Scalable Proof Techniques for Formally Secure Compilation of Compartmentalized Programs

Dissertation
zur Erlangung des Doktorgrades (Dr. rer. nat.)
an der Fakultät für Informatik der Ruhr-Universität Bochum



vorgelegt von

Jérémy Thibault

Erstgutachter: Cătălin Hriţcu
Zweitgutachter: Gilles Barthe
Drittgutachter: Frank Piessens

Datum der mündlichen Prüfung: 18.09.2025

Abstract

Scalable Proof Techniques for Formally Secure Compilation of Compartmentalized Programs

Jérémy Thibault

Current software systems are distressingly insecure, and despite constant attempts at patching systems and implementing protection mechanisms, many programs, especially those written in fundamentally unsafe languages such as C, admit vulnerabilities that are sometimes exploited to devastating effect.

A typical mitigation is compartmentalization: software system designers split their applications in several compartments that can only interact via trusted interfaces, in the hope a vulnerability in one compartment doesn't jeopardize the security of the entire application. Unfortunately, compartmentalization mechanisms are still fallible, and and regularly broken, allowing an attacker to exploit vulnerabilities to attack other compartments. Indeed, these mechanisms are rarely designed with formal guarantees in mind, and even when they are, the sheer complexity of the implementation makes it difficult to guarantee security bugs do not slip through.

In this thesis, we work towards improving the situation, from the perspective of secure compilation. Secure compilation is a research area that aims at giving programmers tools to reason about the security of their programs in their source languages, putting together techniques from programming languages, compiler design, systems, and security in order to build secure compilation chains. Often, in the formal security community, designers of simple secure compilation chains try to prove that they satisfy formal guarantees; however, these proofs are often extremely lengthy and complex, even for these very simple compilation chains. Given the very high stakes for security and the challenge of scaling up formally secure compilation to realistic compilers, secure compilation proofs are the prime target for mechanization in a proof assistant, yet such mechanizations are still few and far between.

We study the formal verification of secure compilers, and in the process, develop new proof techniques that make it easier to formally verify a secure compiler. In particular, we improve upon an existing proof technique by introducing *recomposition*, a way to avoid reasoning about partial semantics, which are known to be tricky to get right and lead to difficult proofs.

Following this, we build SECOMP, a secure compiler for a compartmentalized variant of C, as an extension of the CompCert compiler. To do so, we extend all of CompCert's languages with a notion of compartments that can only interact with each other in protected ways, and we adapt the compilation passes of CompCert to this setting. We prove not only the correctness of SECOMP, but also that it is secure, in the sense that undefined behaviours are restricted to the compartment that triggers them. In the process of proving security, we introduce several proof engineering novelties that were necessary to scale up to C. SECOMP is the first formally verified secure compiler for a realistic programming language with compartments.

We conclude this thesis by presenting two interesting extensions of the proof technique we use: one that allows handling memory sharing between compartments, a common programming pattern that is not yet supported by SECOMP; and a back-translation technique that allows one to prove a much stronger criterion with little extra effort.

Kurzfassung

Skalierbare Beweistechniken für die formal sichere Kompilierung von kompartimentierten Programmen

Jérémy Thibault

Aktuelle Softwaresysteme sind erschreckend unsicher. Trotz ständiger Versuche, Systeme zu patchen und Schutzmechanismen zu implementieren, weisen viele Programme, insbesondere solche, die in grundsätzlich unsicheren Sprachen wie C geschrieben sind, Schwachstellen auf, die mitunter mit verheerender Wirkung ausgenutzt werden.

Eine typische Abschwächungsmaßnahme ist die Kompartimentierung: Softwareentwickler teilen ihre Anwendungen in mehrere Bereiche auf, die nur über vertrauenswürdige Schnittstellen interagieren können, in der Hoffnung, dass eine Schwachstelle in einem Bereich die Sicherheit der gesamten Anwendung nicht gefährdet.

Leider sind Kompartimentierungsmechanismen immer noch fehleranfällig und werden regelmäßig beschädigt, sodass Angreifer Schwachstellen ausnutzen und andere Bereiche angreifen können.

Tatsächlich werden diese Mechanismen selten mit formalen Garantien entwickelt, und selbst wenn dies der Fall ist, erschwert die Komplexität der Implementierung die Gewährleistung, dass keine Sicherheitslücken durchrutschen.

In dieser Arbeit arbeiten wir daran, die Situation aus der Perspektive der sicheren Kompilierung zu verbessern. Sichere Kompilierung ist ein Forschungsgebiet, das Programmierern Werkzeuge an die Hand gibt, um die Sicherheit ihrer Programme in ihren Quellsprachen zu beurteilen. Dazu werden Techniken aus Programmiersprachen, Compilerdesign, Systemen und Sicherheit kombiniert, um sichere Kompilierungsketten zu erstellen.

In der formalen Sicherheitscommunity versuchen Entwickler einfacher sicherer Kompilierungsketten oft nachzuweisen, dass sie formale Garantien erfüllen. Diese Beweise sind jedoch oft extrem langwierig und komplex, selbst für diese sehr einfachen Kompilierungsketten.

Angesichts der hohen Sicherheitsanforderungen und der Herausforderung, formal sichere Kompilierung auf realistische Compiler zu skalieren, sind sichere Kompilierungsbeweise das wichtigste Ziel für die Mechanisierung in einem Beweisassistenten. Solche Mechanisierungen sind jedoch noch selten.

Wir untersuchen die formale Verifikation sicherer Compiler und entwickeln dabei neue Beweistechniken, die die formale Verifikation sicherer Compiler erleichtern. Insbesondere verbessern wir eine bestehende Beweistechnik durch die Einführung von *Rekomposition*. Diese Methode vermeidet das Denken über partielle Semantik, die bekanntermaßen schwierig zu verstehen ist und zu schwierigen Beweisen führt.

Anschließend entwickeln wir SECOMP, einen sicheren Compiler für eine kompartimentierte Variante von C, als Erweiterung des CompCert-Compilers. Dazu erweitern wir alle CompCert-Sprachen um das Konzept von Kompartimenten, die nur geschützt miteinander interagieren können, und passen die Kompilierdurchläufe von CompCert an diese Einstellung an. Wir beweisen nicht nur die Korrektheit von SECOMP, sondern auch dessen Sicherheit, da undefiniertes Verhalten auf das auslösende Kompartiment beschränkt ist.

Im Zuge des Sicherheitsnachweises führen wir mehrere Neuerungen im Proof-Engineering ein, die für die Skalierung auf C notwendig waren. SECOMP ist der erste formal verifizierte sichere Compiler für eine realistische Programmiersprache mit Kompartimenten.

Wir schließen diese Arbeit mit der Präsentation zweier interessanter Erweiterungen der von uns verwendeten Beweistechnik ab: Eine ermöglicht die gemeinsame Nutzung von Speicher zwischen Kompartimenten, ein gängiges Programmiermuster, das von SECOMP noch nicht unterstützt wird, und eine Rückübersetzungstechnik, mit der sich mit geringem Mehraufwand ein deutlich stärkeres Kriterium beweisen lässt.

Contents

Contents iv

1 Introduction 2

A First Foray into Secure Compilation Proofs 13

2 A Look at Secure Compilation Criteria and Proofs 14
2.1 Journey Beyond Full Abstraction: A Summary . . . 15
2.1.1 Robust Trace Property Preservation . . . 17
2.1.2 Robust Safety Property Preservation (RSP) . . . 18
2.1.3 Hyperproperties and Relational Hyperproperties . . . 19
2.2 An Example of a Secure Compilation Chain . . . 20
2.2.1 Overview . . . 20
2.2.2 A Security Proof for this Compiler with a Trace-Based Back-Translation . . . 23
2.2.3 Discussion on the Proof . . . 25
2.3 Fully Abstraction Compilation and Robust Property Preservation . . . 27

3 Secure Compilation, Compartmentalization and Undefined Behaviour 33
3.1 Background: Robustly Safe Compartmentalizing Compilation . . . 33
3.2 Background: A Proof Technique for $\mathsf{RSC}^{\mathsf{DC}}_{\mathsf{MD}}$ . . . 37

4 Recomposition 43
4.1 Proving Recomposition . . . 44
4.2 Discussion . . . 46

SECOMP: A Secure Compiler From Compartmentalized C to RISC-V Assembly 48

5 Compiling Compartments from C Down to Assembly 49
5.1 Compartment Model . . . 49
5.2 Adding Compartments to CompCert . . . 51
5.2.1 Memory Model . . . 51
5.2.2 Call and Return Events . . . 51
5.2.3 Compartments at Higher Levels . . . 52
5.2.4 Compartments at the RISC-V Level . . . 53
5.3 Proving Correctness . . . 55

6 The Secure Compilation Proof of SECOMP 58
6.1 Back-translation . . . 58
6.2 Recomposition . . . 61
6.2.1 Simulation Diagrams for Recomposition . . . 62
6.2.2 Instantiating the Diagrams . . . 65
6.3 Discussion . . . 68

Extensions 70

7 Memory Sharing 71
7.1 Memory Sharing Model . . . 72

7.2 Extending the Back-Translation Proof with Data-flow Traces . . . 73
7.2.1 Outline of data-flow back-translation proof . . . 73
7.2.2 Data-flow Back-Translation, in Practice . . . 74
7.3 Verification of SecurePtrs . . . 76
7.3.1 Verifying the Data-Flow Back-Translation . . . 76
7.3.2 Rocq Mechanization . . . 77

8 Nanopass Back-Translation **78**
8.1 Key Ideas . . . 80
8.1.1 Towards a proof of a stronger criterion . . . 80
8.1.2 Back-translating multiple traces . . . 81
8.1.3 Nanopass back-translation . . . 83
8.2 Design of the nanopass back-translation . . . 85
8.2.1 The tree languages in the back-translation . . . 85
8.2.2 Flattening and code generation . . . 88
8.3 Proving the back-translation correct using forward simulations . . . 89
8.3.1 Forward simulations . . . 89
8.3.2 Simulation relations and proofs . . . 89
8.4 Formalization in Rocq . . . 91

Discussion **93**

9 Related Work **94**
9.1 Compiler Correctness and CompCert . . . 94
9.2 Secure Compilation . . . 95
9.2.1 Full Abstraction and Proof Techniques . . . 95
9.2.2 Other Kinds of Secure Compilers . . . 96
9.3 Formalization of Secure Compilers . . . 97

10 Conclusion and Future Work **98**

Bibliography **100**

Acknowledgments

First and foremost, I would like to thank my advisor Cătălin Hriţcu. Cătălin, you took me in when I knew very little about research, and I've learnt a great deal from you. That you constantly push for improvement and that you do so with such kindness are an inspiration to us all.

I would also like to thank the reviewers of this thesis, as well as the members of the committee, for having taken the time to engage with the manuscript and for the insightful discussions.

During the many years it took me to complete this thesis, I had the chance to meet so many wonderful people, and I shall not attempt to make an exhaustive list, lest I forget someone who deserves to be on it. However, I will mention first the little group in Paris, whose passion for computer science sent me on the track where I am now; then, the people from MPI-SP here in Germany, both those who were already there — who were so welcoming when I arrived in the middle of the pandemic — and those who arrived later and carried on the tradition of making the institute a lovely and welcoming place to work; and finally, all the people whom I interacted with at conferences and other events and who, together, make computer science the best discipline to pursue a PhD in.

I take collaboration to be essential to research, and this thesis would not exist without the contributions of my many coauthors. I am thankful to have worked with you, and I'm looking forward to doing so again in the future.

I would not be here without extraordinary luck, and, most of all, the constant support and encouragement from my family, my (non-academic) friends, and my cat Momo, who's always there to remind me of what really matters in life.

And finally, thank you, Lin, for everything you brought to my life, and your presence during the hardest moments. You gave me the strength to persevere, and I couldn't have done it without you.

1 Introduction

In today's society, software is omnipresent and runs our lives. All kinds of devices — as varied as domestic appliances, phones, banking systems, and even critical infrastructure such as power plants — are built on top of extremely complex software systems.

As the size and complexity of such systems grow, it is natural to worry about the consequences security vulnerabilities can have on these systems. Despite many attempts to implement protections against attacks, people regularly show that these mechanisms are flawed and are able to exploit vulnerabilities to bypass these protections (for instance, sandboxing [9]) and compromise software systems. In addition, the widespread use of memory-unsafe languages such as C is worrying, as memory safety bugs, such as buffer overflows, use-after-free, or dereferencing a null pointer often lead to disastrous consequences in terms of security. Attackers often exploit these memory safety bugs to compromise programs [10], or to leak data. For instance, recently, Heartbleed [11] demonstrated that vulnerabilities in widely-used open-source libraries still exist and can be exploited to endanger the privacy of millions of users: by exploiting a buffer overflow, attackers could obtain the private cryptographic keys used by the server. More generally, experts at large companies estimate that around 70% of vulnerabilities are memory safety bugs [12, 13], which contributes to building a quite dire view of what software security is like nowadays.

[9]: Boozarjomehri (2024), *Rapidly Leveling up Firefox Security*

[10]: Aleph One (1996), 'Smashing the Stack for Fun and Profit'
[11]: Durumeric et al. (2014), 'The Matter of Heartbleed'

[12]: Chromium Security Team (2022), *Memory safety in Chromium*
[13]: Cimpanu (2019), *Microsoft: 70 percent of all security bugs are memory safety issues*

Thankfully, the research community has known about these issues for a long time, and a large body of work regarding the security of software systems has emerged. Researchers have proposed many ways of writing more secure programs, ranging from simple notions such as structured control flow to automated static analyses to (potentially extremely) complex and time-consuming techniques such as full-fledged program verification. In this thesis, we call all these features "high-level abstractions", as they hide the concrete ("low-level") behaviour of the machine from the programmer, and provide ways to think about the security behaviour of the program without being encumbered by details.

The programmer concerned with the security of their program will rely on some of these features to write a program that they believe to be secure.[1] For instance, consider the following program in a C-like language:

1: This belief may be unjustified: it is entirely possible that the programmer made a mistake and wrote an insecure program.

```
int main() {

  int public_key = get_public_key();

  int res = compute_something(10);

  send_message(&public_key, res);
  return 0;
}
```

The purpose of this code is to get some public key used in public key cryptography, compute some result that is completely independent of the key (using `compute_something`), and then encrypt and send that result over the network, using the key. If an attacker were to manipulate the

variable `public_key` and change the value it contains to its own public key before the call to `send_message`, then that attacker could read the secret message.

However, the programmer writing this application has good reasons to assume this cannot happen, because their high-level language provides some guarantees about scoping and control-flow: the local variable `public_key` is out-of-scope of `compute_something`, so there should be no way for `public_key` to change before the call to `send_message`, hence the program to be secure.

However, this code is not the one that is executed by the machine: indeed, this code must first be *compiled* (translated) to code that can be understood by the machine. What happens is the following: before being executed by the machine, the high-level code (the "source") is processed by a *compiler*, which transforms it into a much lower-level language (the "target"). This language is often what is typically called an "assembly language" (also called "assembly"), a language that is much closer to what's executed on the machine directly, and lacks all the high-level abstractions that were provided by the high-level language and that the programmer relied on to reason about security. This might be fine if the entire program being executed was written by the programmer[2].

2: Assuming the compiler is correct and doesn't introduce bugs or vulnerabilities

However, such cases are extremely rare in practice. Indeed, in the real world, people do not write whole programs and execute them in a vacuum, but instead the norm is to write *partial programs* that link against other partial programs (libraries, for instance) that one does not have control over. In fact, it is often considered best practice to not reinvent the wheel, and to rely on such external libraries: they not only significantly reduce the effort required to develop a new product, but are also likely to be more thoroughly tested and reliable.

In the previous example, the programmer might not have written the entirety of the `compute_something` function themselves, instead choosing to use a library for some or all of its implementation. That library might have been written in the same source language, but it might also have been written in a completely different language with different source abstractions, or it might even be written in or incorporate pieces of low-level code (such as hand-written assembly). One cannot trust such library blindly, because it could very well ignore the abstractions that the programmer relies on, and invalidate any security guarantee the programmer thought they had.

So indeed, what happens when this external library is straight-up malicious [14], or simply vulnerable to attacks, and can get compromised by an attacker, and then starts attacking the developer's program at a lower-level? This is unaccounted for by most compilers, and all hope for security is gone. And while it is true that some compilers can incorporate some security features[3], they are rarely designed with that in mind from the start, and enforcing security might interfere with their goal of (and most people's desire for) performance of the compiled program.

[14]: Claburn (2024), *Malicious SSH backdoor sneaks into xz, Linux world's data compression library*

3: Protections against stack smashing, for instance.

As an example, an implementation of `compute_something` could just include assembly code that accesses the stack pointer `SP`, offset it the right amount to reach the location of `public_key` with `addi` (by guessing an offset), and store (`sw`) a completely different value that is controlled by the adversary.

```
fact:
  ; ...
  addi R1, SP, imm(16)
```

```
  sw R1, imm(104)
  ; ...
```

Now, if this program is run, the secret will leak to the adversary!

