components.)

Lemmas 6.3 (Per-thread confidentiality results for CDDC `While` model)**.**

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

Proof. For each of the threads, applying the security type system is mostly automated using Eisbach proof methods [57], with some user intervention (see Section 6.5).

With a thread program successfully type-checked to completion, Theorem 4.27 gives us that the thread is com-secure. (The theorem's side-condition yields-stable-types *mds* is true trivially for mds_0, because by Definition 4.14 mds_0's assumption sets are empty.)

Lemma 6.2 then gives us that each thread is $\mathsf{com\text{-}secure}_{\mathsf{no\text{-}locks\text{-}held}}$, as required. □

We are now in a position to prove the whole-system confidentiality theorem:

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

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

Proof. By Theorem 6.1 (i.e. Theorem 3.8 with $\mathit{INIT} \triangleq \mathsf{no\text{-}locks\text{-}held}$, $\mathit{EXTRA} \triangleq (\lambda_\,.\,\mathsf{True})$).

Lemmas 6.3 satisfy our first obligation—that is, to show that $\mathsf{com\text{-}secure}_{\mathsf{no\text{-}locks\text{-}held}}$ (Definition 3.7) holds for all three components with the initial mds_0.

We are then obliged to show that sound-mode-use (Definition 3.10) holds for all initial states of the system that obey no-locks-held; this entails proving local-mode-compliance (Definition 3.11) and global-modes-compatibility (Definition 3.12).

The former I discharge for each component using the compliance check of Section 4.2, which is sound for local-mode-compliance via Theorem 4.4. Existing automation from Murray et al. [65], adapted by me for the mutex lock additions of Chapter 4, allows me to discharge this check for all three components without intervening.

Finally, I use Lemma 4.15 to obtain that imposing no-locks-held as the *INIT* condition is enough to ensure global-modes-compatibility. □

6.5 Automation and user intervention

I now touch on the extent to which verifying the `While` model (in Section 6.4) using the techniques of Chapter 4 was automated, versus requiring manual intervention. In short, use of the security type system is semi-automated, making use of existing automation support implemented in Isabelle as Eisbach proof methods [57] that exercise the security type system rules [65], which I extended [84] for the locking support added by Chapter 4.

User intervention is required in two kinds of cases:

1. Sometimes the automatic typing inference rule for if-conditionals from [67] (and a fully-automated Eisbach method using it) is not enough to guess an adequate

final typing environment (including predicates expected to hold) at the end of both alternatives of the if-conditional. In these cases the user needs to intervene by applying the full IF typing rule, in order to specify the final typing environment with which to proceed with further type-checking once the conditional has completed.

For the CDDC model, this is required only twice, during the verification of the SWITCH component:

- In one instance, it is to reflect a typing environment rewrite (described in more detail below), which makes both final typing environments of an if-conditional (to be specific, the one depicted in Figure 6.6b) consistent with each other.
- In the other instance, it is to throw away predicates deduced by the type system only on one side of an if-conditional branch (wherein SWITCH handles a mouse event if present), that are unneeded after that branch has completed.

2. Sometimes, it is necessary to rewrite the typing environment using the security type system's REWRITE rule from [67].

   For the CDDC model, after SWITCH copies in the data from the input-event buffer (as depicted in Figure 6.6b), I rewrite the security type of the local copy of the incoming event, to make this security type depend only on its own local copies of control variables for the incoming data.

Use of the local guarantee compliance check of Murray et al. [65] (extended for lock support by Section 4.2) is also automated using Eisbach methods, and in this case no user intervention is required when applying it to any components of the CDDC model.

6.6 Confidentiality-preserving compilation to RISC model

I now turn to applying the COVERN wr-compiler of Chapter 5 to my While-language model of the CDDC's HID switch; we then have automatically that it preserves the security properties (as proved in Section 6.4) down to the compiler's RISC-language output.

The wr-compiler is *executable* in the Isabelle proof assistant [84]. Using Isabelle's eval tactic, I execute the wr-compiler's main function, compile-cmd (whose implementation was described in Section 5.2) 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 5.2 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 6.5 (RISC-language program texts of CDDC model's components)**.**

$$\begin{aligned} \text{OVERLAY}_{\texttt{RISC}} &\triangleq \mathsf{map\ fst\ (fst\ (compile\text{-}cmd\ C_0\ None\ 0\ OVERLAY))} \\ \text{INPUT}_{\texttt{RISC}} &\triangleq \mathsf{map\ fst\ (fst\ (compile\text{-}cmd\ C_0\ None\ 0\ INPUT))} \\ \text{SWITCH}_{\texttt{RISC}} &\triangleq \mathsf{map\ fst\ (fst\ (compile\text{-}cmd\ C_0\ None\ 0\ SWITCH))} \end{aligned}$$

My 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 5.21). Recall, this was:

Theorem 5.21 (Preservation of per-thread confidentiality by the wr-compiler)**.**

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

Note that this only works for non–High-branching programs. Fortunately, a lemma present in Murray et al. [65] (and unaffected by the changes in Chapter 4) gives us that when a thread program typechecks with final typing environment $\Gamma', \mathcal{S}', P'$, the bisimulation $\mathcal{B}_{\Gamma', \mathcal{S}', P'}$ constructed by the security type system enforces no-high-branching (Definition 5.5). Thus, applying the security type system in fact yields $\mathsf{com\text{-}secure}^{\mathsf{no\text{-}high\text{-}branching}}$, a stronger property than com-secure. (The same will not be able to be said after I update the security type system in Chapter 7 to support High-branching.) This is enough to give us the needed per-thread properties for preservation by the wr-compiler via Theorem 5.21:

Lemmas 6.6 (Even stronger per-thread confidentiality results for CDDC `While` model)**.**

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

Proof. As mentioned above, the security type system is sound for $\mathsf{com\text{-}secure}^{\mathsf{no\text{-}high\text{-}branching}}$. Then once again I use Lemma 6.2 to obtain $\mathsf{com\text{-}secure}^{\mathsf{no\text{-}high\text{-}branching}}_{\mathsf{no\text{-}locks\text{-}held}}$ for free. □

Then, for compile-cmd to execute successfully (i.e. to return *failed* = False), the model must pass the stability-checks discussed in Section 5.2. 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 security typecheck and local compliance check of Chapter 4.

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

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

Proof. A few approaches are available; I obtained formal proofs of this theorem in Isabelle/HOL [84] using all three of the following alternatives:

Option 1. Use Theorem 5.26 (unfolding Definition 6.5), 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 6.6). 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 $\textsc{Overlay}_{\texttt{RISC}}$, $\textsc{Input}_{\texttt{RISC}}$, and $\textsc{Switch}_{\texttt{RISC}}$ were obtained in this manner.

Option 2. Use Theorem 5.21 and Lemma 5.24 to obtain $\mathsf{com\text{-}secure}^{\mathsf{pc\text{-}security}}_{\mathsf{no\text{-}locks\text{-}held}}$ and local-mode-compliance (resp.) for each of $\textsc{Overlay}_{\texttt{RISC}}$, $\textsc{Input}_{\texttt{RISC}}$, and $\textsc{Switch}_{\texttt{RISC}}$, and then use the compositionality result of Murray et al. [67] (Theorem 3.8) 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. However, Lemma 5.25 can still be used to prove the global-modes-compatibility requirement, provided all threads are initialised with mds_0.

Option 3. Use the more generic whole-system refinement framework from Murray et al. [67] (as formalised in Isabelle/HOL by [66]). This option can be used for systems where all of the `RISC` threads are secure refinements (according to Definition 3.15) of some `While` program that satisfied sound-mode-use (with no-locks-held initially), but some might be obtained by other means than the wr-compiler (i.e. not all via the refinement $\mathcal{R}_{\mathsf{wr}}$). □

6.7 Publications and acknowledgements

Further work, building on the *modelling and program verification efforts* of Sections 6.1–6.5 for this case study, has appeared in the publication already noted in Section 4.5:

- [68] Toby Murray, Robert Sison, and Kai Engelhardt. Covern: A logic for compositional verification of information flow control. In *European Symposium on Security and Privacy*, pages 16–30, London, United Kingdom, April 2018. IEEE.

In that publication, an environmental property checked at runtime by the `While` model in this chapter (described in Section 6.3) is instead enforced as an explicit assume–guarantee condition by the Covern program logic derived from the work of Chapter 4. Some functional schematic illustrations (Figures 6.1, 6.2) are also reproduced from that publication.

Meanwhile, a precursor to the *verified compiler-based security preservation result* of Section 6.6—for a much earlier `While` model of this case study—appeared in the publication already noted in Sections 4.5 and 5.6:

- [85] Robert Sison and Toby Murray. Verifying that a compiler preserves concurrent value-dependent information-flow security. In *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141, pages 27:1–27:19, Portland, USA, September 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

That publication presented the compilation to `RISC` of a 2-component precursor to this chapter's 3-component model; this precursor consisted of a combined Input–Switch component, alongside the Overlay driver component largely as described in this chapter.

6.8 Summary

This chapter has validated the main claim of this thesis: that proving confidentiality and its preservation by a compiler is feasible for mixed-sensitivity concurrent programs. The case study presented here demonstrated applicability of the techniques presented by Chapter 4, and of the proved results of Chapter 5, respectively to:

1. **Formal verification of confidentiality for a concurrent program with mixed-sensitivity reuse of system resources, driven by a real-world use case.**

   This is a concrete demonstration to program verifiers that it is possible to prove confidentiality for such programs in a compositional manner.

2. **Preservation of the verified confidentiality property, via a compiler verification result, down to a model of the program in a lower-level language.**

   This is a confirmation to verified compiler developers that it is possible to offer a compiler capable of preserving formally proved confidentiality properties, even in the presence of such complicating characteristics.

Here I modelled in the `While` language of Chapter 4 the HID switch for the Cross Domain Desktop Compositor [13] (Sections 6.1–6.3). I then proved this model secure using the program verification techniques of Chapter 4 (Sections 6.4–6.5). Finally, deriving from that formal proof of security, I used the compiler verification result of Chapter 5 to obtain automatically that the compilation of that `While` model yields a secure `RISC` model of the program (Section 6.6). As noted in Section 6.4, doing so required the `While` model to use read-atomicity locks as described in Section 5.2.1 to protect the atomicity of assignments needed for small-step semantic (and thereby timing-sensitive security) preservation.

Aside from validating this thesis' main claims, this case study illustrates characteristics of modelling and verification that a program verifier should expect from the interaction between value-dependent classifications needed for mixed-sensitivity reuse of resources, and concurrency of access to those resources. Furthermore, it highlights to compiler verifiers and programming language designers how the timing sensitivity of the confidentiality is reliant on the atomicity of the source and target languages' small-step semantics.

Chapter 7

Security typing extension for secret-dependent control flow

This chapter presents a fourth and final contribution: the first instance of syntax-directed reasoning about secret-dependent control flow for mixed-sensitivity concurrent programs. Here, I extend the security type system of Chapter 4 to support `While` programs that contain limited forms of conditional branching on secrets, and prove that it remains sound.