This is made even worse by the fact that so many existing programs are written in memory-unsafe languages such as C. As we mentioned earlier, memory safety bugs are particularly bad for security, because they often lead to the compromise of the program. There is no general protection against these bugs in mainstream compilers, because they are examples of *undefined behaviours* (*UB*s): situations where the source behaviour of the program is left undefined by the standard.

Compilers are allowed to assume undefined behaviours do not happen, which they do because it lets them skip many runtime checks that would slow down the program, and perform very aggressive optimizations. On the other side of the same coin, this means that compilers don't need to provide guarantees to programs that encounter undefined behaviours: they do not have to prevent or mitigate the consequences of buffer overflows, use-after-free bugs, etc.

What this means is that one can implement a C program that exploits undefined behaviour to attack the previous program.

```
1  int compute_something(int n) {
2    int res = 1;
3    /* do something */
4    /* ... */
5
6    char* addr = (char*) &res;
7    *(addr + 48) = 104;
8
9    return res;
10 }
```

The program takes the address of a variable on the stack (line 6), offsets it by just the right amount, and manages to overwrite the variable in the parent stack frame (line 7). Even this program written entirely in C would lead to the leak of the secret message. One might object that this is an adversarial context written specifically to attack another program. However, memory safety vulnerabilities have been shown to often lead to full compromise, allowing an attacker to execute arbitrary code; hence, even a library written with no ill intention could contain a vulnerability that leads to a similar attack.

Because it would be highly unrealistic to rewrite all existing C code in safer languages, this highlights the need for new protection mechanisms that allow programmers to reason soundly about the security of their programs, even in the presence of C code possibly having undefined behaviours.

Secure compilation is the field that attempts to fill these gaps, and, as stated on the webpage of the Dagstuhl seminar on Secure Compilation, "allow sound reasoning about security properties in the source language"[4]. Said otherwise, the goal of the field of secure compilation is to build compilation chains that allow a programmer to reason about the security of their (partial) program solely in the source language, as if all the other components were also respecting the high-level abstractions of the source, despite potentially being written in a completely different language. We say "compilation chains" and not "compilers", because obtaining such results generally requires the entire toolchain, from source

4: https://www.dagstuhl.de/21481

to target, including the compiler but also the linker, the runtime, and some enforcement mechanism to be designed together to preserve the security properties.

For protecting components written in a C-like language with undefined behaviours, *compartmentalization* is the most commonly proposed mechanism, whether it takes the form of process isolation [15, 16], software fault isolation [17, 18], or modules in WebAssembly [19], to only cite a few. Compartmentalization divides programs into multiple compartments according to the principle of least privilege, so that a vulnerability in one compartment does not automatically compromise the entire system. What we would want is for the programmer to be able to write a program that looks a bit like that:

[15]: Kilpatrick (2003), 'Privman: A Library for Partitioning Applications'
[16]: Reis et al. (2009), 'Isolating web programs in modern browser architectures'
[17]: Wahbe et al. (1993), 'Efficient software-based fault isolation'
[18]: Yee et al. (2009), 'Native Client: A Sandbox for Portable, Untrusted x86 Native Code'
[19]: Rossberg et al. (2018), 'Bringing the web up to speed with WebAssembly'

```
c_main imports c_lib.compute_something;

c_main int main() {

  int public_key = get_public_key();

  int res = compute_something(10);

  send_message(&public_key, res);
  return 0;
}
```

In this program, `main` is explicitly marked as being part of compartment `c_main`, and `c_main` *imports* the function that computes from another compartment, `c_lib`. That way, the compartmentalization mechanism should guarantee that `c_lib` does not have access to `c_main`'s private memory. In such a world, if one tried to run one of the `compute_something` above, it would be stopped before being able to overwrite the public key.

Obviously, a compartmentalization mechanism is hardly worth anything if one doesn't understand the guarantees it provides. Thankfully, recent work by Abate et al. [8] formalized what one should expect from a compartmentalizing compilation chain as a secure compilation theorem. The idea behind most secure compilation theorems is the following: if one or several source (partial) programs always satisfy some security property(-ies) even when linked against an *arbitrary* source *context* (i.e., another partial program, which for our example would be the library implementing `compute_something`), then the compiled program(s) will also satisfy the same security property(-ies) when linked against an arbitrary *target* context. More explicitly, the contexts are seen as attackers that try to make the partial programs violate some security properties, and such a statement means that source and target attackers have the same attacking power against certain types of security properties.

[8]: Abate et al. (2018), 'When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise'

This kind of statement is very strong: indeed, the arbitrary target context should *a priori* be much more powerful, because it doesn't have to abide by the same restrictions as a source context. Hence, it comes as no surprise that building a compilation chain that enforces this and proving such a theorem is difficult. Typical proofs of secure compilation use a technique known as back-translation: given some low-level target context, one proves that *there exists* a high-level source context that is *as powerful as* the target context. Paper proofs of such results are often lengthy[5] and often use elaborate techniques, and they often involve many minute details that are tedious to check.

5: Such proofs routinely go over 100 pages in length

The case for formal verification Another issue is that it's not just that the proofs are difficult: protection mechanisms that promise security guarantees, including compartmentalization mechanisms, get broken all the time, and every time they do, consequences can be terrible. Hence, we assert that secure compilation chains are the perfect use case for *formal verification* in a proof assistant.

There is already a long line of research since the 1960s that focuses on verifying compilers [20, 21] to be correct, often done using proof assistants. The basic idea is the following: the programs in the source and target languages are assigned a mathematical meaning, their *semantics*, and a mathematical proof shows that for any source program, its compilation has the same semantics. The area of compiler correctness has been very fruitful, and works such as CompCert [22] and CakeML [23] have shown that it is entirely possible to build realistic optimizing compilers that are provably (in the mathematical sense, and in a proof assistant) correct.

[20]: McCarthy et al. (1967), *Correctness of a compiler for arithmetic expressions*
[21]: Morris (1973), 'Advice on structuring compilers and proving them correct'

[22]: Leroy (2009), 'A Formally Verified Compiler Back-end'

[23]: Tan et al. (2019), 'The verified CakeML compiler backend'

Hence, it seems natural to try to extend this with security proofs. Yet, even now, only a small number of secure compilation chains have been mechanized and verified in a proof assistant. The main reason for this is the difficulty of these proofs, and the fact that the proof techniques mentioned above are often difficult to mechanize properly.

Contributions and Outline

In this thesis, our ultimate goal is to design and implement SECOMP, a secure compartmentalizing compilation chain for C that allows programmers to annotate their code with compartment information and reason soundly about the security of their programs, even in the presence of foreign and potentially adversarial code, and undefined behaviours. We implement SECOMP in the Rocq proof assistant, as an extension of the already existing CompCert C compiler. However, before reaching this goal, we must start by presenting several works upon which SECOMP is built. These chapters deal with various questions related to secure compilation and the mechanization of secure compilation proofs.

In chapter 2, we start by observing that *full abstraction*, the gold-standard secure compilation criterion, is inadequate for our goal of building a formally verified secure compiler: its security guarantees aren't clear, and it is often too difficult to enforce and too difficult to prove. Following these observations, we summarize a paper by Abate et al. [6], where a class of secure compilation criteria based on the preservation of classes of trace properties is proposed. Some of these criteria are weaker but easier to enforce, while others are much stronger, but difficult to enforce; and different proof techniques apply in each case. We argue that choosing from these criteria is superior to sticking to full abstraction for the purpose of secure compilation, because they give more clarity about what security properties are preserved by the compiler and because they provide more flexibility on what to implement and prove. To illustrate our argument, we show a proof of a security theorem for a simple compiler that compiles static types to dynamic types. The proof is relatively simple, and captures many of the basic ideas that we will eventually use and improve in order to prove the security of SECOMP.

[6]: Abate et al. (2019), 'Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation' **[Co-authored]**

In 2018, Abate et al. [8] proposed a new criterion for secure compilation in the presence of undefined behaviour, based on the idea of compartmentalization, that we plan to reuse for SECOMP. The core idea of their criterion

[8]: Abate et al. (2018), 'When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise'

is the following: programs are divided into compartments; compartments are protected from each other, in such a way that undefined behaviour doesn't propagate from one compartment to another. They illustrate this criterion by implementing a secure compilation chain for a toy C-like language. Despite the simplicity of their setting, their proof of security remains complex, relying on "partial semantics" (giving semantics to partial programs) and steps called "decomposition" and "composition" to go between partial and whole-program semantics. In chapter 3, we describe the ideas behind this criterion and their mechanization.

In chapter 4, we endeavour to improve this previous work by simplifying this proof. To get rid of partial semantics, we replace the steps of decomposition and composition with a new step called "recomposition" [7]. Implementing recomposition considerably simplifies the existing proof by a few thousand lines of code and, more importantly, removes the reliance on partial semantics, which are known to be difficult to define and tricky to use.

[7]: Abate et al. (2019), *When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise* **[Co-authored]** (arXiv version with recomposition)

The innovations from these chapters are crucial for us to scale secure compilation to a realistic setting, and we can now introduce the main contribution of this thesis: SECOMP, the first formally verified secure compiler for a realistic compartmentalized language [2].

[2]: **Thibault** et al. (2024), 'SECOMP: Formally Secure Compilation of Compartmentalized C Programs' **[Co-authored]**

First (chapter 5), we present our variant of C, Compartmentalized C, and describe our extension of the CompCert C compiler to this compartmentalized setting. We discuss the design decisions that took place during the development of SECOMP, as well as how we are able to update the compiler correctness proof with minimal changes.

Then, in chapter 6, we discuss how we prove SECOMP secure. We adapt the proof technique from Abate et al. [8] and chapter 3, and succeed at showing that one can build and verify a realistic secure compiler using such a proof technique. We also describe improvements to the individual proof steps, and in particular a new structure for the recomposition proof using CompCert-style simulation diagrams. These diagrams make it easier to prove recomposition as a series of smaller lemmas, compared to the proof from chapter 4.

[8]: Abate et al. (2018), 'When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise'

Finally, in the last part of this thesis, we describe two additional improvements and extensions of this proof technique. First, in chapter 7, we show that the compiler from chapter 3 can be extended to support *memory sharing*, a feature that we intentionally left out in SECOMP. To prove the compiler secure in the presence of memory sharing, we focus on one new technique in particular: back-translation using data-flow traces. Data-flow traces record every memory and register operations as they happen, allowing to simulate low-level memory operations in the source, without having to keep extremely complex metadata [3]. Then, in chapter chapter 8, we describe another extension to the back-translation proof, that allows one to obtain much stronger criteria while keeping the bulk of the proof the same [1]. The idea is to structure the back-translation as a *nanopass compiler*, i.e., a series of small steps instead of a massive single transformation.

[3]: El-Korashy et al. (2022), 'SecurePtrs: Proving Secure Compilation with Data-Flow Back-Translation and Turn-Taking Simulation' **[Co-authored]**

[1]: **Thibault** et al. (2025), 'Nanopass Back-Translation of Call-Return Trees for Mechanized Secure Compilation Proofs' **[Co-authored]**

We conclude by discussing some related work (chapter 9) and giving some concluding remarks and outlining future work (chapter 10).

A note on authorship Most of the work described in this thesis has been done in collaboration with other people. At the beginning of each chapter, I, the author of this thesis, take a moment to describe my

individual contribution to that chapter. Papers of which I am a co-author are annotated with a **[Co-authored]** mark.

List of Publications

Conference Publications

Jérémy Thibault, Joseph Lenormand, and Catalin Hritcu. 'Nanopass Back-Translation of Call-Return Trees for Mechanized Secure Compilation Proofs'. In: *16th International Conference on Interactive Theorem Proving, ITP 2025, September 28 to October 1, 2025, Reykjavik, Iceland*. Ed. by Yannick Forster and Chantal Keller. Vol. 352. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025, 29:1–29:20. DOI: 10.4230/LIPICS.ITP.2025.29.

Jérémy Thibault, Roberto Blanco, Dongjae Lee, Sven Argo, Arthur Azevedo de Amorim, Aïna Linn Georges, Cătălin Hriţcu, and Andrew Tolmach. 'SECOMP: Formally Secure Compilation of Compartmentalized C Programs'. In: *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. CCS '24. ACM, Dec. 2024, pp. 1061–1075. DOI: 10.1145/3658644.3670288.

Akram El-Korashy, Roberto Blanco, **Jérémy Thibault**, Adrien Durier, Deepak Garg, and Cătălin Hriţcu. 'SecurePtrs: Proving Secure Compilation with Data-Flow Back-Translation and Turn-Taking Simulation'. In: *2022 IEEE 35th Computer Security Foundations Symposium (CSF)*. IEEE, Aug. 2022, pp. 64–79. DOI: 10.1109/csf54842.2022.9919680.

Carmine Abate, Roberto Blanco, Ștefan Ciobâcă, Adrien Durier, Deepak Garg, Cătălin Hriţcu, Marco Patrignani, Éric Tanter, and **Jérémy Thibault**. 'Trace-Relating Compiler Correctness and Secure Compilation'. In: *Programming Languages and Systems*. Springer International Publishing, 2020, pp. 1–28. DOI: 10.1007/978-3-030-44914-8_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: *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. IEEE, June 2019. DOI: 10.1109/csf.2019.00025.

Journal Publications

Carmine Abate, Roberto Blanco, Ştefan Ciobâcă, Adrien Durier, Deepak Garg, Catalin Hritcu, Marco Patrignani, Éric Tanter, and **Jérémy Thibault**. 'An Extended Account of Trace-relating Compiler Correctness and Secure Compilation'. In: *ACM Trans. Program. Lang. Syst.* 43.4 (2021), 14:1–14:48. DOI: 10.1145/3460860.

Informal Publications

Carmine Abate, Arthur Azevedo de Amorim, Roberto Blanco, Ana Nora Evans, Guglielmo Fachini, Catalin Hritcu, Théo Laurent, Benjamin C. Pierce, Marco Stronati, **Jérémy Thibault**, and Andrew Tolmach. *When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise*. Extended version of a paper

previously published at CCS'18. Updates the proof technique to use *recomposition*. 2019. URL: https://arxiv.org/abs/1802.00588.

Participation in Conferences and Other Noteworthy Academic Events

- **CCS 2024**, Salt Lake City, UT, USA.
  Presented [1].
- **POPL 2023**, Boston, MA, USA.
  Presented an earlier version of [1] at the PriSC workshop.
- **Dagstuhl Seminar 21481** on Secure Compilation (2021)
- **POPL 2021**, Online.
  Presented an earlier version of [2] at the PriSC workshop.
- **OPLSS 2019**.
  Oregon Programming Languages Summer School (OPLSS), Summer 2019, Eugene, OR, USA.
- **POPL 2019**, Cascais, Portugal.
  Participated in the Student Research Competition with a poster titled "A Trace-Based Proof Technique for Secure Compilation"

[1]: **Thibault** et al. (2025), 'Nanopass Back-Translation of Call-Return Trees for Mechanized Secure Compilation Proofs' **[Co-authored]**

[2]: **Thibault** et al. (2024), 'SECOMP: Formally Secure Compilation of Compartmentalized C Programs' **[Co-authored]**

Notations

Throughout this thesis, we often consider a source and a target language, with a compiler going from the source to the target.

By convention, we write elements of the source language in a **blue and bold** font and elements of the target in a red and sans-serif font. We also try to use subscripts where it makes sense to disambiguate as much as possible, but we also sometimes drop such subscripts and allow ourselves to vary slightly from these conventions to avoid sensory overload.

We typically use the notations $\rightsquigarrow$ and $\rightsquigarrow$ to denote the finite-trace producing semantics of programs, $\bowtie$ and $\bowtie$ to denote linking, $\downarrow$ to denote a compiler, and $\uparrow$ for a back-translation function. We often consider three kinds of objects: whole programs, $\mathbf{W}$ and W, partial programs $\mathbf{P}$ and P, and contexts $\mathbf{C}$ and C.

To avoid having too many colours that would distract the reader, we tend to not distinguish i, $\mathbf{i}$, and i when they are used as indexes. For instance, in the formula $\forall i.\ f(\mathbf{P}_\mathbf{i}) = \mathsf{P}'_\mathsf{i}$, $\mathbf{i}$ and i both refer to the same variable i that is universally bound.

A First Foray into Secure Compilation Proofs

A Look at Secure Compilation Criteria and Proofs

2

What makes a good secure compilation criterion? In [6], we answer that question by proposing a new taxonomy of secure compilation criteria based on the preservation of classes of security property, even when linking against adversarial context. For each of them, we give a "property-free" characterization of the criterion, that suggests a proof technique based on back-translation, a classical proof technique from the literature that involves building a source context that is "equivalent" to a given target context. We compare these criteria, both in terms of expressivity, how difficult they are to achieve, and what information can be used to prove back-translation.

[6]: Abate et al. (2019), 'Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation' **[Co-authored]**

The main two benefits of these criteria are that, first, they provide clear security guarantees: if a source partial program always satisfies a particular property of the given class, then so does its compilation, even when linked against any low-level adversarial context. In addition, one can choose which criterion from this taxonomy to use depending on their needs and capabilities: choosing a strong criterion will necessitate stronger enforcement and will lead to more complicated proofs, while choosing a weaker criterion will lead to weaker security guarantees, but these guarantees might be sufficient for one's needs.