As explained in Section 2.2.5, an **if**-conditional branch on a secret can cause implicit flows of that secret, where the two possible control-flow paths have different observable effects; when execution time differs, this is a timing leak. Banning *all* branching on secrets prevents all such implicit flows, and is a common practice against wall-clock (external) timing leaks[1] because many hardware platforms exhibit nondeterministic wall-clock timing behaviour. However, enough may be known about a scheduler's notion of time, to prevent the storage leaks caused by scheduler-relative (internal) timing leaks[2]; security analyses of such scope can be more precise than banning all secret-dependent control flow.

To this end, this chapter relaxes the `While`-language security type system from Murray et al. [67] and Chapter 4, to allow secret-dependent **if**-conditionals, while still remaining sound with respect to this thesis' purpose of preventing storage leaks that stem from internal timing leaks. This entails adding a new typing rule, IfH, which prevents both (1) internal timing leaks (assuming an instruction-based scheduler that counts time in `While`-language evaluation steps), and (2) implicit flows to untrusted memory locations.

This chapter proceeds as follows. Section 7.1 presents the intuitions, and Section 7.2 the formal definition, of the new typing rule, IfH. Then, Section 7.3 presents an update to the bisimulation construction, which is crucial for the re-establishment of a proof of soundness. Ultimately, Section 7.4 presents proof that the updated security type system, now supporting limited forms of branching on secrets, remains sound for proving concurrent value-dependent noninterference (com-secure, Definition 3.7). Section 7.5 summarises.

[1] To an outsider, as measured by an independent "wall clock" that is external to the system.

[2] Between threads, as measured in the timing units by which the system's internal scheduler makes scheduling decisions, and at which it can interleave the threads' execution. (Section 1.1.1)

```
if (h1 != 0) then
  h2 := 1
else
  h3 := 2
fi;
l := 3
```

(a) Assigning to High-classified variables. Note the final assignment, to an unprotected Low-classified variable, would be a timing leak if the branches took a different number of steps.

```
lock(k);
if (h1 != 0) then
  l := 1
else
  l := 2
fi;
l := 3;
unlock(k);
```

(b) Assigning to a read-locked Low-classified variable. The final assignment, erasing the h1-dependent storage leak before the unlock, is security critical.

```
lock(k);
if (h1 != 0) then
  h1 := h2 && h3;
  skip;
  h2 := 3;
  h3 := 7
else
  skip;
  l := 2;
  h3 := h1;
  skip
fi;
l := 3;
unlock(k);
```

(c) Both kinds of assignment, plus padding so both branches take the same number of steps.

Figure 7.1: Examples of secret-dependent control flow admitted by typing rule IfH. In all of these examples, `l` is a Low-classified variable, `h1`, `h2`, etc. are High-classified variables, and `k` is a lock granting exclusive read–write access to `l`.

7.1 Intuitions behind the new security typing rule IfH

In short, the new typing rule allows **if**-conditional branching **if** h **then** c_1 **else** c_2 **fi** on secrets in h, but restricts the behaviours of the two branches c_1 and c_2, so that they have no observable differences between each other in their (1) timing and (2) memory effects.

The restrictions imposed by IfH are permissive enough to admit `While`-language programs that branch on High-sensitivity information to operate on variables that are classified High (as in Figure 7.1a) or assumed not to be readable by other threads (as in Figure 7.1b), as long as those programs ensure that the two branches have no discernible timing difference (for example, by padding out the shorter of the two branches, as in Figure 7.1c)—even if they contain nested branching (as in Figure 7.2, on page 108).

To this end, the new typing rule will enforce the running times of both branches c_1 and c_2 to be not only equal to each other, but also *a constant (statically determined) number of steps for all executions.* Consequently, the following restrictions will apply:

- Any **if**-conditionals nested inside each of c_1 and c_2 must themselves have branches whose running times are a constant over all executions, and equal to each other.
- Attempts to **while**-loop or to acquire any locks in either of c_1 or c_2 are banned, because they could result in variable running times.[3]

Then to prevent any observable differences between c_1 and c_2's memory effects, the new typing rule will go further and *disallow any observable memory effects* in either of c_1 or c_2.[4] This will come in the form of the following restrictions to c_1 and c_2:

[3] I leave for future work any `While`-language and type system support for **for**-loops, which could conceivably be judged whether or not to have a constant running time from their loop guards.

[4] Strictly speaking, this is not always necessary, as two branches could have observable memory effects as long as those effects are both identical, and well-timed to occur at exactly the same moment. But it

- Attempts to assign values to memory locations considered observable by the security property (i.e. $v := e$ for any readable and Low-classified v) are explicitly banned, *regardless of the sensitivity of the value being assigned* (i.e. of the expression e).

  As this restriction implicitly applies to all control variables—due to their always being treated as observable by the security property—it also automatically prevents any changes to the observability of variables via their value-dependent classification.

- Attempts to acquire and release locks are banned, as (1) acquisition attempts can take variable running time, and (2) restrictions to locking disciplines (in Section 4.1.2) render any changes to lock variables as observable by the security property.[5]

  Note this automatically prevents any changes to the observability of variables via changes to assumptions about their readability by other threads.

The upshot of this approach is that well-typedness need not apply to the branches c_1 or c_2 of any secret-dependent **if**-conditionals, nor to any of c_1 or c_2's intermediate states. Upon identifying a secret-dependent **if**-conditional, the updated security type system need only ensure that the aforementioned restrictions on observable effects apply to c_1 and c_2, and its analysis can expect the variables considered observable not to change because those restrictions prevent changes to control variables and lock state. Consequently, simple static checks will suffice to enforce these requirements, whose implementations are much simpler than the rest of the security type system (whose complexity stems from tracking information to account for possibilities excluded by the bans above).

However, the price paid for the new typing rule tracking less information (in no longer enforcing well-typedness throughout c_1 and c_2) is that it makes no guarantees afterwards about the state of any variables that were modified by either of c_1 or c_2 (it throws away the relevant predicates), nor does it retain any nuanced typing information about those variables (it raises all of their security types to High).

7.2 Updates to `While`-language security type system

I now move on to describing the implementation of the scheme outlined in Section 7.1, as an extension to the security type systems of Murray et al. [67, 65] and Chapter 4. (For more formal definitions, see Appendix B or the Isabelle/HOL theories for this thesis [84].)

To determine running times and prevent them from being secret-dependent, I define a function H-region-steps :: *cmd* ⇒ *nat option*, which returns Some n only if the given `While`-language command always completes in a constant time of n evaluation steps. To this end, it imposes the checks mentioned in Section 7.1: that c does not attempt to acquire or release any locks, nor engage in any **while**-loops, and that if it does branch on

is arguably a reasonable restriction, because any conditional-branching that does this could just as easily be refactored to move these simultaneous observable memory effects out of the **if**-conditional—from that perspective, there is really no point in putting them in a secret-dependent conditional branch at all.

[5] Recall that these restrictions were chosen precisely to prevent secrets from leaking into the state of any lock, from which they would leak into the control flow of any threads attempting to acquire it.

an **if**-conditional, then both alternative control flow paths must likewise complete simultaneously in constant-time. If any are violated, it returns None.

Then, I define a predicate that the new security typing rule will use to ensure that no changes to visible memory may occur in secret-dependent control flow paths. This predicate, vars-modified-NoRW-or-H $c\,\mathcal{S}$, asserts that all variables modified by any potential control-flow path of a given program c either (1) are statically classified High, or (2) are non-control variables over which the current thread is tracked (by $\mathcal{S}$) as having read–write-exclusive access. Note that these are precisely the ways by which shared variables can be considered not to be visible by the security property (see Definition 3.3).

Finally, I define two helpers intended to make "safe overapproximations" of the effects that a region of secret-dependent control flow might have on the typing environment and predicate set, which will be used for further type checking after the completion of the **if**-conditional. Stated simply, we will be pessimistically regarding all variables potentially modified in that region to contain sensitive data, and making no promises about any predicates that mention variables potentially modified during that time:

- The helper tyenv-raise-modified $\Gamma\ c$ effectively raises to High the sensitivity tracked by typing environment Γ for the contents of all variables potentially modified by c.
- The helper preds-remove-modified $P\ c$ then removes all predicates from a given predicate set P that mention any variables potentially modified by the program c.

The new security typing rule IFH then applies when the **if**-conditional expression e is judged to be dependent on a secret, on account of the security type t tracked by Γ for e (i.e. $\Gamma \vdash e : t$) *not* being Low according to the current predicate set P (i.e. $\neg\, P \vdash t$). After using H-region-steps and vars-modified-NoRW-or-H to implement the checks described in Section 7.1, its remaining guards assert that the final typing environment Γ' be "equivalent" to the Γ_H (resp. final predicate set P' be entailed by the P_H) that took "all bets off" for all variables touched by the branches c_1 and c_2 of the secret-dependent **if**-conditional. (See Appendix B for formal definitions for the above notions from Murray et al. [67].)

Definition 7.1 (Security typing rule for secret-dependent **if**-conditional branching)**.**

$$\frac{\begin{array}{c}\Gamma \vdash e : t \qquad \neg\, P \vdash t \\ \text{vars-modified-NoRW-or-H } c_1\ \mathcal{S} \qquad \text{vars-modified-NoRW-or-H } c_2\ \mathcal{S} \\ \text{H-region-steps } c_1 \neq \text{None} \qquad \text{H-region-steps } c_1 = \text{H-region-steps } c_2 \\ \Gamma_H = \text{tyenv-raise-modified (tyenv-raise-modified } \Gamma\ c_1)\ c_2 \\ P_H = \text{preds-remove-modified (preds-remove-modified } P\ c_1)\ c_2 \\ \Gamma_H =:_{P_H} \Gamma' \qquad P_H \vdash P' \\ \forall mds.\ \text{tyenv-wellformed } mds\ \Gamma_H\ \mathcal{S}\ P_H \longrightarrow \text{tyenv-wellformed } mds\ \Gamma'\ \mathcal{S}\ P'\end{array}}{\vdash \Gamma, \mathcal{S}, P\ \{\textbf{\textit{if}}\ e\ \textbf{\textit{then}}\ c_1\ \textbf{\textit{else}}\ c_2\ \textbf{\textit{fi}}\}\ \Gamma', \mathcal{S}, P'}\ \textsc{IfH}$$

7.3 Updates to bisimulation construction

The addition of the new typing rule IFH means that the security type system is now liable to allow programs that may take a different control flow path depending on secret information. However, the strong low-bisimulation (modulo modes) construction $\mathcal{B}_{\Gamma',\mathcal{S}',P'}$ defined by Murray et al. [67, 65] did not allow pairing any program configurations that are not at the same location (i.e. do not have the same `While` command). As this construction will be crucial (in Section 7.4) for re-proving the soundness of the updated security type system, this section will describe the updates I made for it to capture the wider set of configuration pairings that may result from programs admitted by IFH.