We start by describing in section 2.1 the secure compilation criteria proposed in [6] and providing a summary of that paper's results.

We then discuss in detail two aspects of this work. First (section 2.2), we discuss a simple secure compiler that compiles programs with static types to programs with dynamic type checks. We prove that this compiler satisfies one of the criteria, using a simple proof technique based on the back-translation of finite sets of finite traces. This criterion is still strong enough to imply the preservation of important security properties such as noninterference and trace equivalence. We compare that proof to one of a much stronger criterion, but that is also more complex, and we argue that our first proof is a better fit for the mechanization of secure compilers.

Then, we discuss some shortcomings of the standard secure compilation criterion known as "fully abstract compilation" that lead to the creation of these new criteria (section 2.3). We show that there are very simple compilers that satisfy the criterion "fully abstract compilation", but do not preserve even very simple security properties.

Author contribution The work presented in this chapter is based on [6], of which I, the author of this thesis, am a co-author. I contributed to all parts of this paper, including the development of the taxonomy of criteria, but my main contribution resides in the examples showcasing shortcomings of fully abstract compilation and in the design of the example secure compiler, languages, and in the proof of the simpler criterion for this compiler.

[6]: Abate et al. (2019), 'Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation' **[Co-authored]**

2.1 Journey Beyond Full Abstraction: A Summary

What is a good security criterion for a compiler? For the longest time, people have claimed that it is what's colloquially known as *Fully Abstract Compilation* (or *Full Abstraction*). Textbooks eloquently define full abstraction as "the preservation and reflection of observational equivalence", where "preservation" and "reflection" refer to each of the two directions of an equivalence, and "observational equivalence" is some notion of equivalence between partial programs.

Definition 2.1.1 (Fully Abstract Compiler) *Given two languages $\mathcal{S}$ and $\mathcal{T}$, two equivalences between programs, one in the source, $=_o$, and one in the target, $=_o$, a compiler $\downarrow_{\mathcal{T}}^{\mathcal{S}}$ from the source language to the target language is said to be* fully abstract *when:*

$$\forall\, \mathbf{P_1}\, \mathbf{P_2}.\ \mathbf{P_1} =_o \mathbf{P_2} \iff \mathbf{P_1}\downarrow_{\mathcal{T}}^{\mathcal{S}} =_o \mathbf{P_2}\downarrow_{\mathcal{T}}^{\mathcal{S}}.$$

Obviously, to understand what such a definition could mean, we must explain how these "observational equivalences" are instanced. The way it is typically done is by using "contextual equivalence": two partial programs are equivalent when, whenever they are linked with the same context, the resulting whole programs are behaviourally equivalent.[1]

1: Below, we give some definitions for the source language, but the same can be defined for the target language, replacing source elements with target elements.

Definition 2.1.2 (Contextual Equivalence) *Two partial programs $\mathbf{P_1}$ and $\mathbf{P_2}$ are* contextually equivalent *when:*

$$\forall\, \mathbf{C}.\ \mathbf{C} \bowtie \mathbf{P_1} \equiv \mathbf{C} \bowtie \mathbf{P_2}$$

where $\bowtie$ represents linking and $\equiv$ is the behavioural equivalence.

There are many ways to define behavioural equivalence. In this thesis, we are mainly concerned about trace-based notions, so we define behavioural equivalence as *trace equivalence*. Assuming that there is a set of traces Trace, and that languages are equipped with trace-producing semantics, two trace-equivalent programs must produce exactly the same traces.

Works on full abstraction for secure compilation typically consider traces made of a single event: termination or divergence

Definition 2.1.3 (Trace Equivalence) *Given a trace-producing semantics $\rightsquigarrow$ for our language, we say that two whole programs $\mathbf{W_1}$ and $\mathbf{W_2}$ are* trace-equivalent *when:*

$$\forall\, t \in \mathsf{Trace}.\ \mathbf{W_1} \rightsquigarrow t \iff \mathbf{W_2} \rightsquigarrow t.$$

Using these definitions, we can unfold full abstraction fully, and obtain the following definition: $\downarrow_{\mathcal{T}}^{\mathcal{S}}$ is fully abstract iff:

$$\begin{aligned}&\forall\, \mathbf{P_1}\, \mathbf{P_2}.\ (\forall\, \mathbf{C}\, t.\ \mathbf{C} \bowtie \mathbf{P_1} \rightsquigarrow t \Leftrightarrow \mathbf{C} \bowtie \mathbf{P_2} \rightsquigarrow t)\\ &\quad\iff \left(\forall\, \mathsf{C}\, t.\ \mathsf{C} \bowtie \mathbf{P_1}\downarrow_{\mathcal{T}}^{\mathcal{S}} \rightsquigarrow t \Leftrightarrow \mathsf{C} \bowtie \mathbf{P_2}\downarrow_{\mathcal{T}}^{\mathcal{S}} \rightsquigarrow t\right).\end{aligned}$$

Fully abstract compilation can be read as stating that there is no context in the source that distinguishes two source programs if and only if there is no context in the target that distinguishes the programs after compilation. In particular, the most interesting direction for security ($\implies$, from source to target), can be read as follows: if two source programs, whenever linked with the same context, produce the same traces, then the two compiled programs also produce the same traces

when linked with the same context. An alternative — perhaps easier to understand — reading can be obtained by taking this direction and going to the contrapositive: if there is a distinguishing context between two compiled programs, meaning there is a target context that makes the two compiled programs produce different traces when linked with them, then there was already a source context that made the two source programs produce different traces.

The issue with trace equivalence, however, is that it isn't the only security property one may want to *robustly preserve* (i.e., preserve even against arbitrary adversarial contexts). Instead, one could be interested in preserving, for instance, classes of trace properties such as safety [24] or liveness [25], or of hyperproperties [26] such as hypersafety, which famously includes many variants of noninterference [27], the state of the art security property that covers data confidentiality and integrity. Later in this chapter, we show that full abstraction is generally not strong enough on its own to imply the robust preservation of any of these properties (section 2.3); and at the same time, the kind of protections one has to put in place for achieving full abstraction seem too complex if one is only interested in safety or hypersafety. Indeed, it is significantly harder to hide the differences between two programs that are observationally equivalent (in the source) but otherwise arbitrary from a low-level attacker, than it is to solely protect the internal invariants and the secret data of a single program. Thus, a secure compilation chain for robust safety or hypersafety will likely be more efficient and easier to implement than one for observational equivalence. Moreover, in the presence of side-channels that can be exploited by low-level attackers, hiding the differences between two observationally equivalent programs in the source is hopeless. Meanwhile, robustly preserving safety properties is not a problem and even robustly preserving noninterference seems possible in specific scenarios [28]. Finally, proving full abstraction is notoriously challenging even for simple languages.

[24]: Lamport et al. (1984), 'Formal Foundation for Specification and Verification'

[25]: Alpern et al. (1985), 'Defining liveness'

[26]: Clarkson et al. (2008), 'Hyperproperties'

[27]: Goguen et al. (1982), 'Security Policies and Security Models'

[28]: Barthe et al. (2018), 'Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time"'

It follows that there is no "one-size-fits-all" criterion for secure compilation and linking against adversarial low-level code. The stronger the guarantees we aim to achieve, the harder it is to enforce them efficiently; hence, one should aim to find a balance between security guarantees, realizability, and efficiency. To this aim, the paper explores a large space of secure compilation criteria based on the notion of robust property preservation. Some of these criteria are strictly stronger than full abstraction, and, more importantly, imply directly the preservation of well-known classes of properties such as safety and hypersafety properties. At the same time, some other criteria that are introduced in this paper seem easier to achieve and prove, but only imply the preservation of smaller classes of properties (such as just safety, without hypersafety). These criteria can still prove useful, as these classes also contain properties that can be of interest.

All the criteria in the paper are phrased in terms of execution traces $t \in \mathsf{Trace}$, just like the criterion above, and these traces should be seen as (finite or infinite) sequences of events such as inputs from and outputs to an external environment [22, 29], and possibly recording the termination behaviour of the execution.

[22]: Leroy (2009), 'A Formally Verified Compiler Back-end'

[29]: Kumar et al. (2014), 'CakeML: a verified implementation of ML'

The general shape of the criteria is as follows: if one or several programs satisfy a property belonging to a certain class of security properties when linked against any source context, so do their compiled versions when linked against any target context. How many programs and what class of security properties is considered are what parametrizes the

criterion. For instance, one can consider one program and only the class of safety properties (properties of the form "never does X"), obtaining RSP; one program, and the class of all (single-) trace properties, obtaining RTP; one program and the class of hypersafety properties (properties such as noninterference, which consider several executions of the same program), obtaining RHSP; or n programs and the class of relational hyperproperties (properties relating the executions of different programs), obtaining RrHP.

All these criteria also have an equivalent property-free characterization that is obtained by taking the contrapositive. The intuition behind this characterization is to say that if compiled programs violate a property of the considered class (witnessed by one or several traces) when linked with some target context, then the source programs also violate the same property when linked with some particular source context. This characterization suggests a proof technique: given a target context and such traces, show there exists a source context that, when linked with the source programs, results in the same traces.

We will now make this more concrete to discuss a few particular criteria of interest.

2.1.1 Robust Trace Property Preservation

The first criterion we discuss here is called RTP, for *Robust Trace Property Preservation*. RTP characterizes compilation chains that preserve all trace properties satisfied against adversarial contexts, and is perhaps the simplest criterion to define.

Trace properties are defined simply as sets of allowed traces [24]. This allows us to define what it means for a trace property to be satisfied by a program.

[24]: Lamport et al. (1984), 'Formal Foundation for Specification and Verification'

Definition 2.1.4 (Trace property satisfaction) *A whole program W is said to* satisfy *a trace property π when the set of traces produced by W is a subset of π or, as a formula, when $\{t \mid W \rightsquigarrow t\} \subseteq \pi$.*

From that, it is possible to generalize that definition and define what it means for a *partial program* P to *robustly satisfy* a trace property [30, 31]:

[30]: Kupferman et al. (1999), 'Robust Satisfaction'
[31]: Swasey et al. (2017), 'Robust and compositional verification of object capability patterns'

Definition 2.1.5 (Robust trace property satisfaction) *A partial program P is said to* robustly satisfy *a trace property π when the set of traces produced by P when linked with any context C is a subset of π or, as a formula, when $\{t \mid \exists C.\ C \bowtie P \rightsquigarrow t\} \subseteq \pi$, where $\bowtie$ is the linking operation.*

Armed with this, *Robust Trace Property Preservation* (RTP) is defined as the preservation of robust satisfaction of all trace properties.[2]

2: Note the difference between *satisfaction* and *preservation*: satisfaction refers to programs at a single level (source or target) having some properties; preservation refers to the compilation chain between two levels (source and target) preserving satisfaction.

Definition 2.1.6 (Robust Trace Property Preservation (RTP)) *A compiler $\downarrow^{\mathcal{S}}_{\mathcal{T}}$ satisfies* RTP *when:*

$$\begin{aligned}\mathsf{RTP}: \quad \forall \pi \in 2^{\mathsf{Trace}}.\ \forall \mathbf{P}.\ (\forall \mathbf{C}_{\mathcal{S}}\ t.\ \mathbf{C}_{\mathcal{S}} \bowtie \mathbf{P} \rightsquigarrow t \Rightarrow t \in \pi) \implies \\ (\forall \mathbf{C}_{\mathcal{T}}\ t.\ \mathbf{C}_{\mathcal{T}} \bowtie \mathbf{P}\downarrow^{\mathcal{S}}_{\mathcal{T}} \rightsquigarrow t \Rightarrow t \in \pi).\end{aligned}$$

Reading this definition aloud, if a source (partial) program $\mathbf{P}$ only produces traces satisfying a certain property π when linked against any source context $\mathbf{C}_{\mathcal{S}}$, then the compilation chain enforces that the compiled

program $\mathbf{P}{\downarrow}^{\mathcal{S}}_{\mathcal{T}}$ only produces traces satisfying the property π when linked against any target context $\mathbf{C}_{\mathcal{T}}$.

This definition directly captures which properties (specifically, all trace properties) of the source are robustly preserved by the compilation chain. However, it seems ill-suited for reasoning about the compilation chain, because it doesn't suggest any proof technique.

Thankfully, one can give an equivalent "property-free" characterization of RTP that fulfils this role of suggesting a proof technique. We call this criterion RTC.

Definition 2.1.7 (RTC) *A compiler* ${\downarrow}^{\mathcal{S}}_{\mathcal{T}}$ *satisfies* RTC *when:*

$$\begin{aligned}\mathsf{RTC}: \quad \forall\, \mathbf{P}.\, \forall\, \mathbf{C}_{\mathcal{T}}.\, \forall t.\, \mathbf{C}_{\mathcal{T}} \bowtie \mathbf{P}{\downarrow}^{\mathcal{S}}_{\mathcal{T}} \rightsquigarrow t \\ \implies \exists\, \mathbf{C}_{\mathcal{S}}.\, \mathbf{C}_{\mathcal{S}} \bowtie \mathbf{P} \rightsquigarrow t.\end{aligned}$$

Read aloud, if a compiled program $\mathbf{P}{\downarrow}^{\mathcal{S}}_{\mathcal{T}}$ linked with a particular context $\mathbf{C}_{\mathcal{T}}$ produces some trace t, then there exists a source context $\mathbf{C}_{\mathcal{S}}$ that, when linked with the source program $\mathbf{P}$, produces the same trace t. It is quite direct to prove the equivalence between the two criteria, but this is not what matters to us here.

What is actually remarkable is that this criterion suggests a proof technique to show a compilation chain satisfies it: given a low-level context and a trace, one simply has to build, either from the trace or the code of the context, or from both, a new high-level context $\mathbf{C}_{\mathcal{S}}$, in a process often called back-translation. What is also remarkable is that it is possible, for all criteria based on the preservation of classes of properties, to give a similar property-free characterization; the main difference between all of them is the amount of information that can be used to build the source context: sometimes, like here, one can use the trace and the program, and sometimes one has to build a context that works for all programs and traces.

2.1.2 Robust Safety Property Preservation (RSP)

Here, we describe *Robust Safety Property Preservation*, a criterion that is easier to achieve than the previous, but still very interesting and expressive [31]. This criterion has to do with *safety properties*: properties that describe what behaviour of a program isn't allowed.

[31]: Swasey et al. (2017), 'Robust and compositional verification of object capability patterns'

Definition 2.1.8 (Safety property) *A trace property* $\pi \in 2^{\mathsf{Trace}}$ *is a* safety property *(*$\pi \in \mathsf{Safety}$*) when:*

$$\forall\, t \notin \pi.\, \exists\, m \leq t.\, \forall\, t' \geq m.\, t' \notin \pi.$$

We write $m \leq t$ for the prefix relation between a finite trace prefix m and a trace t.

This definition of safety properties is standard [24], and makes use of *bad prefixes*: a safety property is any property that can be invalidated by a "bad prefix" m, a finite prefix of a trace that cannot be extended in a way that satisfies the property.

[24]: Lamport et al. (1984), 'Formal Foundation for Specification and Verification'

We can define RSP just like RTP, but changing the quantification over properties.

Definition 2.1.9 (Robust Safety Property Preservation (RSP)) *A compiler* $\downarrow^{\mathcal{S}}_{\mathcal{T}}$ *satisfies* RSP *when:*

$$\begin{aligned}\mathsf{RSP}: \quad \forall \pi \in \mathsf{Safety}.\, \forall \mathbf{P}.\, (\forall \mathbf{C}_{\mathcal{S}}\, t.\, \mathbf{C}_{\mathcal{S}} \bowtie \mathbf{P} \rightsquigarrow t \Rightarrow t \in \pi) \implies \\ (\forall \mathsf{C}_{\mathcal{T}}\, t.\, \mathsf{C}_{\mathcal{T}} \bowtie \mathbf{P}{\downarrow^{\mathcal{S}}_{\mathcal{T}}} \rightsquigarrow t \Rightarrow t \in \pi).\end{aligned}$$

And similarly, we get a property-free characterization, RSC.

Definition 2.1.10 (RSC) *A compiler* $\downarrow^{\mathcal{S}}_{\mathcal{T}}$ *satisfies* RSC *when:*

$$\begin{aligned}\mathsf{RSC}: \quad \forall \mathbf{P}.\, \forall \mathsf{C}_{\mathcal{T}}.\, \forall m.\, \mathsf{C}_{\mathcal{T}} \bowtie \mathbf{P}{\downarrow^{\mathcal{S}}_{\mathcal{T}}} \rightsquigarrow m \\ \implies \exists \mathbf{C}_{\mathcal{S}}.\, \mathbf{C}_{\mathcal{S}} \bowtie \mathbf{P} \rightsquigarrow m.\end{aligned}$$

We use the same symbol $\rightsquigarrow$ to denote the production of a finite trace prefix m instead of a full trace t.