To explain what needs to be added, I need to dive into further details of $\mathcal{B}_{\Gamma',\mathcal{S}',P'}$ than I did in Chapter 4. In Murray et al. [67, 65], and left unchanged throughout the security type system improvements described in Chapter 4, this was defined as the union $\mathcal{B}_{\Gamma',\mathcal{S}',P'} \triangleq \mathcal{B}^1_{\Gamma',\mathcal{S}',P'} \cup \mathcal{B}^3_{\Gamma',\mathcal{S}',P'}$ of two components, with the first ($\mathcal{B}^1_{\Gamma',\mathcal{S}',P'}$) responsible for pairing atomic commands, and the other ($\mathcal{B}^3_{\Gamma',\mathcal{S}',P'}$) responsible for pairing sequences of commands that both have a common, well-typed suffix. I recount their definitions from the Isabelle/HOL formalisation [65] of Murray et al. [67] as follows:

Reproduced definitions (Bisimulation construction components, Murray et al. [67])**.**

$$\begin{aligned}
\mathcal{B}^1_{\Gamma',\mathcal{S}',P'} \triangleq \{\ & (\langle c, mds, mem_1\rangle, \langle c, mds, mem_2\rangle) \mid \exists \Gamma\ \mathcal{S}\ P. \\
& \vdash \Gamma, \mathcal{S}, P\ \{c\}\ \Gamma', \mathcal{S}', P' \wedge \\
& \mathsf{tyenv\text{-}wellformed}\ mds\ \Gamma\ \mathcal{S}\ P \wedge mem_1 =_\Gamma mem_2 \wedge \\
& \mathsf{preds\text{-}hold}\ P\ mem_1 \wedge \mathsf{preds\text{-}hold}\ P\ mem_2 \wedge \mathsf{tyenv\text{-}sec}\ mds\ \Gamma\ mem_1\ \} \\
\mathcal{B}^3_{\Gamma',\mathcal{S}',P'} \triangleq \{\ & (\langle c_1\,;c, mds, mem_1\rangle, \langle c_2\,;c, mds, mem_2\rangle) \mid \exists \Gamma\ \mathcal{S}\ P. \\
& (\langle c_1, mds, mem_1\rangle, \langle c_2, mds, mem_2\rangle) \in \mathcal{B}^1_{\Gamma,\mathcal{S},P} \cup \mathcal{B}^3_{\Gamma,\mathcal{S},P} \wedge \\
& \vdash \Gamma, \mathcal{S}, P\ \{c\}\ \Gamma', \mathcal{S}', P'\ \}
\end{aligned}$$

This partially followed the structure of the bisimulation $\mathcal{R}^{\Gamma'} \triangleq \mathcal{R}^{\Gamma'}_1 \cup \mathcal{R}^{\Gamma'}_2 \cup \mathcal{R}^{\Gamma'}_3$ (whose $\mathcal{R}^{\Gamma'}_1, \mathcal{R}^{\Gamma'}_3$ have the same roles as $\mathcal{B}^1_{\Gamma',\mathcal{S}',P'}, \mathcal{B}^3_{\Gamma',\mathcal{S}',P'}$ described above) given by the Isabelle/HOL formalisation [33] of Mantel et al. [54] for their security type system, on which ours was based. The $\mathcal{R}^{\Gamma'}_2$ component of that construction (missing an analogue in Murray et al. [67, 65]) served to relate two individually type-checked branches of a secret-dependent conditional, but only if the user of the theory could obtain and supply a proof of bisimulation between them without any further help from their security type system.

In this thesis, I provide a new component $\mathcal{B}^2_{\Gamma',\mathcal{S}',P'}$ for the bisimulation construction, which (like $\mathcal{R}^{\Gamma'}_2$ of Mantel et al. [54]) will be responsible for capturing pairings between alternative branches of a secret-dependent conditional. In this case however, I will prove (in Section 7.4) that membership of $\mathcal{B}^2_{\Gamma',\mathcal{S}',P'}$ follows automatically from the application of the new security type system rule IFH (Definition 7.1) and its helpers.

The key differences of $\mathcal{B}^2_{\Gamma',\mathcal{S}',P'}$ from what has already been seen are as follows:

- Unlike $\mathcal{B}^1_{\Gamma',\mathcal{S}',P'}$ (and $\mathcal{R}^{\Gamma'}_2$ of Mantel et al. [54]), and as mentioned in Section 7.1, $\mathcal{B}^2_{\Gamma',\mathcal{S}',P'}$ will not assert the well-typedness of any programs.
- Instead of applying tyenv-raise-modified and preds-remove-modified directly (as IFH does to obtain "pessimistic" Γ_H, P_H for the branches on entry), it will instead enforce that nominated Γ, P are "sufficiently pessimistic" for the branch programs left to be executed, using the following helpers (defined formally in Appendix B):
  - Predicate vars-modified-tyenv-H c Γ' serves to specify the intended outcome of applying $\Gamma' =$ tyenv-raise-modified Γ c: It effectively asserts that all variables modified by c have type High in typing environment Γ'.
  - Predicate vars-modified-no-preds c P' likewise serves to specify the intended outcome of applying $P' =$ preds-remove-modified P c: No predicates in set P' should mention any variables modified by c.

Apart from these differences, the formal definition of $\mathcal{B}^2_{\Gamma',\mathcal{S}',P'}$ bears a number of features already seen in this chapter (summarised immediately after this definition):

Definition 7.2 (Bisimulation component for secret-dependent conditional branches)**.**

$$\begin{aligned}\mathcal{B}^2_{\Gamma',\mathcal{S}',P'} \triangleq \{\ (\langle c_1, mds, mem_1\rangle, \langle c_2, mds, mem_2\rangle) \mid \exists \Gamma\ P.\\ \quad \textsf{tyenv-wellformed}\ mds\ \Gamma\ \mathcal{S}'\ P\ \wedge\ mem_1 =_\Gamma mem_2\ \wedge\\ \quad \textsf{preds-hold}\ P\ mem_1\ \wedge\ \textsf{preds-hold}\ P\ mem_2\ \wedge\ \textsf{tyenv-sec}\ mds\ \Gamma\ mem_1\ \wedge\\ \quad \textsf{vars-modified-NoRW-or-H}\ c_1\ \mathcal{S}'\ \wedge\ \textsf{vars-modified-NoRW-or-H}\ c_2\ \mathcal{S}'\ \wedge\\ \quad \textsf{H-region-steps}\ c_1 \neq \textsf{None}\ \wedge\ \textsf{H-region-steps}\ c_1 = \textsf{H-region-steps}\ c_2\ \wedge\\ \quad \textsf{vars-modified-tyenv-H}\ c_1\ \Gamma\ \wedge\ \textsf{vars-modified-tyenv-H}\ c_2\ \Gamma\ \wedge\\ \quad \textsf{vars-modified-no-preds}\ c_1\ P\ \wedge\ \textsf{vars-modified-no-preds}\ c_2\ P\ \wedge\\ \quad \Gamma =:_P \Gamma'\ \wedge\ P \vdash P'\ \wedge\\ \quad (\forall mds'.\ \textsf{tyenv-wellformed}\ mds'\ \Gamma\ \mathcal{S}'\ P \longrightarrow \textsf{tyenv-wellformed}\ mds'\ \Gamma'\ \mathcal{S}'\ P')\}\end{aligned}$$

In short, this new bisimulation component $\mathcal{B}^2_{\Gamma',\mathcal{S}',P'}$ asserts:

- the same checks as the component $\mathcal{B}^1_{\Gamma',\mathcal{S}',P'}$ (reproduced from Murray et al. [67] earlier in this section) with respect to the mode state and memories of the configurations being paired, regarding: well-formedness, memory equality according to a typing environment, that all predicates must hold, and type environment security.
- the same checks as IFH (Definition 7.1), enforcing that both branches left to be executed will only modify variables considered non-observable by the security property (according to vars-modified-NoRW-or-H), and take the same number of H-region-steps.
- the same relationships that IFH asserts between the nominated (Γ, P) and the final (Γ', P') extended typing environment, regarding typing environment equivalence,

predicate set entailment, and preservation of well-formedness. Furthermore, like IFH it does not permit changes to the stable set $\mathcal{S}$ (owing to the ban on locking).

Finally, I redefine the bisimulation construction $\mathcal{B}_{\Gamma',\mathcal{S}',P'}$ (and its component for sequential composition, $\mathcal{B}^3_{\Gamma',\mathcal{S}',P'}$) to include this new component $\mathcal{B}^2_{\Gamma',\mathcal{S}',P'}$:

Definition 7.3 (New bisimulation construction, modified from Murray et al. [67])**.**

$$\begin{aligned}
\mathcal{B}^3_{\Gamma',\mathcal{S}',P'} \triangleq \{\ (\langle c_1\,;c, mds, mem_1\rangle, \langle c_2\,;c, mds, mem_2\rangle) \mid \exists \Gamma\ \mathcal{S}\ P.\\
(\langle c_1, mds, mem_1\rangle, \langle c_2, mds, mem_2\rangle) \in \mathcal{B}^1_{\Gamma,\mathcal{S},P} \cup \mathcal{B}^2_{\Gamma,\mathcal{S},P} \cup \mathcal{B}^3_{\Gamma,\mathcal{S},P} \wedge\\
\vdash \Gamma, \mathcal{S}, P\ \{c\}\ \Gamma', \mathcal{S}', P'\ \}\\
\mathcal{B}_{\Gamma',\mathcal{S}',P'} \triangleq \mathcal{B}^1_{\Gamma',\mathcal{S}',P'} \cup \mathcal{B}^2_{\Gamma',\mathcal{S}',P'} \cup \mathcal{B}^3_{\Gamma',\mathcal{S}',P'}
\end{aligned}$$

7.4 Proof of soundness of the new typing rule

The proof of soundness consists of two parts:

1. Proof that the updated bisimulation construction satisfies the requirements of an adequate witness to the security of the program. (Section 7.4.1)
2. Proof that the updated bisimulation construction captures the evaluation steps of programs admitted by the updated type system. (Section 7.4.2)

7.4.1 The new bisimulation construction is still a security witness

Re-establishing that the updated construction $\mathcal{B}_{\Gamma',\mathcal{S}',P'}$ (Definition 7.3) can serve as a witness to com-secure (Definition 3.7) entails proving that it is still a strong low-bisimulation (modulo modes)—this was reproduced from Murray et al. [67] in Section 3.1 as follows:

Definition 3.4 (Strong low bisimulation, modulo modes)**.**

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

My approach will be to prove that the requirements demanded of the overall bisimulation $\mathcal{B}_{\Gamma',\mathcal{S}',P'}$ apply—directly if possible, or approximately—to its new component $\mathcal{B}^2_{\Gamma',\mathcal{S}',P'}$ (Definition 7.2), and then to use these facts to re-establish them for $\mathcal{B}_{\Gamma',\mathcal{S}',P'}$ as a whole.

I begin with the cg-consistent (Definition 3.6) and symmetry requirements:

Lemma 7.4 ($\mathcal{B}^2_{\Gamma',\mathcal{S}',P'}$ is closed under globally consistent changes)**.**

$$\mathsf{cg\text{-}consistent}\ \mathcal{B}^2_{\Gamma',\mathcal{S}',P'}$$

Proof. The only 4 conditions of $\mathcal{B}^2_{\Gamma',\mathcal{S}',P'}$ to mention memory are identical to the 4 that do so in $\mathcal{B}^1_{\Gamma',\mathcal{S}',P'}$, so I adapt straightforwardly an existing proof for cg-consistent $\mathcal{B}^1_{\Gamma',\mathcal{S}',P'}$.

In short, the condition $mem_1 =_\Gamma mem_2$ would have to be violated by a change to a variable not tracked by Γ, because Γ only tracks stable variables (accounted for by $\mathcal{S}$) currently assumed not to be written by other threads. But for variables not tracked by Γ, this effectively simplifies to $mem_1 =^{\mathsf{Low}}_{mds} mem_2$ (for a mds consistent with $\mathcal{S}$), and this is something already asserted by cg-consistent when considering changes to such variables.

Then, the remaining three conditions (preds-hold P mem_1, preds-hold P mem_2, and tyenv-sec mds Γ mem_1) cannot be violated because that would similarly require changes to variables assumed to be stable. □

Lemma 7.5 ($\mathcal{B}^2_{\Gamma',\mathcal{S}',P'}$ is symmetric)**.**

$$\mathsf{sym}\ \mathcal{B}^2_{\Gamma',\mathcal{S}',P'}$$

Proof. The only nontrivial asymmetry in Definition 7.2 is that tyenv-sec is asserted only for mem_1, but it is rather straightforwardly obtained for mem_2 using Isabelle's `metis` tactic given $mem_1 =_\Gamma mem_2$ (itself symmetric) along with a number of relevant definitions. □

This leaves the third conjunct of Definition 3.4, which asserts that the bisimulation maintains low-equivalence of memory modulo modes ($=^{\mathsf{Low}}_{mds}$, Definition 3.3), and that it "progresses to itself" (maintaining modes-equality). The first of these is true because all three components of $\mathcal{B}_{\Gamma',\mathcal{S}',P'}$ (Definition 7.2 of $\mathcal{B}^2_{\Gamma',\mathcal{S}',P'}$ included) assert equivalence of memories according to the typing environment ($=_\Gamma$, Definition 4.25), and this implies low-equivalence modulo modes ($=^{\mathsf{Low}}_{mds}$, Definition 3.3 for mds for which Γ is wellformed) via a pre-existing lemma [65]. Thus for $\mathcal{B}^2_{\Gamma',\mathcal{S}',P'}$ in particular, we have:

Lemma 7.6 ($\mathcal{B}^2_{\Gamma',\mathcal{S}',P'}$ maintains low-equivalence of memory modulo modes)**.**

$$\forall lc_1\ lc_2.\ (lc_1, lc_2) \in \mathcal{B}^2_{\Gamma',\mathcal{S}',P'}\ \wedge\ lc_1 =_{\mathsf{mds}} lc_2\ \longrightarrow\ lc_1 =^{\mathsf{Low}}_{\mathsf{mds}} lc_2$$

Therefore, it remains just to prove that $\mathcal{B}_{\Gamma',\mathcal{S}',P'}$ "progresses to itself". To this end, I first prove a somewhat weaker lemma for $\mathcal{B}^2_{\Gamma',\mathcal{S}',P'}$ that does not assert the existence of any steps, but merely that if both configurations take a step, then their destinations remain modes-equal and in the relation:

Lemma 7.7 ($\mathcal{B}^2_{\Gamma',\mathcal{S}',P'}$ is closed under any modes-equal pairwise steps that may occur)**.**

$$\begin{aligned}&\forall lc_1\ lc_2.\ (lc_1, lc_2) \in \mathcal{B}^2_{\Gamma',\mathcal{S}',P'}\ \wedge\ lc_1 =_{\mathsf{mds}} lc_2 \longrightarrow\\ &\quad (\forall lc_1'\ lc_2'.\ lc_1 \rightsquigarrow lc_1'\ \wedge\ lc_2 \rightsquigarrow lc_2'\ \wedge\ lc_1' =_{\mathsf{mds}} lc_2'\ \longrightarrow\ (lc_1', lc_2') \in \mathcal{B}^2_{\Gamma',\mathcal{S}',P'})\end{aligned}$$

Proof. We are given that the battery of checks asserted by $\mathcal{B}^2_{\Gamma',\mathcal{S}',P'}$ (Definition 7.2) holds for some $\Gamma, \mathcal{S}, P$ before a pairwise evaluation step, and are obliged to show that they still hold for the new $c_1, c_2, mds, mem_1, mem_2$ after the step.

It is easiest just to choose the same $\Gamma, \mathcal{S}, P$, which discharges the first few checks.

The next few "vars-modified-" checks (each asserting that the variables modified by c_1, c_2 satisfy some requirement relating to $\Gamma, \mathcal{S}', P$) only ever get easier to prove, as the set of variables still yet to be modified by c_1, c_2 only shrinks as their executions progress.

For the remaining checks, I prove and use a number of smaller lemmas ensuring that `While`-language programs captured by $\mathcal{B}^2_{\Gamma',\mathcal{S}',P'}$ satisfy the intuitions described in Section 7.1, by running in constant time, and having "no visible effects": They do not assign to any Low-observable variables, nor invoke any locking primitives.

Consequently, we know that the evaluation steps each make no changes to mode state, and only change memory in such a way that preserves $=_\Gamma$ (Definition 4.25). Furthermore, the vars-modified-no-preds check ensured that P cannot mention any variables modified by c_1, c_2, therefore the preds-hold checks are maintained. Then the absence of visible effects ensures that there are no changes to control variables; this rules out any instances of *unsafe downgrade*—whereby a program lowers a variable's classification while it contains sensitive data—which is the only remaining possibility of violation of the tyenv-sec check.

Then, the final two checks are discharged by a lemma that H-region-steps simply returns a decremented result for the successor of a program already checked by H-region-steps: This reflects that the programs remain constant-time, but one step closer to finishing. □

The result I just proved (Lemma 7.7) will be used with the following, that $\mathcal{B}^2_{\Gamma',\mathcal{S}',P'}$ asserts consistent stopping behaviour for configurations with the same mode state:

Lemma 7.8 ($\mathcal{B}^2_{\Gamma',\mathcal{S}',P'}$ asserts consistent stopping behaviour for modes-equal states)**.**

$$\forall lc_1\ lc_2.\ (lc_1, lc_2) \in \mathcal{B}^2_{\Gamma',\mathcal{S}',P'}\ \wedge\ lc_1 =_{\mathsf{mds}} lc_2\ \longrightarrow\ \mathsf{stops}\ lc_1 = \mathsf{stops}\ lc_2$$

Proof. $\mathcal{B}^2_{\Gamma',\mathcal{S}',P'}$ (Definition 7.2) asserts that both programs have the same constant number of steps remaining according to H-region-steps, which in turn only ever gives zero for **stop**, the only `While`-language command that does not evaluate (as demanded by stops). □

These enable proving a lemma (established previously for $\mathcal{B}^1_{\Gamma',\mathcal{S}',P'}$ and $\mathcal{B}^3_{\Gamma',\mathcal{S}',P'}$ by [65]) that any pair related by $\mathcal{B}^2_{\Gamma',\mathcal{S}',P'}$ "progresses to" the overall bisimulation construction $\mathcal{B}_{\Gamma',\mathcal{S}',P'}$. Note that—unlike Lemma 7.7—this asserts the existence of a matching step:

Lemma 7.9 ($\mathcal{B}^2_{\Gamma',\mathcal{S}',P'}$ progresses to $\mathcal{B}_{\Gamma',\mathcal{S}',P'}$)**.**

$$\begin{aligned}\forall lc_1\ lc_2\ lc_1'.\ (lc_1, lc_2) \in \mathcal{B}^2_{\Gamma',\mathcal{S}',P'}\ \wedge\ lc_1 =_{\mathsf{mds}} lc_2\ \wedge\ lc_1 \rightsquigarrow lc_1'\ \longrightarrow\\ (\exists lc_2'.\ lc_2 \rightsquigarrow lc_2'\ \wedge\ lc_1' =_{\mathsf{mds}} lc_2'\ \wedge\ (lc_1', lc_2') \in \mathcal{B}_{\Gamma',\mathcal{S}',P'})\end{aligned}$$

Proof. Isabelle's automation support (`sledgehammer`) finds a proof that uses Lemma 7.7, Lemma 7.8, and the new introduction rule for $\mathcal{B}_{\Gamma',\mathcal{S}',P'}$ for its new $\mathcal{B}^2_{\Gamma',\mathcal{S}',P'}$ component. □

This covers the major lemmas about $\mathcal{B}^2_{\Gamma',\mathcal{S}',P'}$ that are analogous to properties demanded by strong-low-bisim-mm (Definition 3.4) of the overall $\mathcal{B}_{\Gamma',\mathcal{S}',P'}$.

There is also some impact on proofs of various lemmas about $\mathcal{B}^3_{\Gamma',\mathcal{S}',P'}$, because its new version (Definition 7.3) now references $\mathcal{B}^2_{\Gamma',\mathcal{S}',P'}$—however, the use of analogous lemmas proved about $\mathcal{B}^2_{\Gamma',\mathcal{S}',P'}$ is mostly straightforward to discharge these.

In all, these results are enough to prove the desired property (to remain a witness to the security property com-secure, by Definition 3.7) for the updated construction.

Lemma 7.10 (The updated construction is a strong low-bisimulation modulo modes)**.**

$$\mathsf{strong\text{-}low\text{-}bisim\text{-}mm}\ \mathcal{B}_{\Gamma',\mathcal{S}',P'}$$

Proof. Adaptation of the proof from Murray et al. [65] is trivial, adding invocations of Lemma 7.6 and Lemma 7.9 where required. □

7.4.2 The construction still captures well-typed program execution

I now turn to proving that the updated bisimulation construction (which I just proved in Section 7.4.1 still serves as an adequate security witness) captures the execution of well-typed programs admitted by the updated type system.

This falls to the "typed step capture" lemma (whose statement has not changed since Lemma 4.26, when I re-proved it for the added locking support in Chapter 4):

Lemma 7.11 (Well-typed program evaluation is captured by bisimulation $\mathcal{B}_{\Gamma',\mathcal{S}',P'}$)**.**

$$\frac{\begin{array}{c}\vdash \Gamma, \mathcal{S}, P\ \{c\}\ \Gamma', \mathcal{S}', P' \qquad mem_1 =_\Gamma mem_2 \\ \mathsf{tyenv\text{-}wellformed}\ mds\ \Gamma\ \mathcal{S}\ P \qquad \mathsf{preds\text{-}hold}\ P\ mem_1 \qquad \mathsf{tyenv\text{-}sec}\ mds\ \Gamma\ mem_1 \\ \langle c, mds, mem_1\rangle_{\mathsf{w}} \rightsquigarrow_{\mathsf{w}} \langle c_1', mds', mem_1'\rangle_{\mathsf{w}}\end{array}}{\begin{array}{c}\exists c_2'\ mem_2'.\ \langle c, mds, mem_2\rangle_{\mathsf{w}} \rightsquigarrow_{\mathsf{w}} \langle c_2', mds', mem_2'\rangle_{\mathsf{w}} \wedge \\ \langle c_1', mds', mem_1'\rangle_{\mathsf{w}}\ \mathcal{B}_{\Gamma',\mathcal{S}',P'}\ \langle c_2', mds', mem_2'\rangle_{\mathsf{w}}\end{array}}$$

Proof. By induction over the structure of the type system. As I noted in Section 7.1, the "preservation" lemma (Lemma 4.24) is now violated by IFH, and here I find a proof that no longer relies on it. In all other respects, existing cases are unchanged.