In terms of proof technique, we note that m is a finite object, so it is easy to construct functions that manipulate such trace prefix. This considerably simplifies the task of back-translating: one could now write the back-translation as a recursive function of the trace prefix, for instance, and this is what we actually do in several examples in the remainder of this thesis.

2.1.3 Hyperproperties and Relational Hyperproperties

Finally, such secure compilation criteria can also be defined in terms of *hyperproperties* and *relational hyperproperties*.

Hyperproperties are properties not of single traces but of several traces of the same program [26]. The most well-known hyperproperty is non-interference [27, 32], which roughly states that public observations are independent of secrets.

[26]: Clarkson et al. (2008), 'Hyperproperties'
[27]: Goguen et al. (1982), 'Security Policies and Security Models'
[32]: McLean (1992), *Proving Noninterference and Functional Correctness Using Traces*

Relational hyperproperties, meanwhile, relate several executions of different programs. They can be used to capture relations between different programs such as "always runs faster than" or different kinds of program equivalence, such as trace equivalence.

We demonstrate two examples of criteria based on preserving relational hyperproperties that will be of relevance later.

The first one is RrHC, with rH standing for "relational hyperproperty".[3]

3: For relational hyperproperties, the property-free characterization is often simpler to state and understand; see the full paper for the characterization in terms of preservation of relational hyperproperties.

Definition 2.1.11 (RrHC) *A compiler* $\downarrow^{\mathcal{S}}_{\mathcal{T}}$ *satisfies* RrHC *when:*

$$\mathsf{RrHC}: \quad \forall \mathsf{C}_{\mathcal{T}}.\, \exists \mathbf{C}_{\mathcal{S}}.\, \forall \mathbf{P}.\, \mathsf{Behaviour}\left(\mathsf{C}_{\mathcal{T}} \bowtie \mathbf{P}{\downarrow^{\mathcal{S}}_{\mathcal{T}}}\right) = \mathsf{Behaviour}\left(\mathbf{C}_{\mathcal{S}} \bowtie \mathbf{P}\right)$$

RrHC requires that for every target context $\mathsf{C}_{\mathcal{T}}$, there exists a corresponding source context $\mathbf{C}_{\mathcal{S}}$ that behaves exactly the same. This is an extremely strong property, and this is reflected in the amount of information one can use when trying to prove it: the back-translated context, $\mathbf{C}_{\mathcal{S}}$, can only depend on the target context $\mathsf{C}_{\mathcal{T}}$, but not on any particular program or execution anymore!

Finally, the last criterion we present, RfrXC ("finite-relational relaxed hypersafety") is also based on relational hyperproperties.

Definition 2.1.12 (RfrXC) *A compiler* $\downarrow_{\mathcal{T}}^{\mathcal{S}}$ *satisfies* RfrXC *when:*

$$\begin{aligned}\text{RfrXC}: \quad & \forall K.\, \forall \mathbf{P_1} \ldots \mathbf{P_K}.\, \forall \mathbf{C_T}.\, \forall x_1 \ldots x_K. \\ & \left(\mathbf{C_T} \bowtie \mathbf{P_1}{\downarrow_{\mathcal{T}}^{\mathcal{S}}} \rightsquigarrow x_1 \wedge \cdots \wedge \mathbf{C_T} \bowtie \mathbf{P_K}{\downarrow_{\mathcal{T}}^{\mathcal{S}}} \rightsquigarrow x_K\right) \implies \\ & \exists \mathbf{C_S}.\, \left(\mathbf{C_S} \bowtie \mathbf{P_1} \rightsquigarrow x_1 \wedge \cdots \wedge \mathbf{C_S} \bowtie \mathbf{P_K} \rightsquigarrow x_K\right)\end{aligned}$$

where $x_1, \ldots, x_K$ *are "relaxed trace prefixes", i.e., trace prefixes that can also record silent divergence.*

Compared to RrHC, RfrXC only considers a finite number of programs and executions; and these executions it considers are always finite, with the exception that they can record silent divergence. Hence, this criterion is weaker than RrHC. Yet, it still implies the preservation of most interesting security properties (for instance, noninterference), and, in many settings, it implies full abstraction.[4] Proving RfrXC also should be in general easier than proving RrHC. Indeed, one can back-translate using more information than just the target context $\mathbf{C_T}$, and the back-translated context only has to be correct for a finite number of programs and executions. We demonstrate this in the next section, section 2.2, where we show a proof technique using a back-translation of a set of finite traces.

4: This is elaborated upon in the paper.

2.2 An Example of a Secure Compilation Chain

In this section, we demonstrate that the criteria we described previously can be enforced and proved, using a simple example of a compiler that replaces static type checks with dynamic type checks. We prove that this compiler satisfies the criterion RfrXC from the previous section, with a relatively simple proof technique based on the idea of back-translating finite sets of finite prefixes. This proof is generic, in the sense that it only depends on the model of interactions between the context and program, but not on the other details of the languages.

Then, we discuss another proof of security for this same compiler, but this time for the criterion RrHC. This second proof is based on a "universal embedding" [33], a type of code-based back-translation. Even though it allows for a stronger criterion, we argue that its lack of genericity and its complexity make it unsuitable for most practical work aimed at building and verifying secure compilers.

[33]: New et al. (2016), 'Fully abstract compilation via universal embedding'

2.2.1 Overview

The two languages we consider are simple first-order languages with named procedures and boolean and natural values. The source language $\mathbf{L}^{\tau}$ is typed while the target language L^{u} is untyped. A program in either language is a collection of function definitions, each function body is a pure expression that can perform comparison and natural operations ($\oplus$), conditional branching, recursive calls, and use let-in bindings. Expressions can also read naturals from the environment and write naturals to the environment, both of which generate trace events. L^{u} has all the features of $\mathbf{L}^{\tau}$ and adds a primitive $\mathsf{e}\ \mathsf{has}\ \tau$ to dynamically check whether an expression e has type τ. A context is an expression that can call functions and perform general computation on the returned values, but it cannot directly generate `read` n and `write` n events, as those are used to define our trace properties.[5] Since contexts are single

5: Another alternative to this choice would be to mark these events with their provenance. Then, it would become possible to define meaningful properties. For simplicity of notation we stick to the choice that only the program can produce events.

expressions, we disallow callbacks from the program to the context: thus calls go from context to program, and returns from program to context.

Despite the apparent simplicity of this setting, compiling $\mathbf{L}^{\tau}$ to L^{u} securely is not totally trivial!

Indeed, consider a source function

$$\mathbf{f(x : Nat) : Nat \mapsto return\ (write\ (x + 1))}$$

that takes a natural number and outputs its successor. The source language, via its type system, enforces that this function can only ever be called with a natural number argument, so, when called, it will always output some number and return. In other words, it's impossible to write a valid source context such as $\mathbf{e = f(true)}$.

The naïve way of compiling the function $\mathbf{f}$ above is to map $\mathbf{f}$ to

$$\mathsf{f(x) \mapsto return\ (write\ (x + 1))},$$

the same function syntactically but with the types removed. This would not be a problem when compiling whole programs, and such compiler would be correct. However, it would not be secure. Indeed, a L^{u} context could try to use f in an invalid manner. For instance, $\mathsf{e = f(true)}$ would lead to the program getting stuck before emitting the event.

So, what is the secure way to compile in this case? It is to insert dynamic type checks at the boundaries between context and program. For our example above, the compiler will insert a check that x has indeed the type Nat:

$$\mathsf{f(x) \mapsto return\ (if\ x\ has\ Nat\ then\ write\ (x + 1)\ else\ fail)}.$$

Then, instead of getting stuck, the compiled program will fail safely when it is called by a malicious context.

We prove that such a compiler, denoted $\downarrow$, achieves RfrXC (Definition 2.1.12).

Theorem 2.2.1 ($\downarrow$ is RfrXC) *Given K source programs $\mathbf{P_1}, \ldots \mathbf{P_K}$, one target context $\mathsf{C_T}$ and K trace prefixes (that also record the termination behaviour of the programs) such that:*

$$\mathsf{C_T} \bowtie \mathbf{P_1}{\downarrow} \rightsquigarrow x_1 \wedge \cdots \wedge \mathsf{C_T} \bowtie \mathbf{P_K}{\downarrow} \rightsquigarrow x_K.$$

There exists a source context $\mathbf{C_S}$ such that:

$$\mathbf{C_S} \bowtie \mathbf{P_1} \rightsquigarrow x_1 \wedge \cdots \wedge \mathbf{C_S} \bowtie \mathbf{P_K} \rightsquigarrow x_K.$$

To prove this result, we devise a technique based on back-translating the set of trace prefixes $\{x_1, \ldots, x_K\}$.

Here's a demonstration on a simple example. Consider the following two programs

$$\begin{aligned}
\mathbf{P_1} = \{&\mathbf{f(x:Nat) : Nat \mapsto return\ x},\\
&\mathbf{g(x:Nat) : Bool \mapsto return\ true}\}\\
\mathbf{P_2} = \{&\mathbf{f(x:Nat) : Nat \mapsto return\ read},\\
&\mathbf{g(x:Nat) : Bool \mapsto return\ true}\}
\end{aligned}$$

Their compiled counterparts are almost identical, with the only addition of dynamic type checks on function arguments:

$$\begin{aligned}\mathbf{P_1}\downarrow &= \{\mathsf{f(x)} \mapsto \mathsf{ret\ (if\ x\ has\ Nat\ then\ x\ else\ fail)},\\ &\quad\ \mathsf{g(x)} \mapsto \mathsf{return\ (if\ x\ has\ Nat\ then\ true\ else\ fail)}\}\\ \mathbf{P_2}\downarrow &= \{\mathsf{f(x)} \mapsto \mathsf{ret\ (if\ x\ has\ Nat\ then\ read\ else\ fail)},\\ &\quad\ \mathsf{g(x)} \mapsto \mathsf{return\ (if\ x\ has\ Nat\ then\ true\ else\ fail)}\}\end{aligned}$$

Now, consider the following target context:

$$\begin{aligned}\mathsf{e} = \mathsf{let}\ \mathsf{x}_1 &= \mathsf{call\ f\ 5\ in}\\ &\mathsf{if}\ \mathsf{x}_1 \geq 5\ \mathsf{then\ call\ g}\ \mathsf{x}_1\ \mathsf{else\ call\ g\ false}\end{aligned}$$

The two programs plugged into this context can generate (at least) the following traces (where $\Downarrow$ indicates termination and $\bot$ indicates failure):

$$\begin{aligned}\mathsf{e} \bowtie \mathbf{P_1}\downarrow &\rightsquigarrow \Downarrow\\ \mathsf{e} \bowtie \mathbf{P_2}\downarrow &\rightsquigarrow \mathtt{read}\ 5; \Downarrow\\ \mathsf{e} \bowtie \mathbf{P_2}\downarrow &\rightsquigarrow \mathtt{read}\ 0; \bot\end{aligned}$$

In the execution of $\mathsf{e} \bowtie \mathbf{P_1}\downarrow$ the program executes completely and terminates, producing no side effects. In the first execution of $\mathsf{e} \bowtie \mathbf{P_2}\downarrow$, the program reads 5, and the *then* branch of the context's conditional is executed. In the second execution of $\mathsf{e} \bowtie \mathbf{P_2}\downarrow$, the program reads 0, the *then* branch of the context's conditional is executed and the program fails in g after detecting a type error.

These traces alone are not enough to construct a source context since they do not record information about the control flow of program executions, specifically on which function produces which input or output. To recover this information we enrich execution prefixes with information about calls (from context to program) and returns (from program to context).

$$\textit{Interactions}\ \beta \triangleq \mathtt{call}\ f\ v \mid \mathtt{ret}\ v$$

We also define informative semantics, that are almost identical to the original ones. The only difference is that rules for calls and returns are enriched to now generate events to model these control flows between program and context. If a call or return occurs internally within the program, no trace event is generated since they are not relevant for back-translating the context. We write μ for informative trace prefixes, and we use $\hookrightarrow$ for the act of producing such informative trace prefixes with the informative semantics.

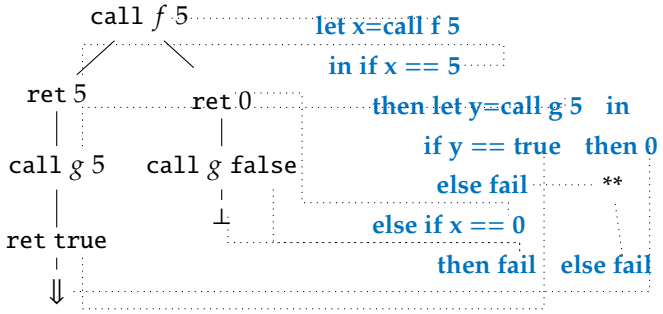
The informative traces produced by the compiled programs plugged into the context become:

$$\begin{aligned}\mathsf{e} \bowtie \mathbf{P_1}\downarrow &\hookrightarrow \mathtt{call}\ f\ 5; \qquad\quad \mathtt{ret}\ 5; \mathtt{call}\ g\ 5; \mathtt{ret}\ \mathtt{true}; \Downarrow\\ \mathsf{e} \bowtie \mathbf{P_2}\downarrow &\hookrightarrow \mathtt{call}\ f\ 5; \mathtt{read}\ 5; \mathtt{ret}\ 5; \mathtt{call}\ g\ 5; \mathtt{ret}\ \mathtt{true}; \Downarrow\\ \mathsf{e} \bowtie \mathbf{P_2}\downarrow &\hookrightarrow \mathtt{call}\ f\ 5; \mathtt{read}\ 0; \mathtt{ret}\ 0; \mathtt{call}\ g\ \mathtt{false}; \bot\end{aligned}$$

In our languages, reads and writes can only be performed by programs, while the context only performs a sequence of calls to the program, possibly performing some computation and branching on return values. Thus, the role of the back-translated source context is to perform the

appropriate calls to the program, depending on the values returned. The inner workings of the programs, that is inputs, outputs, and internal calls and returns, are not a concern of the back-translation and are obtained through compiler correctness. Furthermore, the context is shared by all executions, but each execution has its own program. Hence, since I/O occurs only in the program, the only source of variation among all executions comes from the program.

From this, one can conclude that the context is a deterministic expression, calling the program, and branching on the returned values. This can be seen in the way traces are organized: ignoring the I/O, the traces form a tree represented on the left. This tree can be translated to a source context using nested conditionals as depicted below on the right. When additional branches are missing (e.g., there is no third trace that analyzes the first return or no second trace that analyzes the second return on the left execution), the back-translation inserts **fail** in the code — they are dead code branches (marked with a **).



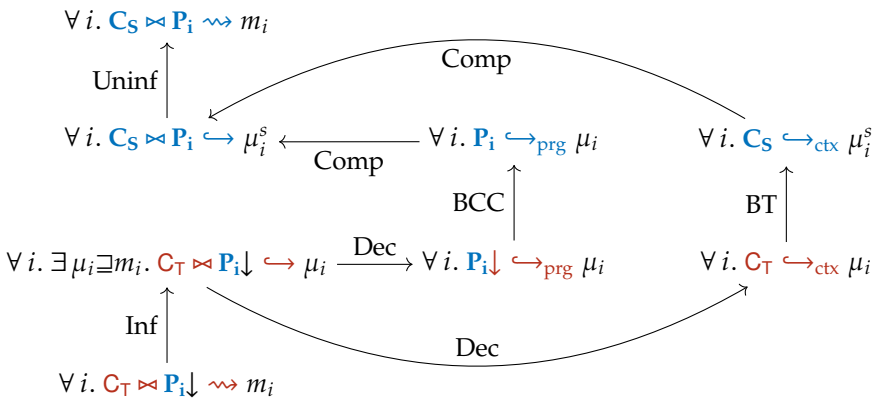
2.2.2 A Security Proof for this Compiler with a Trace-Based Back-Translation

We will now describe how we prove that this compiler, $\downarrow$, satisfies the RfrXC criterion.

The proof internally uses partial semantics: semantics that are given to partial programs ($\hookrightarrow_{\text{prg}}$) and contexts ($\hookrightarrow_{\text{ctx}}$). Partial semantics abstract away the other component, and we relate them to the whole program semantics by proving two theorems, *decomposition* and *composition*. Decomposition states that given a whole program execution, then the program part and the context part both have an execution in the partial semantics that produces the same (informative) trace prefix. Composition is the converse: given a partial execution for a partial program and a partial execution for a context that produce the same trace prefix, recomposition states that the whole program obtained by linking the two also produces the same trace prefix.

Before delving into the proof, we make one last remark: we will not be able to prove that the back-translated source context produces exactly the same informative trace prefixes as the target context. In the example above, since calling **g** with a boolean is ill-typed, our back-translated context cannot contain **call g false**. Instead, our back-translation shifts the failure from the program to the context, by inserting the failure in the context, hence the picture linking `call` g `false` action to a **fail**. This is reflected on the informative trace, and we capture this possible shift of failure with a notation, μ^s.

Let us now discuss the proof. The proof technique is better explained by looking at the proof diagram.



We start from the very bottom, having

$$\forall i.\ \mathbf{C_T} \bowtie \mathbf{P_i}\downarrow \rightsquigarrow m_i.$$

We first prove that, for each trace prefix m_i, there exists an informative trace μ_i that captures the same I/O events ($\sqsupseteq$) that can be produced by the same program:

$$\forall i.\ \exists \mu_i \sqsupseteq m_i.\ \mathbf{C_T} \bowtie \mathbf{P_i}\downarrow \hookrightarrow \mu_i.$$

This first step is almost trivial, as the informative semantics simply record more information than the non-informative ones.

We then use decomposition in the target language, showing that from the whole program executions, one can obtain partial program executions and context executions in the partial semantics that produce the same informative trace prefixes:

$$\forall i.\ \mathbf{P_i}\downarrow \hookrightarrow_{\text{prg}} \mu_i \text{ and}$$
$$\forall i.\ \mathbf{C_T} \hookrightarrow_{\text{ctx}} \mu_i.$$

We then treat the programs and the context differently. We know that the programs are compiled, and we rely on backward compiler correctness for partial semantics (BCC) to show that the source programs also produce the same informative traces in the source partial semantics:

$$\forall i.\ \mathbf{P_i} \hookrightarrow_{\text{prg}} \mu_i.$$

On the other side, for the context, we back-translate (BT) our set of traces as described previously to obtain a source context $\mathbf{C_S}$. We prove that this source context has the same traces in the partial semantics for the context, except that failure might happen one step earlier, just before a call instead of just after.

$$\forall i.\ \mathbf{C_S} \bowtie \mathbf{P_i} \hookrightarrow \mu_i^s$$

After this, we can use composition in the source language, to prove that given two executions in the source partial semantics, one for the program and one for the context, producing the same informative trace prefix

(modulo the final switch of failure), one can obtain a whole program execution producing this trace prefix:

$$\forall\, i.\ \mathbf{C_S} \bowtie \mathbf{P_i} \hookrightarrow \mu_i^s.$$

Finally, we just have to remove the informative events and go back to the standard semantics, to obtain the final result:

$$\forall\, i.\ \mathbf{C_S} \bowtie \mathbf{P_i} \rightsquigarrow m_i.$$

2.2.3 Discussion on the Proof

The proof we presented is interesting in several aspects that deserve to be discussed, and should be compared to another proof technique that can be used to prove the stronger criterion RrHC for the same compiler.

Proof via Universal Embedding This other proof technique is still using a back-translation, but this back-translation is code-based and uses a technique known as universal embedding [33] to embed the target context in the source. The idea behind universal embedding is that there exists a type in the source that can be used to represent all target values; then, wrappers are used to encode and decode values. Specifically here, this type is **Nat** and the back-translation encodes true as **0**, false as **1** and an arbitrary natural number n as **n + 2**.

[33]: New et al. (2016), 'Fully abstract compilation via universal embedding'

Based on this encoding, the back-translation is defined using two macros, $\mathbf{inject}_\tau(\mathbf{e})$ that takes an expression **e** of type τ and returns an expression of the back-translation type **Nat**; and $\mathbf{extract}_\tau(\mathbf{e})$ that takes an expression **e** of the back-translation type and returns an expression of type τ.

$$\begin{aligned}
\mathbf{inject}_{\mathbf{Nat}}(\mathbf{e}) &= \mathbf{e} + \mathbf{2} \\
\mathbf{inject}_{\mathbf{Bool}}(\mathbf{e}) &= \mathbf{if\ e\ then\ 1\ else\ 0} \\
\mathbf{extract}_{\mathbf{Nat}}(\mathbf{e}) &= (\mathbf{let\ x{=}e\ in\ if\ x} \geq \mathbf{2\ then\ x} - \mathbf{2\ else\ fail}) \\
\mathbf{extract}_{\mathbf{Bool}}(\mathbf{e}) &= \begin{pmatrix} \mathbf{let\ x{=}e\ in\ if\ x} \geq \mathbf{2\ then\ fail} \\ \mathbf{else\ if\ x} + \mathbf{1} \geq \mathbf{2\ then\ true\ else\ false} \end{pmatrix}
\end{aligned}$$

Then, the back-translation is defined on the code of the context:

$$\langle\!\langle \mathsf{true} \rangle\!\rangle^{\mathbf{L}^u}_{\mathbf{L}^\tau} = \mathbf{1} \qquad \langle\!\langle \mathsf{false} \rangle\!\rangle^{\mathbf{L}^u}_{\mathbf{L}^\tau} = \mathbf{0} \qquad \langle\!\langle \mathsf{n} \rangle\!\rangle^{\mathbf{L}^u}_{\mathbf{L}^\tau} = \mathbf{n} + \mathbf{2} \qquad \langle\!\langle \mathsf{x} \rangle\!\rangle^{\mathbf{L}^u}_{\mathbf{L}^\tau} = \mathbf{x}$$

$$
\begin{aligned}
\langle\!\langle \mathsf{e} \geq \mathsf{e}' \rangle\!\rangle^{\mathrm{L}^{\mathrm{u}}}_{\mathrm{L}^{\tau}} &= \mathbf{let\ x1 : Nat{=}extract_{Nat}}(\langle\!\langle \mathsf{e} \rangle\!\rangle^{\mathrm{L}^{\mathrm{u}}}_{\mathrm{L}^{\tau}}) \\
&\quad \mathbf{in\ let\ x2 : Nat{=}extract_{Nat}}(\langle\!\langle \mathsf{e}' \rangle\!\rangle^{\mathrm{L}^{\mathrm{u}}}_{\mathrm{L}^{\tau}}) \\
&\quad \mathbf{in\ inject_{Bool}}(\mathbf{x1 \geq x2}) \\
\langle\!\langle \mathsf{e} \oplus \mathsf{e}' \rangle\!\rangle^{\mathrm{L}^{\mathrm{u}}}_{\mathrm{L}^{\tau}} &= \mathbf{let\ x1 : Nat{=}extract_{Nat}}(\langle\!\langle \mathsf{e} \rangle\!\rangle^{\mathrm{L}^{\mathrm{u}}}_{\mathrm{L}^{\tau}}) \\
&\quad \mathbf{in\ let\ x2 : Nat{=}extract_{Nat}}(\langle\!\langle \mathsf{e}' \rangle\!\rangle^{\mathrm{L}^{\mathrm{u}}}_{\mathrm{L}^{\tau}}) \\
&\quad \mathbf{in\ inject_{Nat}}(\mathbf{x1 \oplus x2}) \\
\langle\!\langle \mathsf{let\ x{=}e\ in\ e}' \rangle\!\rangle^{\mathrm{L}^{\mathrm{u}}}_{\mathrm{L}^{\tau}} &= \mathbf{let\ x : Nat{=}}\langle\!\langle \mathsf{e} \rangle\!\rangle^{\mathrm{L}^{\mathrm{u}}}_{\mathrm{L}^{\tau}} \mathbf{\ in\ } \langle\!\langle \mathsf{e}' \rangle\!\rangle^{\mathrm{L}^{\mathrm{u}}}_{\mathrm{L}^{\tau}} \\
\left\langle\!\left\langle \begin{pmatrix} \mathsf{if\ e\ then} \\ \mathsf{e'\ else\ e''} \end{pmatrix} \right\rangle\!\right\rangle^{\mathrm{L}^{\mathrm{u}}}_{\mathrm{L}^{\tau}} &= \mathbf{if\ extract_{Bool}}(\langle\!\langle \mathsf{e} \rangle\!\rangle^{\mathrm{L}^{\mathrm{u}}}_{\mathrm{L}^{\tau}}) \mathbf{\ then\ } \langle\!\langle \mathsf{e}' \rangle\!\rangle^{\mathrm{L}^{\mathrm{u}}}_{\mathrm{L}^{\tau}} \mathbf{\ else\ } \langle\!\langle \mathsf{e}'' \rangle\!\rangle^{\mathrm{L}^{\mathrm{u}}}_{\mathrm{L}^{\tau}} \\
\langle\!\langle \mathsf{e\ has\ Bool} \rangle\!\rangle^{\mathrm{L}^{\mathrm{u}}}_{\mathrm{L}^{\tau}} &= \mathbf{let\ x : Nat{=}}\langle\!\langle \mathsf{e} \rangle\!\rangle^{\mathrm{L}^{\mathrm{u}}}_{\mathrm{L}^{\tau}} \mathbf{\ in\ if\ x \geq 2\ then\ 0\ else\ 1} \\
\langle\!\langle \mathsf{e\ has\ Nat} \rangle\!\rangle^{\mathrm{L}^{\mathrm{u}}}_{\mathrm{L}^{\tau}} &= \mathbf{let\ x : Nat{=}}\langle\!\langle \mathsf{e} \rangle\!\rangle^{\mathrm{L}^{\mathrm{u}}}_{\mathrm{L}^{\tau}} \mathbf{\ in\ if\ x \geq 2\ then\ 1\ else\ 0} \\
\langle\!\langle \mathsf{call\ f\ e} \rangle\!\rangle^{\mathrm{L}^{\mathrm{u}}}_{\mathrm{L}^{\tau}} &= \mathbf{inject_{\tau'}(call\ f\ extract_{\tau}}(\langle\!\langle \mathsf{e} \rangle\!\rangle^{\mathrm{L}^{\mathrm{u}}}_{\mathrm{L}^{\tau}})) \\
&\quad \text{if } \mathbf{f} : \tau \to \tau' \in \bar{\mathbf{I}} \\
\langle\!\langle \mathsf{fail} \rangle\!\rangle^{\mathrm{L}^{\mathrm{u}}}_{\mathrm{L}^{\tau}} &= \mathbf{fail}
\end{aligned}
$$

As one can see, the back-translation is complex; the proof of security using this back-translation also ends up being complex. Basically, it relies on a cross-language Kripke logical relation to relate target terms and their back-translation.

Benefits of the trace-based back-translation We see several benefits to the trace-based back-translation, compared to the code-based back-translation we just described.

First, the trace-based proof uses a back-translation that is in essence quite simple and generic: the back-translation takes a single input, a finite set of finite trace prefixes recording calls and returns between the program and the context. This is a very generic trace model that can be used in many more settings, including settings where the difference between source and target languages is much bigger than here. For instance, one could assign the same trace model to both a high-level functional language and a low-level assembly-like language with a flat memory model, and there would still be hope that the back-translation function could be used. The same cannot be said of the code-based back-translation: the back-translation is very specific to the target and source language; and it would be difficult to write such back-translation for very different languages, because one would have to find the right encoding for very low-level values in a high level language.

In terms of mechanization in a proof assistant, the proof using a trace-based back-translation should be much easier to mechanize. Indeed, the back-translation function is very easy to define, and the proof mainly uses basic induction and simple invariants.[6] Conversely, it is notoriously known that logical relations are difficult to implement properly in proof assistants, even for the simplest uses [34].

6: We refer the reader to the appendix of [6] for the proof details.

[34]: Pientka (2018), 'POPLMark reloaded: mechanizing logical relations proofs (invited talk)'

On the usage of partial semantics One drawback of the trace-based proof is using partial semantics, that is semantics of partial programs, for both the source and the target. Partial semantics are difficult to use, as they are inherently nondeterministic; moreover, existing correct compilers

rarely come with proofs of correctness for partial semantics (or even come with partial semantics for that matter). Ideally, one would not have to define and use partial semantics, and could instead use a proof technique that only uses whole program semantics. This would alleviate the effort of proving security, while also potentially allowing the reuse of existing compiler correctness theorems more directly.

In the next chapter (chapter 3), we will discuss one step towards this goal: a proof technique that is designed to reuse existing compiler correctness theorems proofs, in that case those similar to CompCert's. Unfortunately, the proof we discuss in the next chapter does still use partial semantics and a composition and decomposition theorem, albeit at a single level (in the target).

In chapter 4, we will present the first contribution of this thesis: an adaption of that proof technique that allows getting rid of partial semantics.

2.3 Fully Abstraction Compilation and Robust Property Preservation

Before proceeding with the rest of the thesis, we elaborate on a claim from section 2.1: "full abstraction is generally not strong enough on its own to imply the robust preservation of any of these properties".

To justify this claim, we describe compilers for very simple languages that enjoy full abstraction and compiler correctness but not even the weakest robust property preservation criteria. This, we argue, suggests that fully abstract compilation, even if achievable, is not the right criterion for secure compilation.

Fully Abstract Compilation Does Not Imply the Weakest Criteria

In this section, we define a compilation chain that does satisfy compiler correctness and fully abstract compilation, but not the two weakest criteria of Abate et al. [6]: RSP and RDP, the former being about safety properties and the latter about dense properties, the equivalent of liveness in the trace model considered here.

[6]: Abate et al. (2019), 'Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation' **[Co-authored]**

First, we state three notions of correctness, from the weakest to the strongest. We will show that our counterexample satisfies these three properties.

Definition 2.3.1 (Backward Compiler Correctness (BCC)) *A compiler* $\downarrow$ *satisfies* Backward Compiler Correctness *when:*

$$\mathsf{BCC}: \quad \forall\, t\ \mathbf{W}.\ \mathbf{W}{\downarrow} \rightsquigarrow t \implies \mathbf{W} \rightsquigarrow t$$

Definition 2.3.2 (Separate Compiler Correctness (SCC)) *A compiler* $\downarrow$ *satisfies* Separate Compiler Correctness *when:*

$$\mathsf{SCC}: \quad \forall\, \mathbf{P}\ \mathbf{C_S}\ t.\ \mathbf{C_S}{\downarrow} \bowtie \mathbf{P}{\downarrow} \rightsquigarrow t \implies \mathbf{C_S} \bowtie \mathbf{P} \rightsquigarrow t$$

Definition 2.3.3 (Compositional Compiler Correctness (CCC)) *A compiler* $\downarrow$ *satisfies* Compositional Compiler Correctness *when there exists a relation* $\approx$ *such that:*

$$\begin{aligned}\mathsf{CCC}: \quad & \forall\, \mathbf{P}\, \mathsf{C_T}\, \mathbf{C_S}\, t. \\ & \mathsf{C_T} \approx \mathbf{C_S} \wedge \mathbf{C_S} \textit{ is linkable with } \mathbf{P} \implies \\ & \quad \mathsf{C_T} \bowtie \mathbf{P}{\downarrow} \rightsquigarrow t \implies \\ & \quad \mathbf{C_S} \bowtie \mathbf{P} \rightsquigarrow t\end{aligned}$$

We also recall robust safety preservation and fully abstract compilation.

Definition 2.3.4 (Robust Safety Property Preservation (RSP)) *A compiler* $\downarrow$ *satisfies* RSP *when:*

$$\begin{aligned}\mathsf{RSP}: \quad & \forall\, \pi \in \mathsf{Safety}.\, \forall\, \mathbf{P}.\, (\forall\, \mathbf{C_S}\, t.\, \mathbf{C_S} \bowtie \mathbf{P} \rightsquigarrow t \Rightarrow t \in \pi) \implies \\ & \qquad (\forall\, \mathsf{C_T}\, t.\, \mathsf{C_T} \bowtie \mathbf{P}{\downarrow} \rightsquigarrow t \Rightarrow t \in \pi).\end{aligned}$$

Definition 2.3.5 (Fully Abstract Compilation) *A compiler* $\downarrow$ *satisfies* Fully Abstract Compilation *when:*

$$\begin{aligned}& \forall\, \mathbf{P_1}\, \mathbf{P_2}.\, (\forall\, \mathbf{C}\, t.\, \mathbf{C} \bowtie \mathbf{P_1} \rightsquigarrow t \Leftrightarrow \mathbf{C} \bowtie \mathbf{P_2} \rightsquigarrow t) \\ & \quad \iff (\forall\, \mathsf{C}\, t.\, \mathsf{C} \bowtie \mathbf{P_1}{\downarrow} \rightsquigarrow t \Leftrightarrow \mathsf{C} \bowtie \mathbf{P_2}{\downarrow} \rightsquigarrow t)\,.\end{aligned}$$

And we finally define the robust preservation of dense properties:

Definition 2.3.6 (Robust Dense Property Preservation (RDP)) *A compiler* $\downarrow$ *satisfies* RDP *when:*

$$\begin{aligned}\mathsf{RDP}: \quad & \forall\, \pi \in \mathsf{Dense}.\, \forall\, \mathbf{P}.\, (\forall\, \mathbf{C_S}\, t.\, \mathbf{C_S} \bowtie \mathbf{P} \rightsquigarrow t \Rightarrow t \in \pi) \implies \\ & \qquad (\forall\, \mathsf{C_T}\, t.\, \mathsf{C_T} \bowtie \mathbf{P}{\downarrow} \rightsquigarrow t \Rightarrow t \in \pi).\end{aligned}$$

where dense properties are exactly the properties that accept at least all terminating traces (in other words, they can only be invalidated by infinite executions).

Armed with those definitions, we can now define the languages and the compiler we consider.

Trace Model We consider languages where exactly one event is produced containing a natural number that represents the final result of the computation. Allowed traces are final result singletons and silent divergence.