For the new IFH case, I invoke the new introduction rule for $\mathcal{B}_{\Gamma',\mathcal{S}',P'}$ for its new $\mathcal{B}^2_{\Gamma',\mathcal{S}',P'}$ component (Definition 7.2), nominating for Γ and P (resp.) the Γ_H raised by tyenv-raise-modified and the P_H trimmed by preds-remove-modified for the variables modified by both branches c_1 and c_2 of the if-conditional. It then follows straightforwardly that vars-modified-tyenv-H and vars-modified-no-preds—which were intended as specifications of these raise/trim functions—hold for both branches, as required by Definition 7.2.

The remaining checks required by Definition 7.2 that were not immediate from the assumptions are then satisfied immediately by the guards of IFH (Definition 7.1). □

Finally, I can re-prove the major security type system soundness result (whose statement has not changed from that of Theorem 4.27, when I re-proved it in Chapter 4):

Theorem 7.12 (Soundness of the security type system with IFH added)**.**

$$\frac{\vdash \Gamma_{mds}, \mathcal{S}_{mds}, \emptyset\ \{c\}\ \Gamma', \mathcal{S}', P' \qquad \mathsf{yields\text{-}stable\text{-}types}\ mds}{\mathsf{com\text{-}secure}\ (c, mem)}$$

Proof. As for Murray et al. [65] (and Theorem 4.27), but instead derived via the re-established Lemma 7.10 and Lemma 7.11 for the newly updated bisimulation construction $\mathcal{B}_{\Gamma',\mathcal{S}',P'}$ (Definition 7.3) and type system (with IFH, Definition 7.1) respectively. □

7.5 Summary

This chapter presented, as a final contribution of this dissertation, the first instance of syntax-directed reasoning about secret-dependent control flow for mixed-sensitivity concurrent programs. Here I demonstrated how a security type system for proving concurrent value-dependent noninterference [67] can soundly be relaxed to allow **if**-conditional branching on secrets, which is commonly banned to prevent implicit flows of those secrets.

This demonstration took the form of a security typing rule IFH for `While` that, rather than rejecting such **if**-conditionals outright as in Murray et al. [67] and Chapter 4, instead automatically detects and rejects them if they exhibit implicit flows (Sections 7.1–7.2). Proving the soundness of the resulting, more precise security type system (in Section 7.4) required me to adapt a component of the bisimulation construction of Mantel et al. [54] and specialise it (in Section 7.3) to witness the indistinguishability of the executions of the two alternative control flow paths of each **if**-conditional admitted by IFH.

This contribution shows the practical consequence of the timing sensitivity, needed for noninterference to be compositional across a program's threads, being relative to the notion of time used by the scheduler of those threads. As I argued in Sections 1.1.1 and 2.2.5, knowing how the scheduler "sees" time is both necessary and enough to prove the absence of the kinds of timing leaks that cause storage leaks in concurrent programs; thus I argue that analyses solely targeting such leaks can and should regain the full precision of reasoning about the time taken by a program in scheduler-relative timing units. Finally, this chapter demonstrated that the value-dependent classifications needed for mixed-sensitivity reuse are no obstacle to such precision; they can be handled straightforwardly through appropriate treatment of the control variables on which they depend.

```
while (1) do
  lock(x_lock);
  /* Note that, because this branch is on a (NoRW-locked) L variable, it is safe
     to acquire and release locks inside it, and there is no need to equalise
     its branch timing. */
  if (x != 0) then
    lock(y_lock);
    lock(z_lock);
    /* Branch on H-variable, opening the H-region.
       Both branches (and all internal nested branching) must have equal timing. */
    if (h1 = 1) then
      skip;
      skip;
      skip;
      h2 := 2;
      skip
    else
      /* Note this branch on (NoW-locked) L variable y. */
      if (y = 0) then
        skip;
        h3 := 3;
        skip;
        skip
      else
        /* Note assignment of a H value to a (NoRW-locked) L variable. */
        z := h2;
        if (h2 = 1) then
          /* Note this assignment to a NoRW-locked L variable. */
          x := 4;
          skip
        else
          /* Note that we are now branching on a H-sensitivity value held by the
             L-classified variable z. */
          if (z = 50) then
            skip
          else
            h1 := 5
          fi
        fi
      fi
    fi;
    /* We did not assign to y, so it is safe to unlock here. */
    unlock(y_lock);
    /* Note we must wipe x and z here, because they were assigned to inside the
       region of secret-dependent control flow. */
    x := 0;
    z := 0;
    unlock(z_lock)
  else
    /* No need to equalise the branch lengths of this branch on x, as it is
       L-classified. */
    skip
  fi;
  unlock(x_lock)
od
```

Figure 7.2: Example program admitted by typing rule IfH with nested branching.
In this example, x,y,z are Low-classified variables; h1, h2, etc. are High-classified variables; x_lock and z_lock grant exclusive read–write access to x, z (resp.); and y_lock grants exclusive read access to y.

Chapter 8

Conclusion

This dissertation has validated my central thesis: **Proving noninterference and its preservation under compilation is feasible for mixed-sensitivity concurrent programs**, using concurrent value-dependent notions of noninterference (CVDNI).

It did so by presenting, for the first time, proofs of the following: (Chapter 4) the requirements that make CVDNI compositional, (Chapter 5) that a compiler satisfies CVDNI-preserving refinement, and consequently, (Chapter 6) noninterference for a mixed-sensitivity concurrent program as a whole, preserved to another language by a compiler. It also achieves a secondary outcome laying a path for future work, by presenting (Chapter 7) the first instance of syntax-directed reasoning about secret-dependent branching for mixed-sensitivity concurrent programs.

This conclusion will recap what was shown by each of the achievements presented by this dissertation, and discuss their implications for software developers and future research.

8.1 What my work showed

The work I have presented in this dissertation showed that the central outcome of my thesis—formal proof of noninterference for a mixed-sensitivity concurrent program and its preservation by a compiler—is enabled by (1) assume–guarantee reasoning and (2) decomposition principles for refinement of confidentiality properties (resp. Sections 8.1.1, 8.1.2). My work also showed that the secondary outcome, source-level proof support for secret-dependent control flow in such programs, is enabled by CVDNI's timing sensitivity allowing syntax-directed reasoning about the time taken by control flow paths (Section 8.1.3).

8.1.1 Assume–guarantee makes it feasible to prove that synchronisation primitives make CVDNI proofs compositional

Programming language developers can prove the conditions for CVDNI's compositionality as invariant from the semantics of synchronisation primitives. In doing so, they can provide a way for program developers to prove noninterference for mixed-sensitivity concurrent programs: For users of the language, this reduces to security type checking each thread.

To demonstrate this principle, Chapter 4 added mutex locking primitives to a `While` language, gave them a semantics that manipulates the assumptions and guarantees, and proved the needed assume–guarantee compatibility condition as an invariant of the updated semantics. It then updated the existing per-thread security-type and guarantee-compliance checks (from prior work on CVDNI [67, 65]) to be sound for the new semantics.

To show the practicality of this approach for real-world programs, Chapter 6 used these checks to prove noninterference for a `While` model for software to handle inputs to the Cross Domain Desktop Compositor of Beaumont et al. [13].

8.1.2 Decomposition principles make it more feasible to prove that compilers satisfy CVDNI-preserving refinement

Compiler developers wishing to prove CVDNI-preserving refinement can use a decomposition principle that separates concerns about timing and termination, from those of semantic preservation. In doing so, they will find it easier to prove that their compiler preserves noninterference for mixed-sensitivity concurrent programs that it compiles.

To demonstrate this, Chapter 5 used a decomposition principle (first explained in Sison and Murray [85], a conference paper published about the work of that chapter) to prove that a compiler preserves CVDNI from `While` to `RISC`, a RISC-style assembly language with mutex locking primitives mirroring the `While` locking semantics.

To show that this compiler preserves noninterference for mixed-sensitivity concurrent programs, Chapter 6 obtained automatically the preservation of noninterference down to the `RISC` model for the CDDC input-handling model by this compiler.

8.1.3 Security type systems can prove CVDNI automatically for programs with secret-dependent control flow

Finally, programming language developers can use the CVDNI properties' timing sensitivity to develop security type systems that prove source-level CVDNI for programs with secret-dependent control flow. In particular, the proof of CVDNI establishes that any subsequent shared memory effects do not have any timing leaks that are convertible to storage leaks, assuming a scheduler making decisions according to the same notion of time.

To provide an example of this, Chapter 7 presents a timing-sensitive security typing rule for `While` that admits **if**-conditional branching on secrets, adds this rule to the security type system of Chapter 4, and proves that it remains sound for proving CVDNI. As mentioned earlier, this is the first demonstration of syntax-directed reasoning about secret-dependent control flow for mixed-sensitivity concurrent programs.

8.2 Why it matters: Scaling up with scarce resources

The vision of this thesis is that software developers can prove noninterference for mixed-sensitivity concurrent programs by running per-thread (dependent) security type checks,

knowing that, once proved, it will be preserved down to assembly. To fulfil this vision and provide these benefits, developers of programming languages and their compilers will need to replicate the achievements just described by this dissertation.

This matters because mixed-sensitivity reuse and concurrency are fundamental ways of scaling up the scope of software using scarce resources: Concurrency emerges where resources are shared between worker threads, and mixed-sensitivity reuse shares them in the service of different customers. Mixed-sensitivity concurrent programs are those where the sharing concerns the dedication of workers to customers—that is, where a thread may service multiple customers, and a customer may be serviced by several threads.

This section will paint a picture of the kinds of real-world programs that are mixed-sensitivity concurrent, before moving on to characterise how developers of programming languages and compilers can expect to be guided by the work of this thesis.

8.2.1 For multithreaded system software that implements sharing

Mixed-sensitivity concurrent programs will be trusted wherever worker threads, customers, and resources are not statically dedicated—that is, wherever multiple threads may service a single customer, and a single thread may service multiple customers.

This is natural to any *system software* that provides a platform or interface to multiple execution instances, wherever it happens to be multithreaded:

- In an *operating system*, this emerges when a thread, device, or component provides a certain functionality to multiple security domains, instead of being duplicated for each domain. For example, the CDDC input handler of Chapter 6 was a componentised operating system running on top of seL4, an operating-system microkernel. Its three threads together implemented the sharing of a single set of input devices between two security domains of differing sensitivity.

- A *hypervisor* is a kind of operating system designed to host execution instances, each of which are typically dedicated only to a single cloud service user. However, mixed-sensitivity concurrent reasoning would be needed if (like the CDDC input handler) the hypervisor itself is implemented using multiple threads, and if any one thread is responsible for implementing the sharing of some scarce resource between execution instances belonging to two or more cloud service users.