Source Language A source language program consists of one function obtaining one input from the context (a natural number or a boolean), performing basic computations on it, and returning a natural number as a result.

$$\begin{aligned}\textit{Program } \mathbf{P} &\triangleq \mathbf{f(x : Nat) \mapsto e} \mid \mathbf{f(x : Bool) \mapsto e} \\ \textit{Expression } \mathbf{e} &\triangleq \mathbf{if\ x\ then\ e\ else\ e} \mid \mathbf{if\ x < n\ then\ e\ else\ e} \mid \mathbf{n} \mid \mathbf{f(e)} \\ \textit{Context } \mathbf{C} &\triangleq \mathbf{f(n)} \mid \mathbf{f(b)}\end{aligned}$$

In this example, the composition of a program and a context of incompatible types is statically disallowed. We do not consider them in the criteria we prove, and implicitly assume that the criteria only apply when the operation $\bowtie$ is defined.

Target Language The target language is identical to the source, but it only admits natural numbers as inputs. As a consequence, linking is always allowed.

$$\begin{aligned} \textit{Program}\ \mathsf{P} &\triangleq \mathsf{f(x : Nat) \mapsto e} \\ \textit{Expression}\ \mathsf{e} &\triangleq \mathsf{if\ x < n\ then\ e\ else\ e} \mid \mathsf{n} \mid \mathsf{f(e)} \\ \textit{Context}\ \mathsf{C} &\triangleq \mathsf{f(n)} \end{aligned}$$

Compiler We define the compiler as one that maps natural numbers to natural numbers, and booleans to 0 and 1. We first define the compilation of expressions:

$$\begin{aligned} (\mathbf{if\ x\ then\ e_1\ else\ e_2})\downarrow &= \mathsf{if\ x < 1\ then}\ \mathbf{e_1}\downarrow\ \mathsf{else}\ \mathbf{e_2}\downarrow \\ (\mathbf{if\ x < n\ then\ e_1\ else\ e_2})\downarrow &= \mathsf{if\ x < n\ then}\ \mathbf{e_1}\downarrow\ \mathsf{else}\ \mathbf{e_2}\downarrow \\ \mathbf{n}\downarrow &= \mathsf{n} \\ \mathbf{true}\downarrow &= \mathsf{0} \\ \mathbf{false}\downarrow &= \mathsf{1} \\ \mathbf{f(e)}\downarrow &= \mathsf{f(}\mathbf{e}\downarrow\mathsf{)} \end{aligned}$$

Then, we define the compilation of partial programs:

$$\begin{aligned} (\mathbf{f(x : Nat) \mapsto e})\downarrow &= \mathsf{f(x : Nat) \mapsto}\ \mathbf{e}\downarrow \\ (\mathbf{f(x : Bool) \mapsto e})\downarrow &= \mathsf{f(x : Nat) \mapsto if\ x < 2\ then}\ \mathbf{e}\downarrow \\ &\qquad \mathsf{else\ if\ x < 3\ then\ f(2)} \\ &\qquad \mathsf{else\ 42} \end{aligned}$$

We can trivially extend this compiler to contexts, and whole programs as well.

The compiler is fully standard, except it introduces two new branches when compiling functions taking a boolean as an input. The first branch happens when the compiled function receives 2 as an argument, and it starts looping. The second branch happens when the compiled function receives anything other than 0, 1, or 2 as an input, and it returns a particular number, here 42.

Lemma 2.3.1 $\downarrow$ *satisfies* BCC, SCC, *and* CCC.

Proof. We start by noting that for this compilation chain, BCC is implied by SCC. Indeed, let $\mathbf{W}$ be a source whole program. Then, by definition, there exists $\mathbf{C}$ and $\mathbf{P}$ such that $\mathbf{W} = \mathbf{C} \bowtie \mathbf{P}$. Furthermore, $\mathbf{W}\downarrow = \mathbf{C}\downarrow \bowtie \mathbf{P}\downarrow$ by definition of the compiler. Hence, to show BCC, one can simply apply SCC to this decomposition of $\mathbf{W}$ into two components.

We then show that for this compilation chain CCC is also implied by SCC. To show this, we need to instantiate the relation $\approx$ and the notion of linkable components in the definition of CCC:

- $\approx$ is defined by $\mathsf{C_T} \approx \mathbf{C_S}$ iff $\mathsf{C_T} = \mathbf{C_S}\downarrow$.

- $\mathbf{P}$ and $\mathbf{C_S}$ are linkable iff they agree on the argument type of the program's function.

Then, by unfolding these definitions in CCC, and substituting equalities, we obtain exactly SCC.

So all that is left to prove is SCC. Suppose $\mathbf{C}{\downarrow} \bowtie \mathbf{P}{\downarrow} \rightsquigarrow t$. We will show that $\mathbf{C} \bowtie \mathbf{P} \rightsquigarrow t$, proceeding by case analysis on $\mathbf{P}$:

- $\mathbf{P} = \mathbf{f(x : Nat)} \mapsto \mathbf{e}$. By induction on $\mathbf{e}$, we can prove that $\mathbf{e}{\downarrow} = \mathbf{e}$: indeed, because the function argument $\mathbf{x}$ must be a natural number, and because $\mathbf{x}$ is the only "variable" ever in scope, $\mathbf{e}$ cannot contain a subexpression $\mathbf{if\ x\ then\ e_1\ else\ e_2}$. Otherwise, it would mean that $\mathbf{x}$ is a boolean, this would be a contradiction.
- $\mathbf{P} = \mathbf{f(x : Bool)} \mapsto \mathbf{e}$. In this case, we know that

  $$\mathbf{P}{\downarrow} = \mathsf{f(x : Nat)} \mapsto \mathsf{if\ x < 2\ then}\ \mathbf{e}{\downarrow}\ \mathsf{else\ if\ x < 3\ then\ f(2)\ else\ 42}.$$

  Since the target context is a compiled context, this steps to the expression $\mathbf{e}{\downarrow}$. Now, $\mathbf{f}$ cannot be called recursively in $\mathbf{e}$, because expressions are natural numbers, but $\mathbf{f}$ expects a boolean. Hence, the same is true in the compiled version. Now, we can conclude by proving the thesis by induction on $\mathbf{e}$. In particular, note that $(\mathbf{if\ x\ then\ e_1\ else\ e_2}){\downarrow} = \mathsf{if\ x < 1\ then}\ \mathbf{e_1}{\downarrow}\ \mathsf{else}\ \mathbf{e_2}{\downarrow}$, that if $\mathbf{x} = \mathbf{true}$ then $\mathsf{x < 1}$ evaluates to true, and if $\mathbf{x} = \mathbf{false}$ then $\mathsf{x < 1}$ evaluates to false, where x is the compilation of $\mathbf{x}$.

□

Lemma 2.3.2 ↓ *satisfies fully abstract compilation.*

Proof. We start by proving the contrapositive form of the statement of the $\Longrightarrow$ implication.

Let $\mathbf{P_1}$ and $\mathbf{P_2}$ be two programs and suppose their compilations are not observationally equivalent; let $\mathsf{C = f(n)}$ be the distinguishing context. We consider three cases:

- $\mathbf{P_1}$ and $\mathbf{P_2}$ both expect a natural number. Take $\mathbf{C = f(n)}$. Since the compiler is the identity for programs that expect a natural number, we obtain the desired result.
- $\mathbf{P_1}$ and $\mathbf{P_2}$ both expect a boolean. If $\mathsf{n = 0}$ or $\mathsf{n = 1}$, then by taking $\mathbf{C = f(true)}$ in the first case, and $\mathbf{C = f(false)}$ in the second case, we obtain the desired result by compiler correctness. Otherwise, this case is discharged by contradiction: $\mathsf{C} \bowtie \mathbf{P_1}{\downarrow}$ and $\mathsf{C} \bowtie \mathbf{P_2}{\downarrow}$ have the same behaviour, by definition of the compiler.
- $\mathbf{P_1}$ and $\mathbf{P_2}$ have different input types. This is a case we do not consider, because then the source context cannot be linked with both programs.

The other direction of the proof is similar, and thus we obtain that ↓ is fully abstract. □

Lemma 2.3.3 ↓ *does not satisfy* RSP.

Proof. Consider the program $\mathbf{P = f(x : Bool)} \mapsto \mathbf{1}$. This program satisfies the safety property "never outputs 42," but its compilation does not (it violates it with input $\mathsf{3}$, for instance). □

Lemma 2.3.4 ↓ *does not satisfy* RDP.

Proof. Consider the same program $\mathbf{P} = \mathbf{f(x : Bool) \mapsto 1}$. This program satisfies the safety property "never silently diverge," but its compilation does not (it violates it with input 2). □

Hence, we have our counterexample: a compilation chain that is fully abstract and correct yet doesn't preserve any class of properties. The issue lies in the fact that full abstraction doesn't dictate what should happen when the compiled program receives an invalid input from a context, but only that programs should respond to it uniformly.

Fully Abstract Compilation Does Not Imply the Preservation of Noninterference

We now extend this compilation chain to show that fully abstract compilation doesn't imply preservation of noninterference either. We introduce a new command at the target level, leak, to model information leakage. The semantics of this new instruction is simple: leak reduces to a non-deterministically chosen natural number. Hence, this command can model looking-up the value of a secret inside memory, and outputting it publicly. All outputs are considered public.

Then, we also modify the compilation of partial programs having a boolean argument:

$$
\begin{aligned}
&(\mathbf{f(x : Bool) \mapsto e})\downarrow = \\
&\quad \mathsf{f(x : Nat) \mapsto if\ x < 2\ then}\ \mathbf{e}\downarrow \\
&\qquad\qquad \mathsf{else\ if\ x < 3\ then\ f(2)} \\
&\qquad\qquad \mathsf{else\ if\ x < 4\ then\ 42} \\
&\qquad\qquad \mathsf{else\ leak()}
\end{aligned}
$$

Simply put, the compiler adds one more error case, that triggers the leak.

It is easy to show that the previous lemmas still hold.

We define termination insensitive noninterference in the following manner: a program satisfies termination insensitive noninterference if and only if its public output doesn't depend on the private inputs. In particular, here there are no private inputs, so termination insensitive noninterference reduces to whole programs being deterministic.

It is easy to show that the source programs all satisfy termination-insensitive noninterference, because whole source programs are completely deterministic.

Now, consider a source program $\mathbf{P}$ with a boolean argument, and the target context $\mathsf{C_T = f(3)}$. Then, $\mathsf{C_T} \bowtie \mathbf{P}\downarrow \rightsquigarrow 0$, and $\mathsf{C_T} \bowtie \mathbf{P}\downarrow \rightsquigarrow 1$. This whole program is not noninterferent, so we have a counterexample.

Discussion

We just showed that full abstraction, even in the presence of other criteria such as compiler correctness, does not imply the preservation of even the simplest security criteria and properties we would expect. However, we are not saying that all the compilers that were proven fully abstract were insecure.

Indeed, we must concede that the compilers we showed here are intentionally designed to break these properties, and that it's quite unlikely that someone would intentionally write compilers like that if they intend for them to be secure. Moreover, as demonstrated by the proof of RrHP from section 2.2, proof techniques for full abstraction should be applicable without too much modification to proofs of some of the criteria presented in [6].

It is still unclear what exactly is the relation between fully abstract compilation and robust preservation-based criteria in general. See for instance [35, 36] for a formal account of the relation between the two in the framework of Mathematical Operational Semantics.

[6]: Abate et al. (2019), 'Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation' **[Co-authored]**

[35]: Abate et al. (2020), 'The Fox and the Hound: Comparing Fully Abstract and Robust Compilation'

[36]: Abate et al. (2021), 'Fully Abstract and Robust Compilation: And How to Reconcile the Two, Abstractly'

Secure Compilation, Compartmentalization and Undefined Behaviour

3

How can one characterize the security of a compartmentalization mechanism, when compartments can dynamically become compromised as they encounter undefined behaviours? Abate et al. [8] provide an answer to this question by defining RSCC, a secure compilation criterion that provides strong guarantees to mutually distrustful components. These guarantees hold until these compartments encounter themselves an undefined behaviour, ensuring that faults are confined to the compartment that triggers them.

[8]: Abate et al. (2018), 'When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise'

They also define $\mathsf{RSC}^{\mathsf{DC}}_{\mathsf{MD}}$, a slight variation on the RSP criterion from chapter 2, and they show that it implies RSCC. In addition, they demonstrate that $\mathsf{RSC}^{\mathsf{DC}}_{\mathsf{MD}}$ can be enforced and proved in practice, by, in a proof assistant, building a compiler for a small toy language that has all the core features of C-like languages, and proving it satisfies $\mathsf{RSC}^{\mathsf{DC}}_{\mathsf{MD}}$. The work of Abate et al. [8] is a major cornerstone for the work described in this thesis, as it provides us with the criterion which we will prove, and it suggests a proof technique that we will improve and reuse.

In this chapter, we focus on explaining the criteria proposed by Abate et al. [8] (section 3.1), and we examine their proof technique (section 3.2).

Author contribution This chapter is focused on explaining background knowledge upon which we build the rest of this thesis. The explanations and examples in both these sections are slight adaptations from the original paper [8], as well as my own paper [2].

[2]: **Thibault** et al. (2024), 'SECOMP: Formally Secure Compilation of Compartmentalized C Programs' **[Co-authored]**

3.1 Background: Robustly Safe Compartmentalizing Compilation

All variants of C compiler correctness, including the most compositional ones [37, 38], completely give up on the whole program after it encounters undefined behaviour. To mitigate this issue, Abate et al. [8] propose a secure compilation notion that restricts the scope of undefined behaviour to the compartments that encounter it. When a compartment encounters an undefined behaviour, it becomes compromised and can start behaving arbitrarily. However, this arbitrary behaviour only occurs inside this compartment; the undefined behaviour does not automatically propagate to other compartments. What the compromised compartments *can do* is try to influence other compartments into triggering undefined behaviour via some controlled interactions respecting their interfaces and the other abstractions of the source language (e.g., the stack discipline on calls and returns).

[37]: Stewart et al. (2015), 'Compositional CompCert'
[38]: Koenig et al. (2021), 'CompCertO: compiling certified open C components'
[8]: Abate et al. (2018), 'When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise'

The set of compromised compartments is not fixed, and instead it is built iteratively as compartments encounter undefined behaviour during the execution. Indeed, it is unrealistic in practice to know in advance which compartments will end up compromised by undefined behaviour. The scope of undefined behaviour is also restricted temporally: compartments still get protection up to the point in time when the undefined behaviour is effectively triggered, for instance by an attacker who manages to exploit the interfaces.

Abate et al. [8] formalize this intuition as an iterative game in which at each step some (initially empty) set of compartments CK is already compromised and tries to attack the remaining uncompromised compartments $K \setminus CK$, for some set of compartment identifiers K, defined in the original compartmentalized program with global interface I, which allows explaining the target behaviour of a compartmentalized whole program.

In each step, the uncompromised programs are put together in a source program $\mathbf{P}$ that is then compiled and linked to an arbitrary target context $\mathsf{C_T}$ that represents the compromised compartments. Then, they explain the execution as one of two possibilities:

1. either it can be explained in terms of source execution: there's a source context $\mathbf{C_S}$ that, when linked with the program, can produce the full execution;
2. or, it can be explained as a source execution with a source context $\mathbf{C_S}$ that ends with undefined behaviour in a compartment of $\mathbf{P}$. Then, one can replace this compartment with a compromised compartment (essentially moving this compartment from $\mathbf{P}$ to $\mathsf{C_T}$) and restart the explanation.

This can be made formal with $\mathsf{RSC}^{\mathsf{DC}}_{\mathsf{MD}}$:

$$\begin{aligned}
&\forall\, K \subseteq \mathit{CompIds}.\ \forall\, I : \mathit{Interface}(K).\ \forall\, CK \subseteq K.\\
&\forall\, \mathsf{C_T} : \lfloor I \rfloor_{CK}.\ \forall\, \mathbf{P} : \lfloor I \rfloor_{K \setminus CK}.\ \forall\, m \not\in \mathsf{Undef}.\\
&\quad (\mathsf{C_T} \bowtie \mathbf{P}{\downarrow}) \rightsquigarrow m.\\
&\qquad \exists\, \mathbf{C_S} : \lfloor I \rfloor_{CK}.\ \exists\, m'.\\
&\qquad\quad (\mathbf{C_S} \bowtie \mathbf{P}) \rightsquigarrow m' \wedge m' \preceq_{K \setminus CK} m.
\end{aligned}$$

To give some more intuition, let us look at an example from [8], in Figure 3.1.

[8]: Abate et al. (2018), 'When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise'

```
component C0 imports E.read, E.write, C2.init, C1.parse,
↪  C2.process;
C0 main() {
  C2.init();
  x := E.read();
  y := C1.parse(x);
  data := C2.process(y);
  if C0.valid(data) { E.write(<data,x>) }
}
C0 valid(data) { ... }

C1 exports parse;
C1 parse(x) { ... }  // (V1) can yield Undef for some x

C2 exports init, process;
C2  init() { ... }
C2  process(y) {
  C2.prepare();        // (V2) can yield Undef if not initialized
  return C2.handle(y); // (V3) can yield Undef for some y
}
C2 prepare() { ... }
C2 handle(y) { ... }
```

Figure 3.1: A compartmentalized application

In this example, $\mathtt{C_0}$ reads a value on the standard input (`E.read`), calls $\mathtt{C_1}$`.parse` to parse it and $\mathtt{C_2}$`.process` to process it, and if the result is valid, outputs it on the standard output (`E.write`).

Importantly, the example compartmentalized application could admit some vulnerabilities:

(V1) `parse` can trigger undefined behaviour if it receives some malformed inputs;
(V2) `prepare` triggers undefined behaviour if `init` hasn't been called beforehand;
(V3) `handle` can trigger undefined behaviour, for instance integer overflow, with some inputs.

After compiling this example application, and executing it, one could obtain the following execution:

$$\{\mathbf{C_1}, \mathbf{C_2}, \mathbf{C_3}\}\downarrow \rightsquigarrow m = \begin{array}{l}\mathtt{C_0.main(); C_2.init(); Ret; E.read; Ret(x); C_1.parse(x)}\\ \mathtt{Ret(y); C_2.process(y); Ret(d);}\\ \mathtt{C_0.valid(d); Ret(true); E.write(< d, x >).}\end{array}$$

The properties of [8] guarantee that such an execution can be explained in terms of source executions and a series of compromises.

[8]: Abate et al. (2018), 'When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise'

We start from no compartment being compromised: ($CK = \emptyset$) and $\mathbf{P} = \{\mathbf{C_1}, \mathbf{C_2}, \mathbf{C_3}\}$. Executing $\mathbf{P}$ leads to the following execution:

$$\mathbf{P} \rightsquigarrow m_1 = \begin{array}{l}\mathtt{C_0.main(); C_2.init(); Ret; E.read; Ret(x); C_1.parse(x);}\\ \mathtt{Undef(C_1)}\end{array}$$

In this source execution, $\mathtt{C_1}$ triggers an undefined behaviour. We add $\mathtt{C_1}$ to the set CK, and $\mathsf{RSC}^{\mathsf{DC}}_{\mathsf{MD}}$ allows us to replace it with some compromised compartment in the source, $\mathtt{A_1}$.

Then, the source execution can go for a bit longer before triggering another undefined behaviour:

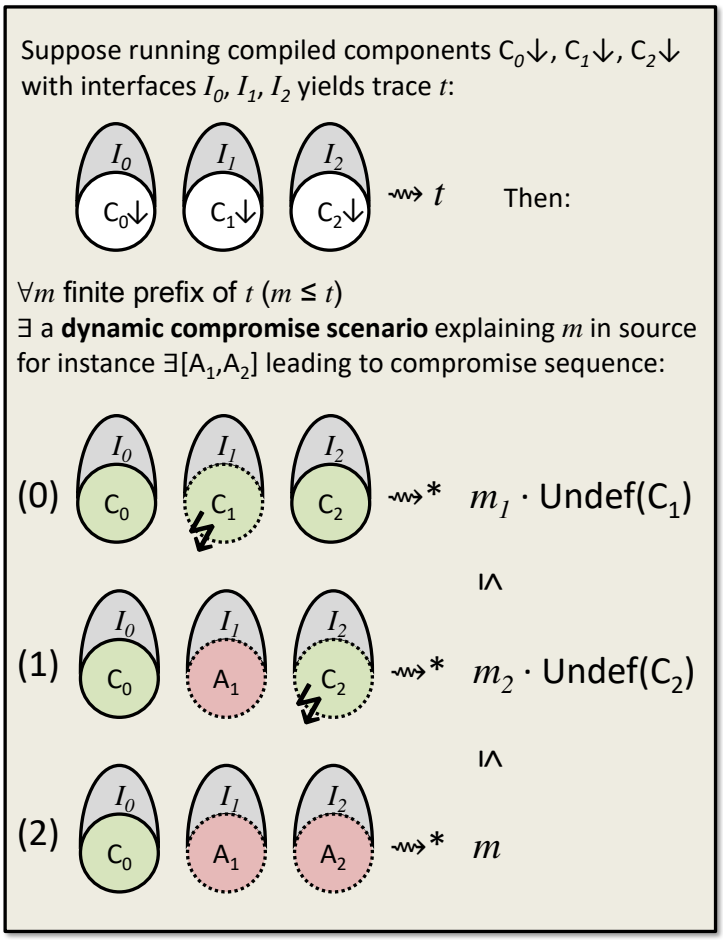
$$\{\mathbf{C_0}, \mathbf{A_1}, \mathbf{C_2}\} \rightsquigarrow m_2 = \begin{array}{l}\mathtt{C_0.main(); C_2.init(); Ret; E.read; Ret(x); C_1.parse(x);}\\ \mathtt{Ret(y); C_2.process(y);}\\ \mathtt{Undef(C_2)}\end{array}$$

This time, $\mathtt{C_2}$ gets compromised. It is added to CK and by $\mathsf{RSC}^{\mathsf{DC}}_{\mathsf{MD}}$ one can finally obtain the full trace, explained as $\mathtt{C_0}$ linked with two adversarial source compartments:

$$\{\mathbf{C_0}, \mathbf{A_1}, \mathbf{A_2}\} \rightsquigarrow m = \begin{array}{l}\mathtt{C_0.main(); C_2.init(); Ret; E.read; Ret(x); C_1.parse(x)}\\ \mathtt{Ret(y); C_2.process(y); Ret(d);}\\ \mathtt{C_0.valid(d); Ret(true); E.write(< d, x >).}\end{array}$$

The entire execution can be summarized by Figure 3.2.

Now that we understand how $\mathsf{RSC}^{\mathsf{DC}}_{\mathsf{MD}}$ works, we must answer one question: how can we implement a compiler that satisfies $\mathsf{RSC}^{\mathsf{DC}}_{\mathsf{MD}}$, and prove it?



The trace prefixes m, m_1, m_2 might, for instance, be:

$$
\begin{aligned}
m &= \texttt{C}_0\texttt{.main}();\texttt{C}_2\texttt{.init}();\texttt{Ret};\texttt{E.read};\texttt{Ret(x)};\texttt{C}_1\texttt{.parse(x)}\\
&\quad\ \texttt{Ret(y)};\texttt{C}_2\texttt{.process(y)};\texttt{Ret(d)};\\
&\quad\ \texttt{C}_0\texttt{.valid(d)};\texttt{Ret(true)};\texttt{E.write}(<\texttt{d},\texttt{x}>)\\
m_1 &= \texttt{C}_0\texttt{.main}();\texttt{C}_2\texttt{.init}();\texttt{Ret};\texttt{E.read};\texttt{Ret(x)};\texttt{C}_1\texttt{.parse(x)};\\
m_2 &= \texttt{C}_0\texttt{.main}();\texttt{C}_2\texttt{.init}();\texttt{Ret};\texttt{E.read};\texttt{Ret(x)};\texttt{C}_1\texttt{.parse(x)};\\
&\quad\ \texttt{Ret(y)};\texttt{C}_2\texttt{.process(y)};
\end{aligned}
$$

Figure 3.2: The dynamic compromise game for the previous compartmentalized application. All compartments start being uncompromised (in green) and are incrementally replaced with arbitrary compartments (in red).

3.2 Background: A Proof Technique for $\mathsf{RSC}^{\mathsf{DC}}_{\mathsf{MD}}$

In this section we briefly review the $\mathsf{RSC}^{\mathsf{DC}}_{\mathsf{MD}}$ secure compilation criterion of Abate et al. [8] as well as their high-level proof structure for this criterion, since we will make use of it in the rest of this thesis. Because secure compilation proofs are notoriously difficult and often include steps analogous to proving compiler correctness, and in order to reduce the amount of proof work in a proof assistant, Abate et al. [8] structure their proof in a way that allows reusing existing correctness proofs as much as possible. More specifically, because CompCert already exists [22], they want to reuse CompCert-style correctness properties. We warm up by reviewing these compiler correctness properties.

[8]: Abate et al. (2018), 'When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise'

[22]: Leroy (2009), 'A Formally Verified Compiler Back-end'

For this both the source language and the target language are given trace-producing semantics, in the style of CompCert. CompCert traces are composed of events recording the calls the whole program makes to system calls performing IO and the results they return. A special event $\mathsf{Undef}(\mathsf{C}_k)$ terminates the trace if undefined behaviour is encountered by compartment C_k. These traces are further extended to record cross-compartment calls and returns, as depicted in Figure 3.3.

$$
\begin{aligned}
a :=\ & \ldots \\
& \mid \texttt{Event_call cp cp.id v} \\
& \mid \texttt{Event_return cp cp v}
\end{aligned}
$$

Figure 3.3: Trace events recording cross-compartment calls and returns

Because the criterion of Abate et al. [8] focuses on safety properties [24], only finite prefixes of traces are considered. We write $\mathbf{W_S} \rightsquigarrow m$ when the whole source program $\mathbf{W_S}$ can produce the finite trace prefix m; and analogously $\mathsf{W_T} \rightsquigarrow m$ when the whole target program $\mathsf{W_T}$ can produce m.

[24]: Lamport et al. (1984), 'Formal Foundation for Specification and Verification'

The compiler correctness guarantee of CompCert states that if a compiled whole program can produce a trace prefix m (i.e., $\mathbf{W_S}{\downarrow} \rightsquigarrow m$) then the original source program can produce a related trace $m' \preceq m$ (i.e., $\mathbf{W_S} \rightsquigarrow m'$), where the relation $m' \preceq m$ is defined as $m' = m$ when $\mathsf{Undef} \notin m'$, and as $m'_0 \cdot m_1 = m$ when $m' = m'_0 \cdot \mathsf{Undef}(\mathsf{C}_k)$ for some C_k. Here, "$\cdot$" denotes concatenation and m_1 is a completely arbitrary trace suffix that the correctly compiled program is allowed to produce when the source program encounters undefined behaviour, which can lead to security vulnerabilities.

Instead of compiling only whole programs though, Abate et al. [8] assume separate compilation—as proposed by Kang et al. [39] and implemented in CompCert since version 2.7—and separately compile a source program $\mathbf{P}$ and a context $\mathbf{C}$, which are intuitively both formed of linked compartments, and which can be linked together to produce a whole program both before compilation using source linking ($\bowtie$), and after compilation using target linking ($\bowtie$). Using these concepts one can define the correctness of a compiler $\downarrow$ like CompCert or our variant SECOMP as follows:

[39]: Kang et al. (2016), 'Lightweight verification of separate compilation'

Definition 3.2.1 (Backward Compiler Correctness (BCC)) *A compiler* $\downarrow$ *satisfies* backward compiler correctness *when:*

$$\forall\, \mathbf{C}\, \mathbf{P}\, m.\ (\mathbf{C}{\downarrow} \bowtie \mathbf{P}{\downarrow}) \rightsquigarrow m \implies \exists\, m'.\ (\mathbf{C} \bowtie \mathbf{P}) \rightsquigarrow m' \wedge m' \preceq m$$

In CompCert this backward compiler correctness (BCC) definition is proved by forward simulations [22] so one also obtains a forward compiler correctness (FCC) result, that the $\mathsf{RSC}^{\mathsf{DC}}_{\mathsf{MD}}$ proof structure of Abate et al. [8] also makes use of. Here, instead of obtaining a related trace prefix they instead assume that the prefix one starts from in the source does not end with undefined behaviour:

[22]: Leroy (2009), 'A Formally Verified Compiler Back-end'

[8]: Abate et al. (2018), 'When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise'

Definition 3.2.2 (Forward Compiler Correctness (FCC) [8]) *A compiler* $\downarrow$ *satisfies* forward compiler correctness *when:*

$$\forall\, \mathbf{C}\, \mathbf{P}.\ \forall\, m \not\in \mathsf{Undef}.\ (\mathbf{C} \bowtie \mathbf{P}) \rightsquigarrow m \implies (\mathbf{C}{\downarrow} \bowtie \mathbf{P}{\downarrow}) \rightsquigarrow m$$

Now, recall the property $\mathsf{RSC}^{\mathsf{DC}}_{\mathsf{MD}}$ that is used as the basic block for formalizing RSCC. compromised and tries to attack the remaining uncompromised compartments $K \backslash CK$, for some set of compartment identifiers K with global interface I, capturing all procedure imports and exports. Uncompromised compartments are linked together into a source program $\mathbf{P}$ with interface $\lfloor I \rfloor_{K \backslash CK}$, and then $\mathbf{P}$ is compiled and linked with a target context $\mathsf{C_T}$, which puts together the compromised compartments and which has interface $\lfloor I \rfloor_{CK}$.

Definition 3.2.3 *A compilation chain satisfies* Robustly Safe Compilation with Dynamic Compromise and Mutual Distrust ($\mathsf{RSC}^{\mathsf{DC}}_{\mathsf{MD}}$) *if there exists a back-translation function* $\uparrow$ *that takes an interface* I*, a target execution* p *producing a finite trace prefix* m*, and a compartment identifier* k*, and generates a source compartment such that*

$$\begin{array}{l}
\forall\, K \subseteq \mathit{CompIds}.\ \forall\, I : \mathit{Interface}(K).\ \forall\, CK \subseteq K. \\
\forall\, \mathsf{C_T} : \lfloor I \rfloor_{CK}.\ \forall\, \mathbf{P} : \lfloor I \rfloor_{K \backslash CK}.\ \forall\, m \not\in \mathsf{Undef}. \\
\quad \forall\, \mathsf{p} : (\mathsf{C_T} \bowtie \mathbf{P}{\downarrow}) \rightsquigarrow m. \\
\qquad \exists\, \mathbf{C_S} : \lfloor I \rfloor_{CK}.\ \mathbf{C_S} = \bowtie_{k \in CK} (I, \mathsf{p}, k)\uparrow \wedge \\
\qquad\quad \exists\, m'.\ (\mathbf{C_S} \bowtie \mathbf{P}) \rightsquigarrow m' \wedge m' \preceq_{K \backslash CK} m
\end{array}$$

The premise states that the compound program $\mathsf{C_T} \bowtie \mathbf{P}{\downarrow}$ has an execution p in the target language producing a trace prefix m, which does not end with an undefined behaviour event (i.e. for a trace $m \cdot \mathsf{Undef}(k')$ one looks only at the prefix m). The conclusion makes a step towards providing an explanation for m with respect to the source language semantics. For this it calls the back-translation function $\uparrow$ on each of the compromised compartments $k \in CK$ and it links together the generated source compartments to obtain a source context $\mathbf{C_S}$ with interface $\lfloor I \rfloor_{CK}$. The $\mathsf{RSC}^{\mathsf{DC}}_{\mathsf{MD}}$ property says that the obtained context $\mathbf{C_S}$ linked with the original source program $\mathbf{P}$ can produce a trace m' that is related to m by the formula $m' \preceq_{K \backslash CK} m$. This is a variant of the $\preceq$ relation from BCC that ensures that $m' = m$ when $\mathsf{Undef} \notin m'$, and that $m'_0 \cdot m_1 = m$ when $m' = m'_0 \cdot \mathsf{Undef}(k)$ for some *uncompromised* compartment $k \in K \setminus CK$.

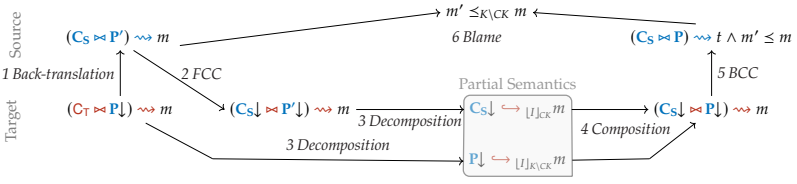


Figure 3.4: The high-level proof structure for $\mathsf{RSC}^{\mathsf{DC}}_{\mathsf{MD}}$ of Abate et al. [8]

For this, Abate et al. [8] propose the high-level proof structure from Figure 3.4 that involves compiler correctness (the FCC and BCC properties above) and four additional security-related properties: back-translation,

decomposition, composition, and blame. The high-level proof starts by back-translating a global interface I and a target execution p producing a trace prefix m repeatedly to generate each of the compartments $k \in K$ of a whole source program producing m.

Definition 3.2.4 (Back-translation) *There exists a function* $\uparrow$ *s.t.*

$$\begin{aligned}
&\forall K.\, \forall I.\, \forall \mathsf{W_T} : I.\, \forall CK \subseteq K.\, \forall m \not\ni \mathsf{Undef}.\\
&\forall \mathsf{p} : \mathsf{W_T} \rightsquigarrow m.\\
&\quad \exists \mathbf{C_S} : \lfloor I \rfloor_{CK}.\, \exists \mathbf{P'} : \lfloor I \rfloor_{K \setminus CK}.\\
&\qquad \mathbf{C_S} \bowtie \mathbf{P'} = \bowtie_{k \in K} (I, \mathsf{p}, k)\uparrow \wedge (\mathbf{C_S} \bowtie \mathbf{P'}) \rightsquigarrow m
\end{aligned}$$

Using the back-translation function $\uparrow$ to generate a *whole* source program $\mathbf{C_S} \bowtie \mathbf{P'}$ not only allows the conclusion of Definition 3.2.4 to be stated in terms of the usual operational semantics of whole programs ($\mathbf{C_S} \bowtie \mathbf{P'} \rightsquigarrow m$), but also allows Abate et al. [8] to compile this whole program and make use of FCC in step 2 from Figure 3.4 to obtain that ($\mathbf{C_S}{\downarrow} \bowtie \mathbf{P'}{\downarrow}) \rightsquigarrow m$.