As software that hosts multiple execution instances dealing with information of differing sensitivity, an *Internet browser* may also feature mixed-sensitivity concurrency—for example, if there is any thread of execution in the browser's implementation that is meant to service both "incognito" and "normal" browsing tabs during its lifetime.

8.2.2 Support by programming languages and their compilers

The ability to replicate the results of this thesis for real programming languages and target platforms ultimately depends on developers of more fully featured programming languages

and compilers. If they can implement these principles, their users can enjoy the benefits that are currently possible for the `While` and `RISC` language—with the added benefit that it will actually apply to real software running on real hardware.

Concurrency demands coordination; it is common knowledge that this is typically provided by synchronisation primitives. Ultimately, those primitives must be implemented in some physical manner involving asynchronous communication on shared state. This thesis guides any programming language developer seeking to provide such synchronisation primitives, on (1) how to provide their users with a way to make explicit the relationships between the primitives' constructs (here, mutex locks) and the resources they protect, and (2) how to prove they meet the coordination requirements (formalised as assumptions and guarantees) needed to make noninterference proofs compositional.

Furthermore, for verified compiler developers this thesis shows that decomposition principles make it easier to prove security preservation requirements (cube-shaped, security-preserving refinements). Compiler verification is typically a square-shaped refinement argument. As demonstrated in this thesis, the decomposition principle for CVDNI allows a separation of concerns between security, and between normal refinement of program semantics. This will allow compiler verification to proceed by existing patterns, rather than forcing compiler verifiers to adopt an entirely new paradigm of cube-shaped refinements to prove noninterference. More generally, this thesis guides compiler developers to look for an appropriate decomposition principle when faced with preserving such hyperproperties. We note that this has already been borne out in the experiences of Barthe et al. [12] on the CompCert compiler, concurrent to this thesis.

8.3 What might come next?

As just argued, the usefulness of this work will be in what it shows possible, using the principles that it demonstrates to developers of real-world programming languages, and to developers of verified compilers for those languages. For the time being, however, these two sets of developers are likely to be (respectively) program- and compiler-verification researchers. Here I suggest some topics for exploration by research in the immediate future, raised by the work just completed for this dissertation.

8.3.1 Languages with indirect-addressing commands

The `While` language does not support pointers and arrays, and the `RISC` language does not support indirect addressing of memory. However, the formalisation of CVDNI for the Covern logic [68] (unlike that for this thesis) does allow the read–write footprint of a command to be expressed in a way that depends on values in memory. We expect future formulations and instantiations of CVDNI for other languages may take such an approach, to allow the indirect addressing needed to dereference pointers and array entries.

8.3.2 More compile-time optimisations on shared memory

The definition of CVDNI-preserving refinement (recalled in Chapter 3 and used by this thesis) imposes a *memory preservation* restriction: It requires the contents of all shared memory locations to be identical for any two configurations paired by the refinement, regardless of their classification level or any assumptions on access by other threads. This is not unusual for compiler verification for programs expected to be in a concurrent context—for example, the CompCert C compiler [48] treats operations on `volatile`-declared memory as observable events that cannot be optimised away.

This memory preservation restriction does not restrict all optimisations on shared memory—for example, it allows the wr-compiler to avoid redundant reads from memory. As explained in Chapter 5, the wr-compiler demonstrates that a compiler can take advantage of the assume–guarantee reasoning to show that tracking of register contents is stable enough to enable it to implement this optimisation.

On the other hand, this restriction effectively prevents the wr-compiler from making any optimisations that would reorder any writes that change the shared memory contents. Thus, supporting a wider range of compile-time optimisations of shared memory operations will require a relaxation of the definition of CVDNI-preserving refinement. Future work will need to explore the best way to generalise any requirements emerging from that relaxation, to make them usable for the most common compile-time optimisations. Doing so will make the kinds of optimisations normally supported only for local variables (e.g. loop hoisting, dead store elimination) also available for the first time for shared memory.

8.3.3 Compiler preservation of general assume–guarantee reasoning

CVDNI-preserving refinement and the wr-compiler support the preservation of assume–guarantee modes on read- or write-access, but not the general assume–guarantee conditions permitted by the Covern logic of Murray et al. [68] (which followed and extended the work of Chapter 4). Adding such support will require a change to the definition of CVDNI-preserving refinement. A wr-compiler proved to preserve such conditions would be able to preserve CVDNI for `While` programs proved secure using the Covern logic—in practice, this allows shared data invariants to be relied on to hold whenever locks are acquired, instead of needing to be checked at runtime.

8.3.4 Further support for secret-dependent control flow

This thesis has left to future work any attempts to verify a compiler to preserve CVDNI for programs with secret-dependent control flow. Initial work could aim to have a compiler preserve CVDNI for programs admitted by the security typing rule provided in Chapter 7 of this dissertation. Ideally, the combined contributions could be used to demonstrate proof and preservation of CVDNI for a real-world use of secret-dependent control flow.

8.4 Closing remarks

This thesis lays the foundations so that future programming languages and compilers for programs with mixed-sensitivity reuse and concurrency will be able to treat information-flow security as a first-class concern, for those software developers who need it. As trusted solutions to fundamental problems of scale in system software, it is high time that their developers have the tools and techniques to make firm guarantees about their information flow, knowing that they will be preserved under compilation. Therefore I argue: This is a challenge worth meeting.

Bibliography

[1] Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hritcu, Marco Patrignani, and Jérémy Thibault. Journey beyond full abstraction: Exploring robust property preservation for secure compilation. In *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*, pages 256–271. IEEE, 2019. URL `https://doi.org/10.1109/CSF.2019.00025`.

[2] Johan Agat. Transforming out timing leaks. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '00, pages 40–53, New York, NY, USA, 2000. ACM. ISBN 1-58113-125-9. doi: 10.1145/325694.325702. URL `http://doi.acm.org/10.1145/325694.325702`.

[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1807–1823, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4946-8. URL `http://doi.acm.org/10.1145/3133956.3134078`.

[4] Mário S. Alvim, Miguel E. Andrés, Konstantinos Chatzikokolakis, and Catuscia Palamidessi. On the relation between differential privacy and quantitative information flow. In Luca Aceto, Monika Henzinger, and Jirí Sgall, editors, *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part II*, volume 6756 of *Lecture Notes in Computer Science*, pages 60–76. Springer, 2011. doi: 10.1007/978-3-642-22012-8_4. URL `https://doi.org/10.1007/978-3-642-22012-8_4`.

[5] Aslan Askarov, Stephen Chong, and Heiko Mantel. Hybrid monitors for concurrent noninterference. In Cédric Fournet, Michael W. Hicks, and Luca Viganò, editors, *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*, pages 137–151. IEEE Computer Society, 2015. doi: 10.1109/CSF.2015.17. URL `https://doi.org/10.1109/CSF.2015.17`.

[6] Gilles Barthe, Amitabh Basu, and Tamara Rezk. Security types preserving compilation: (extended abstract). In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation, 5th International Confer-*

ence, VMCAI 2004, Venice, Italy, January 11-13, 2004, Proceedings, volume 2937 of *Lecture Notes in Computer Science*, pages 2–15. Springer, 2004. doi: 10.1007/978-3-540-24622-0_2. URL https://doi.org/10.1007/978-3-540-24622-0_2.

[7] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*, pages 100–114. IEEE Computer Society, 2004. doi: 10.1109/CSFW.2004.17. URL https://doi.ieeecomputersociety.org/10.1109/CSFW.2004.1310735.

[8] Gilles Barthe, Tamara Rezk, and Amitabh Basu. Security types preserving compilation. *Comput. Lang. Syst. Struct.*, 33(2):35–59, 2007. doi: 10.1016/j.cl.2005.05.002. URL https://doi.org/10.1016/j.cl.2005.05.002.

[9] Gilles Barthe, Tamara Rezk, Alejandro Russo, and Andrei Sabelfeld. Security of multithreaded programs by compilation. In Joachim Biskup and Javier López, editors, *Computer Security - ESORICS 2007, 12th European Symposium On Research In Computer Security, Dresden, Germany, September 24-26, 2007, Proceedings*, volume 4734 of *Lecture Notes in Computer Science*, pages 2–18. Springer, 2007. doi: 10.1007/978-3-540-74835-9_2. URL https://doi.org/10.1007/978-3-540-74835-9_2.

[10] 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.

[11] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. Secure compilation of side-channel countermeasures: The case of cryptographic "constant-time". In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*, pages 328–343. IEEE Computer Society, 2018. doi: 10.1109/CSF.2018.00031. URL https://doi.org/10.1109/CSF.2018.00031.

[12] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. Formal verification of a constant-time preserving C compiler. *Proc. ACM Program. Lang.*, 4(POPL):7:1–7:30, 2020. doi: 10.1145/3371075. URL https://doi.org/10.1145/3371075.

[13] 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 Stephen Schwab, William K. Robertson, and Davide Balzarotti, editors, *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*, pages 533–545. ACM, 2016. URL http://dl.acm.org/citation.cfm?id=2991087.

[14] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium*

on Principles of Programming Languages, POPL '04, page 14–25, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 158113729X. doi: 10.1145/964001.964003. URL https://doi.org/10.1145/964001.964003.

[15] L. Beringer and M. Hofmann. Secure information flow and program logics. In *20th IEEE Computer Security Foundations Symposium (CSF'07)*, pages 233–248, July 2007. doi: 10.1109/CSF.2007.30.

[16] John Boyland. Checking interference with fractional permissions. In Radhia Cousot, editor, *Static Analysis*, pages 55–72, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-44898-3.

[17] Stephen Brookes. A semantics for concurrent separation logic. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR 2004 - Concurrency Theory*, pages 16–34, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-28644-8.

[18] Ana Cavalcanti and David A. Naumann. Forward simulation for data refinement of classes. In Lars-Henrik Eriksson and Peter Alexander Lindsay, editors, *FME 2002:Formal Methods—Getting IT Right*, pages 471–490, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN 978-3-540-45614-8.

[19] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 232–245, New York, NY, USA, 1993. ACM. ISBN 0-89791-560-7. URL http://doi.acm.org/10.1145/158511.158639.

[20] 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.

[21] David Cock, Qian Ge, Toby Murray, and Gernot Heiser. The last mile: An empirical study of timing channels on sel4. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 570–581. ACM, 2014. URL https://doi.org/10.1145/2660267.2660294.

[22] Ellis Cohen. Information transmission in computational systems. In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, SOSP '77, pages 133–139, New York, NY, USA, 1977. ACM. doi: 10.1145/800214.806556. URL http://doi.acm.org/10.1145/800214.806556.

[23] David Costanzo and Zhong Shao. A separation logic for enforcing declarative information flow control policies. In Martín Abadi and Steve Kremer, editors, *Principles of Security and Trust*, pages 179–198, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-642-54792-8.

[24] Willem P. de Roever and Kai Engelhardt. *Data Refinement: Model-oriented Proof Theories and their Comparison*, volume 46 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.

[25] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, July 1977. ISSN 0001-0782. doi: 10.1145/359636.359712. URL `http://doi.acm.org/10.1145/359636.359712`.

[26] Florian Dewald, Heiko Mantel, and Alexandra Weber. AVR processors as a platform for language-based security. In *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part I*, pages 427–445, 2017. URL `https://doi.org/10.1007/978-3-319-66402-6_25`.

[27] Gidon Ernst and Toby Murray. SecCSL: Security concurrent separation logic. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*, volume 11562 of *Lecture Notes in Computer Science*, pages 208–230. Springer, 2019. URL `https://doi.org/10.1007/978-3-030-25543-5_13`.

[28] Robert W. Floyd. Nondeterministic algorithms. *J. ACM*, 14(4):636–644, October 1967. ISSN 0004-5411. URL `http://doi.acm.org/10.1145/321420.321422`.

[29] Robert W. Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19:19–32, 1967.

[30] R. Focardi, R. Gorrieri, and V. Panini. The security checker: a semantics-based tool for the verification of security properties. In *Proceedings The Eighth IEEE Computer Security Foundations Workshop*, pages 60–69, June 1995. doi: 10.1109/CSFW.1995.518553.

[31] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8:1–27, April 2018. ISSN 2190-8508. doi: https://doi.org/10.1007/s13389-016-0141-6.

[32] 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.

[33] Sylvia Grewe, Heiko Mantel, and Daniel Schoepe. A formalization of assumptions and guarantees for compositional noninterference. *Archive of Formal Proofs*, April 2014. ISSN 2150-914x. `http://isa-afp.org/entries/SIFUM_Type_Systems.html`, Formal proof development.

[34] J. Halpern and K. O'Neill. Secrecy in multiagent systems. In *Proceedings 15th IEEE Computer Security Foundations Workshop CSFW-15*, page 32, Los Alamitos, CA, USA, jun 2002. IEEE Computer Society. doi: 10.1109/CSFW.2002.1021805. URL https://doi.ieeecomputersociety.org/10.1109/CSFW.2002.1021805.

[35] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969. ISSN 0001-0782. doi: 10.1145/363235.363259. URL http://doi.acm.org/10.1145/363235.363259.

[36] Sebastian Hunt and David Sands. On flow-sensitive security types. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 79–90, New York, NY, USA, 2006. ACM. ISBN 1-59593-027-2. URL http://doi.acm.org/10.1145/1111037.1111045.

[37] J. Jacob. Security specifications. In *Proceedings. 1988 IEEE Symposium on Security and Privacy*, pages 14–23, 1988.

[38] Dean Jacobs and David Gries. General correctness: A unification of partial and total correctness. *Acta Informatica*, 22(1):67–83, Apr 1985. ISSN 1432-0525. doi: 10.1007/BF00290146. URL https://doi.org/10.1007/BF00290146.

[39] Anita K. Jones and Richard J. Lipton. The enforcement of security policies for computation. In *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, SOSP '75, pages 197–206, New York, NY, USA, 1975. ACM. doi: 10.1145/800213.806538. URL http://doi.acm.org/10.1145/800213.806538.

[40] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, October 1983. ISSN 0164-0925. doi: 10.1145/69575.69577. URL http://doi.acm.org/10.1145/69575.69577.

[41] Cliff B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. D.Phil. thesis, University of Oxford, June 1981.

[42] Aleksandr Karbyshev, Kasper Svendsen, Aslan Askarov, and Lars Birkedal. Compositional non-interference for concurrent programs via separation and framing. In Lujo Bauer and Ralf Küsters, editors, *Principles of Security and Trust - 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10804 of *Lecture Notes in Computer Science*, pages 53–78. Springer, 2018. URL https://doi.org/10.1007/978-3-319-89722-6_3.

[43] Gerwin Klein and Tobias Nipkow. A machine-checked model for a java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.*, 28(4):619–695, July 2006. ISSN 0164-0925. doi: 10.1145/1146809.1146811. URL http://doi.acm.org/10.1145/1146809.1146811.

[44] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, Feb 2014.

[45] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[46] Boris Köpf and Heiko Mantel. Transformational typing and unification for automatically correcting insecure programs. *Int. J. Inf. Secur.*, 6(2-3):107–131, March 2007. ISSN 1615-5262. URL `http://dx.doi.org/10.1007/s10207-007-0016-z`.

[47] Ramana Kumar, Magnus Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In Peter Sewell, editor, *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 179–191, San Diego, January 2014. ACM Press. doi: 10.1145/2535838.2535841.

[48] 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`.

[49] Andreas Lochbihler. Verifying a compiler for java threads. In Andrew D. Gordon, editor, *Programming Languages and Systems*, pages 427–447, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-11957-6.

[50] 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`.

[51] 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.

[52] Nancy Lynch and Frits Vaandrager. Forward and backward simulations. *Inf. Comput.*, 128(1):1–25, July 1996. ISSN 0890-5401. doi: 10.1006/inco.1996.0060. URL `https://doi.org/10.1006/inco.1996.0060`.

[53] Heiko Mantel and Henning Sudbrock. Flexible scheduler-independent security. In Dimitris Gritzalis, Bart Preneel, and Marianthi Theoharidou, editors, *Computer Security – ESORICS 2010*, pages 116–133, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-15497-3.

[54] 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.

[55] Heiko Mantel, Markus Müller-Olm, Matthias Perner, and Alexander Wenner. Using dynamic pushdown networks to automate a modular information-flow analysis. In *25th International Symposium on Logic Based Program Synthesis and Transformation (LOPSTR)*, 2015.

[56] Stephen P. Masticola and Barbara G. Ryder. Non-concurrency analysis. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, pages 129–138, New York, NY, USA, 1993. ACM. ISBN 0-89791-589-5. URL `http://doi.acm.org/10.1145/155332.155346`.

[57] Daniel Matichuk, Toby Murray, and Makarius Wenzel. Eisbach: A proof method language for isabelle. *Journal of Automated Reasoning*, 56(3):261–282, Mar 2016. ISSN 1573-0670. URL `https://doi.org/10.1007/s10817-015-9360-2`.

[58] John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. *Mathematical aspects of computer science*, 1, 1967.

[59] Robin Milner and Richard Weyhrauch. Proving compiler correctness in a mechanized logic. *Machine intelligence*, 7(3):51–70, 1972.

[60] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997. ISBN 0262631814.

[61] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *Proceedings of the 8th International Conference on Information Security and Cryptology*, ICISC'05, pages 156–168, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-33354-1, 978-3-540-33354-8. doi: 10.1007/11734727_14. URL `http://dx.doi.org/10.1007/11734727_14`.

[62] Carroll Morgan. The shadow knows: Refinement of ignorance in sequential programs. In *Proceedings of the 8th International Conference on Mathematics of Program Construction*, MPC'06, pages 359–378, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-35631-2, 978-3-540-35631-8. doi: 10.1007/11783596_21. URL `http://dx.doi.org/10.1007/11783596_21`.

[63] Markus Müller-Olm. *Modular Compiler Verification - A Refinement-Algebraic Approach Advocating Stepwise Abstraction*, volume 1283 of *Lecture Notes in Computer Science*. Springer, 1997.

[64] Toby Murray. On high-assurance information-flow-secure programming languages. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 43–48, Prague, Czech Republic, July 2015.

[65] Toby Murray, Robert Sison, Edward Pierzchalski, and Christine Rizkallah. A dependent security type system for concurrent imperative programs. *Archive of Formal*

Proofs, June 2016. ISSN 2150-914x. http://isa-afp.org/entries/Dependent_SIFUM_Type_Systems.html, Formal proof development.

[66] 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.

[67] Toby Murray, Robert Sison, Edward Pierzchalski, and Christine Rizkallah. Compositional verification and refinement of concurrent value-dependent noninterference. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*, pages 417–431. IEEE Computer Society, 2016. doi: 10.1109/CSF.2016.36. URL https://doi.org/10.1109/CSF.2016.36.

[68] Toby Murray, Robert Sison, and Kai Engelhardt. COVERN: A logic for compositional verification of information flow control. In *European Symposium on Security and Privacy*, pages 16–30, London, United Kingdom, April 2018. IEEE.

[69] Leonor Prensa Nieto. *Verification of parallel programs with the Owicki-Gries and Rely-Guarantee methods in Isabelle, HOL*. PhD thesis, Technical University Munich, Germany, 2002.

[70] Leonor Prensa Nieto and Javier Esparza. Verifying Single and Multi-mutator Garbage Collectors with Owicki-Gries in Isabelle/HOL. In Mogens Nielsen and Branislav Rovan, editors, *Mathematical Foundations of Computer Science 2000, 25th International Symposium, MFCS 2000, Bratislava, Slovakia, August 28 - September 1, 2000, Proceedings*, volume 1893 of *Lecture Notes in Computer Science*, pages 619–628. Springer, 2000. doi: 10.1007/3-540-44612-5_57. URL https://doi.org/10.1007/3-540-44612-5_57.

[71] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283. 2002. ISBN 978-3-540-43376-7. doi: 10.1007/3-540-45949-9.

[72] Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In Laurent Fribourg, editor, *Computer Science Logic*, pages 1–19, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN 978-3-540-44802-0.

[73] Peter W. O'Hearn. Resources, concurrency and local reasoning. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR 2004 - Concurrency Theory*, pages 49–67, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-28644-8.

[74] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs i. *Acta Informatica*, 6(4):319–340, Dec 1976. ISSN 1432-0525. doi: 10.1007/BF00268134. URL https://doi.org/10.1007/BF00268134.

[75] 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.

[76] Marco Patrignani, Amal Ahmed, and Dave Clarke. Formal approaches to secure compilation: A survey of fully abstract compilation and related work. *ACM Comput. Surv.*, 51(6):125:1–125:36, February 2019. URL `http://doi.acm.org/10.1145/3280984`.

[77] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, page 55–74, USA, 2002. IEEE Computer Society. ISBN 0769514839.

[78] Alejandro Russo, John Hughes, David Naumann, and Andrei Sabelfeld. Closing internal timing channels by transformation. In Mitsu Okada and Ichiro Satoh, editors, *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues*, pages 120–135, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-77505-8.

[79] Andrei Sabelfeld and Andrew C. Myers. A model for delimited information release. In Kokichi Futatsugi, Fumio Mizoguchi, and Naoki Yonezaki, editors, *Software Security - Theories and Systems*, pages 174–191, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-37621-7.

[80] 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`.

[81] Daniel Schoepe. *Flexible Information-Flow Control*. PhD thesis, Chalmers University of Technology and Göteborg University, 2018.

[82] Daniel Schoepe, Toby Murray, and Andrei Sabelfeld. VERONICA: Expressive and precise concurrent information flow security. In *33rd IEEE Computer Security Foundations Symposium, CSF 2020, Boston, MA, USA, June 22-26, 2020*, pages 79–94. IEEE, 2020. doi: 10.1109/CSF49147.2020.00014. URL `https://doi.org/10.1109/CSF49147.2020.00014`.

[83] Robert Sison. Per-thread compositional compilation for confidentiality-preserving concurrent programs. In *2nd Workshop on Principles of Secure Compilation*, Los Angeles, January 2018. Cătălin Hriţcu.

[84] Robert Sison. COVERN-RG release of Isabelle/HOL theories for Robert Sison's PhD thesis, October 2020. URL `http://handle.unsw.edu.au/1959.4/unsworks_72923`.

[85] Robert Sison and Toby Murray. Verifying that a compiler preserves concurrent value-dependent information-flow security. In *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141, pages 27:1–27:19, Portland, USA, September 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[86] Konrad Slind and Michael Norrish. A brief overview of HOL4. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer, 2008. URL `https://doi.org/10.1007/978-3-540-71067-7_6`.

[87] Geoffrey Smith. Recent developments in quantitative information flow (invited tutorial). In *Proceedings of the 2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, LICS '15, pages 23–31, Washington, DC, USA, 2015. IEEE Computer Society. ISBN 978-1-4799-8875-4. doi: 10.1109/LICS.2015.13. URL `http://dx.doi.org/10.1109/LICS.2015.13`.

[88] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 355–364, New York, NY, USA, 1998. ACM. ISBN 0-89791-979-3. doi: 10.1145/268946.268975. URL `http://doi.acm.org/10.1145/268946.268975`.

[89] Gregor Snelting, Dennis Giffhorn, Jürgen Graf, Christian Hammer, Martin Hecker, Martin Mohr, and Daniel Wasserrab. Checking probabilistic noninterference using joana. *it - Information Technology*, 56:280–287, November 2014. doi: 10.1515/itit-2014-1051.

[90] Deian Stefan, Alejandro Russo, Pablo Buiras, Amit Levy, John C. Mitchell, and David Maziéres. Addressing covert termination and timing channels in concurrent information flow systems. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 201–214, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1054-3. doi: 10.1145/2364527.2364557. URL `http://doi.acm.org/10.1145/2364527.2364557`.

[91] Deian Stefan, Pablo Buiras, Edward Z. Yang, Amit Levy, David Terei, Alejandro Russo, and David Mazières. Eliminating cache-based timing attacks with instruction-based scheduling. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *Computer Security – ESORICS 2013*, pages 718–735, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-40203-6.

[92] Nikhil Swamy, Juan Chen, and Ravi Chugh. Enforcing stateful authorization and information flow policies in fine. In Andrew D. Gordon, editor, *Programming Lan-*

guages and Systems, pages 529–549, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-11957-6.

[93] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 266–278. ACM, 2011. doi: 10.1145/2034773.2034811. URL `https://doi.org/10.1145/2034773.2034811`.

[94] The Coq Development Team. The coq proof assistant, version 8.11.0, January 2020. URL `https://doi.org/10.5281/zenodo.3744225`.

[95] Filippo Del Tedesco, David Sands, and Alejandro Russo. Fault-resilient noninterference. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*, pages 401–416. IEEE Computer Society, 2016. doi: 10.1109/CSF.2016.35. URL `https://doi.org/10.1109/CSF.2016.35`.

[96] Tachio Terauchi and Alex Aiken. Secure information flow as a safety problem. In Chris Hankin and Igor Siveroni, editors, *Static Analysis*, pages 352–367, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-31971-9.

[97] Stephen Tse and Steve Zdancewic. Run-time principals in information-flow type systems. In *2004 IEEE Symposium on Security and Privacy (S&P 2004), 9-12 May 2004, Berkeley, CA, USA*, pages 179–193. IEEE Computer Society, 2004. URL `https://doi.org/10.1109/SECPRI.2004.1301323`.

[98] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. In *Proceedings. 11th IEEE Computer Security Foundations Workshop (Cat. No.98TB100238)*, pages 34–43, June 1998. doi: 10.1109/CSFW.1998.683153.

[99] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2-3):167–187, January 1996. ISSN 0926-227X. URL `http://dl.acm.org/citation.cfm?id=353629.353648`.

[100] Dennis M. Volpano and Geoffrey Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(1), 1999. URL `http://content.iospress.com/articles/journal-of-computer-security/jcs129`.

[101] Steve Zdancewic and Andrew C. Myers. Observational determinism for concurrent program security. In *16th IEEE Computer Security Foundations Workshop (CSFW-16 2003), 30 June - 2 July 2003, Pacific Grove, CA, USA*, page 29. IEEE Computer Society, 2003. URL `https://doi.org/10.1109/CSFW.2003.1212703`.

[102] Lantian Zheng and Andrew C. Myers. Dynamic security labels and noninterference (extended abstract). In Theodosis Dimitrakos and Fabio Martinelli, editors, *Formal Aspects in Security and Trust: Second IFIP TC1 WG1.7 Workshop on Formal Aspects in Security and Trust (FAST), an event of the 18th IFIP World Computer Congress, August 22-27, 2004, Toulouse, France*, volume 173 of *IFIP*, pages 27–40. Springer, 2004. doi: 10.1007/0-387-24098-5_3. URL `https://doi.org/10.1007/0-387-24098-5_3`.

[103] Lantian Zheng and Andrew C. Myers. Dynamic security labels and static information flow control. *Int. J. Inf. Sec.*, 6(2-3):67–84, 2007. doi: 10.1007/s10207-007-0019-9. URL `https://doi.org/10.1007/s10207-007-0019-9`.

Appendix A

Extra details about the Covern wr-compiler

A.1 Label allocation and sequential composability

The wr-compiler's fixing of the label type $Lab \triangleq nat$ (noted in Section 5.2) allows it to ensure freshness merely by using the highest natural number reached so far on a "next label" counter (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 (nl' in the example).

Relative to this scheme, I 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, I 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 I 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.

A.2 Register allocation scheme model

Like Tedesco et al. [95] I 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. I 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 I model the (user-supplied) register allocation scheme with two functions *reg_alloc* and *reg_alloc_cached* on the *register record* Φ (see Section 5.2) 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* Φ 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* Φ 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.

A.3 Informal descriptions of cases of refinement relation $\mathcal{R}_{\mathsf{wr}}$

A.3.1 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 4.1.1).

- `lock_acq`: This case relates **lock**(k) with **LockAcq** k.

- `lock_rel`: This case relates **unlock**(k) with **LockRel** k.

A.3.2 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 Section A.1) 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 A.1), 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 $pc >$ 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 **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 $\mathcal{R}_{\mathsf{wr}}$ 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 A.1, on page 131; for comparison, Figure A.2 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 $\mathcal{R}_{\mathsf{wr}}$ 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 the "epilogue" step forms: **Jmp** and **Nop** (see Section 5.4.2).

  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 **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 $\mathcal{R}_{\mathsf{wr}}$ 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 **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 $\mathcal{R}_{\mathsf{wr}}$ 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.

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

Figure A.1: Introduction rule for case `if_expr` of refinement relation $\mathcal{R}_{\mathsf{wr}}$.
This case pertains to the expression-evaluation part of an **if**-conditional compiled by compile-cmd (see Figure A.2). 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 A.2: 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.

Appendix B

Extra details about the IfH security typing rule

Here I provide formal definitions that were elided from the exposition of Chapter 7.

A number of definitions to follow use a function vars-modified-by :: $cmd \Rightarrow Var\ set$, that I define trivially to return the set of all variables that appear on the left-hand side of any assignment in the given `While`-language command.

Definition B.1 (Variables modified by c are not considered visible with stable set $\mathcal{S}$)**.**

$$\textsf{vars-modified-NoRW-or-H}\ c\ \mathcal{S} \triangleq \forall x \in \textsf{vars-modified-by}\ c.\ \mathcal{L}_{bexp}\ x = \{\textsf{False}_{bexp}\} \lor (x \notin \mathcal{C} \land x \in \textsf{snd}\ \mathcal{S})$$

The following definitions are for helpers given in Section 7.2. Here, I use the notation "$m_1 +\!+ m_2$" to denote the combination of two partial maps, preferring entries from m_2; modified variables are given the default unsatisfiable type predicate set $\{\textsf{False}_{bexp}\}$.

Definition B.2 (Raising to High in typing environment Γ of variables modified by c)**.**

$$\textsf{tyenv-raise-modified}\ \Gamma\ c \triangleq \Gamma +\!+ (\lambda x.\ \textsf{if}\ (x \in \textsf{dom}\ \Gamma \land (x \in \textsf{vars-modified-by}\ c))\ \textsf{then Some}\ \{\textsf{False}_{bexp}\}\ \textsf{else None})$$

Definition B.3 (Removal from predicate set P of variables modified by c)**.**

$$\textsf{preds-remove-modified}\ P\ c \triangleq \{e \mid e \in P \land \mathit{bexp\text{-}vars}\ e \subseteq (-\ \textsf{vars-modified-by}\ c)\}$$

The following definitions, which are used by the IfH typing rule (Definition 7.1), are recalled from the background work Murray et al. [67]. Note that, as typing environments are partial maps, the floor notation $\lfloor \Gamma\ x \rfloor$ denotes the security type that the typing environment Γ has recorded for the variable x, assuming one is present.

Reproduced definitions (From Murray et al. [67], adding to those in Chapter 4)**.**

$$\Gamma \vdash e : \bigcup_{x \in \textit{bexp-vars}\ e} \mathsf{if}\ (x \in \mathsf{dom}\ \Gamma)\ \mathsf{then}\ \lfloor \Gamma\ x \rfloor\ \mathsf{else}\ \mathcal{L}_{bexp}\ x$$

(Typing rule for boolean expressions.)

$$P \vdash P' \triangleq \forall mem.\ \mathsf{preds\text{-}hold}\ P\ mem \longrightarrow \mathsf{preds\text{-}hold}\ P'\ mem$$

(Predicate set entailment.)

$$t =:_P t' \triangleq t \leq:_P t' \wedge t' \leq:_P t$$

(Type equivalence between predicate sets t, t' under predicate set P.)

$$\Gamma =:_P \Gamma' \triangleq \mathsf{dom}\ \Gamma = \mathsf{dom}\ \Gamma' \wedge \forall x \in \mathsf{dom}\ \Gamma'.\ \lfloor \Gamma\ x \rfloor =:_P \lfloor \Gamma'\ x \rfloor$$

(Typing environment equivalence under predicate set P.)

The following definitions are then for helpers given in Section 7.3, and are used by the definition of the new component $\mathcal{B}^2_{\Gamma',\mathcal{S}',P'}$ (Definition 7.3) of the bisimulation construction.

Definition B.4 (All variables modified by c have type High in typing environment Γ)**.**

$$\mathsf{vars\text{-}modified\text{-}tyenv\text{-}H}\ c\ \Gamma \triangleq$$
$$\forall x \in \mathsf{vars\text{-}modified\text{-}by}\ c.\ x \in \mathsf{dom}\ \Gamma \longrightarrow \Gamma\ x = \mathsf{Some}\ \{\mathsf{False}_{bexp}\}$$

Definition B.5 (No predicates in set P mention any variables modified by c)**.**

$$\mathsf{vars\text{-}modified\text{-}no\text{-}preds}\ c\ P \triangleq$$
$$\forall x \in \mathsf{vars\text{-}determining\text{-}preds}\ P.\ x \notin \mathsf{vars\text{-}modified\text{-}by}\ c$$