Then, steps 3 and 4 involve the use of partial semantics, and much like in the proof in section 2.2, involve two lemmas: decomposition and composition, to relate whole program executions and partial program executions in the partial semantics. In contrast with this previous proof, composition and decomposition are only used at the target level: the goal is to start from two executions at the target level producing the same trace prefix, decomposing them into four partial semantics executions, and taking two of them (one from each whole program execution) and compose them together to obtain a new execution producing the same trace prefix.

Partial semantics are written $\hookrightarrow_{\lfloor I \rfloor_{CK}}$ for partial semantics being given to the compromised part, i.e., partial programs made of compartments from CK, and $\hookrightarrow_{\lfloor I \rfloor_{K \setminus CK}}$ for the uncompromised part. Step 3 decomposes the two target-level executions, keeping only one partial semantics execution for each: the one corresponding to $\mathbf{P}{\downarrow}$ and the one corresponding to $\mathbf{C_S}{\downarrow}$.

Definition 3.2.5 (Decomposition)

$$\begin{aligned}
&\forall K.\, \forall I.\, \forall CK \subseteq K.\\
&\forall \mathsf{C_T} : \lfloor I \rfloor_{CK}.\, \forall \mathsf{P_T} : \lfloor I \rfloor_{K \setminus CK}.\, \forall m.\\
&\quad (\mathsf{C_T} \bowtie \mathsf{P_T}) \rightsquigarrow m \implies \mathsf{C_T} \hookrightarrow_{\lfloor I \rfloor_{CK}} m
\end{aligned}$$

Then, those two are composed in step 4, to obtain a whole program execution of $\mathbf{C_S}{\downarrow} \bowtie \mathbf{P}{\downarrow}$.

Definition 3.2.6 (Composition)

$$\begin{aligned}
&\forall K.\, \forall I.\, \forall CK \subseteq K.\\
&\forall \mathsf{C_T} : \lfloor I \rfloor_{CK}.\, \forall \mathsf{P_T} : \lfloor I \rfloor_{K \setminus CK}.\, \forall m.\\
&\mathsf{C_T} \hookrightarrow_{\lfloor I \rfloor_{CK}} m \wedge \mathsf{P_T} \hookrightarrow_{\lfloor I \rfloor_{K \setminus CK}} m \implies\\
&\quad (\mathsf{C_T} \bowtie \mathsf{P_T}) \rightsquigarrow m
\end{aligned}$$

Composition and decomposition capture that the partial semantics is adequate w.r.t. the whole program semantics, in the manner of fully abstract trace semantics [40].

Step 5 uses BCC to turn target execution $(\mathbf{C_S}{\downarrow} \bowtie \mathbf{P}{\downarrow}) \rightsquigarrow m$ back into source execution $(\mathbf{C_S} \bowtie \mathbf{P}) \rightsquigarrow m'$, where the relation $m' \preceq m$ accounts for the possibility of undefined behaviour in $\mathbf{C_S} \bowtie \mathbf{P}$. The context $\mathbf{C_S}$ is, however, generated by the back-translation and has no undefined behaviour along the trace m, which one shows in step 5 of the proof. So if there is an undefined behaviour in m' then this can only be blamed on an as-yet-uncompromised compartment $k \in K \backslash CK$, as required by the conclusion of $\mathsf{RSC}^{\mathsf{DC}}_{\mathsf{MD}}$ from Definition 3.2.3.

Definition 3.2.7 (Blame)

$$\begin{aligned}&\forall K.\, \forall I.\, \forall CK \subseteq K.\, \forall \mathbf{C_S} : \lfloor I \rfloor_{CK}.\, \forall \mathbf{P}, \mathbf{P}' : \lfloor I \rfloor_{K \backslash CK}.\\ &\quad \forall m.\, (\mathbf{C_S} \bowtie \mathbf{P}') \rightsquigarrow m \wedge (\mathbf{C_S} \bowtie \mathbf{P}) \rightsquigarrow m' \wedge m' \preceq m \implies\\ &\qquad m' \preceq_{K \backslash CK} m.\end{aligned}$$

Implementation and Proof

Abate et al. [8] not only provide a proof that this proof technique can be used to achieve $\mathsf{RSC}^{\mathsf{DC}}_{\mathsf{MD}}$, but they also build an example compiler from a simple imperative language to a relatively low-level compartmentalized machine with a RISC-like instruction set. The languages are equipped with small-step operational semantics $\xrightarrow{a}$ and $\xrightarrow{a}$ producing the events. Both languages use a CompCert-style memory model [41], where memory is divided into blocks, each belonging to a single compartment, and where memory cannot be shared. We will not dwell here on the other details on the languages; they can be found in the Rocq development of [8], and are not necessary to understand the rest of this section. We simply describe the two most interesting steps, back-translation and composition/decomposition, as much of the work inside this thesis is aiming at improving those core two steps.

[41]: Leroy et al. (2012), *The CompCert Memory Model, Version 2*

Back-translation Back-translation is defined to satisfy Definition 3.2.4 as a function $\uparrow$ that takes a finite prefix m and an interface I and returns a whole program $\mathbf{W}$ that produces m. The back-translation relies on state: each compartment uses the location `local[0]` to track how many events it has already emitted. When a procedure is called, it increments `local[0]` and emits the n-th event in m, if `local[0]`'s value is n.

For instance, consider the following trace, which is one of the intermediary traces of the compartmentalized application from section 3.1, but adapted to our setting:

$$\begin{aligned}m = {}&\texttt{Event_call}\ \texttt{C}_0\ \texttt{C}_2\texttt{.init}\ 0; \texttt{Event_return}\ \texttt{C}_2\ \texttt{C}_0\ 0;\\ &\texttt{Event_call}\ \texttt{C}_0\ \texttt{C}_1\texttt{.parse}\ 27; \texttt{Event_return}\ \texttt{C}_1\ \texttt{c}_0\ 35;\\ &\texttt{Event_call}\ \texttt{C}_0\ \texttt{C}_2\texttt{.process}\ 35;\end{aligned}$$

The back-translation will need to generate code for each of $\texttt{C}_0$, $\texttt{C}_1$, and $\texttt{C}_2$. Let us look in particular at $\texttt{C}_0$. According to the trace, it starts by calling $\texttt{C}_2$`.init` with argument 0, so the code for $\texttt{C}_0$ will start with

```
1  C0 main() {
2    local[0]++;
3    if (local[0] == 1) {
4      C2.init();
5      C0.main();
6    } else { ... }
7  }
```

The program inspects the local buffer, finds it is now equal to 1, and according to that emits the first event, the call to C_2.init. When the compartment regains control after the call to C_2, it calls itself recursively: the counter will be incremented again, and so we will enter another branch.

Similarly, the next event coming from C_0 is Event_call C_0 C_1.parse 27; and later comes a call to C_2.process. This means we back-translation of C_0 will look like the previous one but with new branches:

```
C0 main() {
  local[0]++;
  if (local[0] == 1) {
    C2.init();
    C0.main();
  } else if (local[0] == 2) {
    C1.parse(27);
    C0.main();
  } else if (local[0] == 3) {
    C2.process(35);
    C0.main();
  }
}
```

Two things are interesting: first, the structure is completely linear, and the compartment is not able to divert from it, so it will always only generate the input trace. Then, because of this, it doesn't need to process return values: here, it completely ignores the value returned from C_1.parse, because we can hard-code the actual values it will pass as argument according to what's recorded on the trace.

To prove back-translation, Abate et al. [8] rely on the fact the trace m has been produced by a program, so it has to satisfy some well-formedness conditions, the most important of which being well-bracketedness: each return matches a previous call appearing in the trace. The proof is essentially a proof by simulation, as it proceeds by induction on the trace left to produce, knowing the history of events that have already been emitted.

Partial Semantics: Decomposition and Composition Decomposition and composition are defined using partial semantics. Abate et al. [8]'s partial semantics are defined in a generic way using the whole program semantics $\xrightarrow{a}$. To do so, they introduce a *partialization function* par that takes a complete state cs and an interface $\lfloor I \rfloor_{CK}$ and removes all information about the compartments from CK from the complete state. For instance, are removed from the partial states the memory and the stack frames from CK.

The step relation for the partial semantics, denoted $\mathsf{ps} \xrightarrow{a}_{\lfloor I \rfloor_{CK}} \mathsf{ps}'$ is given by the following inference rule:

$$\frac{\mathsf{cs} \xrightarrow{a} \mathsf{cs}' \qquad \mathrm{par}(\mathsf{cs}, \lfloor I \rfloor_{CK}) = \mathsf{ps} \qquad \mathrm{par}(\mathsf{cs}', \lfloor I \rfloor_{CK}) = \mathsf{ps}'}{\mathsf{cs} \xrightarrow{a}_{\lfloor I \rfloor_{CK}} \mathsf{cs}'}$$

A step can occur between two partial states ps and ps′ whenever a step can occur between two complete states cs and cs′ that partialize to ps and ps′. Essentially, this definition abstracts away parts of the state and replaces them with non-determinism.

Proving decomposition is direct from this definition. However, proving composition is not. The formal proof of composition ends up being 6000 lines of code long, one of the largest parts of the proof. This bodes ill when it comes to scaling up to realistic languages: in a more realistic compilation chain such as the one in SECOMP (chapter 5), the target language will be even more complex (more than ten times the amount of instructions), so we'd expect the proof to become gigantic. Thankfully, we will see in the next chapter (chapter 4) that composition and decomposition can be replaced by a single step that simplifies the proof significantly.

4 Recomposition

We can now present the first contribution of this thesis: **recomposition**. Recomposition is a single theorem that replaces both decomposition and composition used in the proof of Abate et al. [8], discarding partial semantics completely. The benefit of discarding partial semantics is huge. First, partial semantics are often hard to define properly, especially for realistic languages. For instance, mainline CompCert doesn't have partial semantics and entire papers have been dedicated to the sole goal of getting partial semantics for CompCert [38]. Then, the proofs of composition and decomposition were massive, and it was unclear whether one would be able to extend them to a non-toy language in a reasonable amount of time. Finally, in this proof, partial semantics were only used as a proof device; there is no good reason to keep them if a simpler technique allows discarding them.

[8]: Abate et al. (2018), 'When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise'

[38]: Koenig et al. (2021), 'CompCertO: compiling certified open C components'

Recomposition starts from two executions of whole programs (with the same interface) producing the same trace, and requires proving that one can take the compromised components from one execution and the uncompromised ones from the other execution, link them together, and obtain an execution producing the same trace.

Definition 4.0.1 (Recomposition)

$$\begin{aligned}&\forall\, K.\ \forall\, I.\ \forall\, CK \subseteq K.\\ &\forall\, \mathsf{C_T}\ \mathsf{C'_T} : \lfloor I \rfloor_{CK}.\ \forall\, \mathsf{P_T}\ \mathsf{P'_T} : \lfloor I \rfloor_{K \setminus CK}.\ \forall m.\\ &\quad (\mathsf{C_T} \bowtie \mathsf{P_T}) \rightsquigarrow m \wedge (\mathsf{C'_T} \bowtie \mathsf{P'_T}) \rightsquigarrow m \implies (\mathsf{C'_T} \bowtie \mathsf{P_T}) \rightsquigarrow m\end{aligned}$$

Recomposition can be used to replace composition and decomposition, obtaining the proof technique depicted in Figure 4.1. The proof proceeds essentially the same, starting with back-translation, then applying FCC, then recomposition, then BCC and finally using blame.

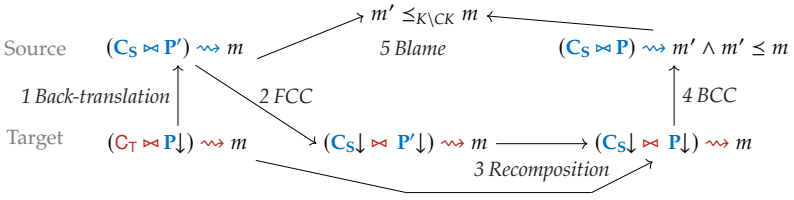


Figure 4.1: Updated proof technique using recomposition

In this chapter, we describe how we implemented recomposition in the context of [8] and replaced the use of composition and decomposition with it [7]. While composition alone took almost 6000 lines of code to prove, recomposition only takes 2000, achieving our goals of simplifying the proofs and eliminating the partial semantics.

[7]: Abate et al. (2019), *When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise* **[Co-authored]** (arXiv version with recomposition)

Author contribution The work on recomposition started in 2019, shortly after composition and decomposition's proofs were finished for [8]. I, the author of this thesis, developed recomposition together with some other authors of [7, 8]. I played a major role in the design of the proof, and the implementation of the mergeability relation and the merge function, and I wrote a major portion of the proof.

[7]: Abate et al. (2019), *When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise* **[Co-authored]** (arXiv version with recomposition)
[8]: Abate et al. (2018), 'When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise'

4.1 Proving Recomposition

Key ideas We prove recomposition by using a *three-way simulation* stated in terms of the underlying small-step semantics. The proof relies on two components: a *mergeability* relation $\sim$ between complete states s and s'', as well as a merge function, merge, that merges two complete states into a single one.[1]

1: We use zero prime symbols for states of $C_T \bowtie P_T$, two prime symbols for states of $C'_T \bowtie P'_T$, and one prime symbol for those of $C'_T \bowtie P_T$.

The idea is to define the mergeability relation $\sim$ such that it is preserved by "parallel executions":

$$\forall\, s_1\, s_2\, s''_1\, s''_2.\; s_1 \sim s''_1 \implies s_1 \xrightarrow{m}{}^{*} s_2 \wedge s''_1 \xrightarrow{m}{}^{*} s''_2 \implies s_2 \sim s''_2$$

If one starts from two related states $s_1 \sim s''_1$, then it should be the case that after taking some steps and producing the same trace prefix, the two resulting states are again related.

We must then define the merge function, parametrized by the interfaces of the program part $\lfloor I \rfloor_{K \setminus CK}$ and of the context $\lfloor I \rfloor_{CK}$. Here, our goal is to define merge so that, given one state s of $C_T \bowtie P_T$ and one state s'' of $C'_T \bowtie P'_T$, $\mathrm{merge}(s, s'') = s'$ is a state of $C'_T \bowtie P_T$. We also want merge to be defined whenever the states are mergeable according to the mergeability relation, and to commute with transitions, in the following sense: if

$$s_1 \sim s''_1,\; s_1 \xrightarrow{m}{}^{*} s_2 \text{ and } s''_1 \xrightarrow{m}{}^{*} s''_2$$

then

$$\mathrm{merge}(s_1, s''_1) \xrightarrow{m}{}^{*} \mathrm{merge}(s_2, s''_2).$$

It is clear that if one is able to define merge and $\sim$ properly, then the recomposition theorem follows directly, simply by proving that initial states are related, and then applying the property above.

Implementing recomposition for our target language In our implementation, the mergeability relation is given an extensional definition:

Definition 4.1.1 (Mergeability relation) *Let s and s'' be two target states. We say s and s'' are* mergeable, *written $s \sim s''$ whenever there exists a trace prefix m such that:*

$$\mathrm{ini}(C_T \bowtie P_T) \xrightarrow{m}{}^{*} s \textit{ and } \mathrm{ini}(C'_T \bowtie P'_T) \xrightarrow{m}{}^{*} s''$$

where ini *is the function constructing the initial state of the program passed as argument.*

This definition trivially implies the first property from above, but has the drawback of being too extensional; we can work around that because our setting is simple enough that we can easily retrieve from this definition an important property[2]: the shapes of the stacks in related states are similar, in that they both record the same sequence of calls.

2: By induction on the trace.

The merge function is more interesting: to merge two states, one has to:

1. find out which execution is currently in control of the run;
2. keep the PC and the other registers from the corresponding state;

3. merge the memories;
4. recombine the stacks, by interleaving frames from C'_T and from P_T while preserving the structure of the original stacks when it comes to cross-compartment calls.

For finding out which execution is in control, it's enough to look at where the PC is pointing.

To merge two memories M and M'', we split each of them into two "partial memories": $M_{|C_T}$ containing the memory blocks from M belonging to compartments from C_T, $M_{|P_T}$ containing the memory blocks from M belonging to compartments from P_T, $M''_{|C'_T}$ containing the memory blocks from M'' belonging to compartments from C'_T, and $M''_{|P'_T}$ containing the memory blocks from M'' belonging to compartments from P'_T. We then merge them, keeping the partial memories of C'_T and P_T:

$$M' = M''_{|C'_T} \cup M_{|P_T}.$$

This is easy, because in our setting a memory is a partial map $M : (\mathit{CompartmentID} * \mathit{BlockID} * \mathit{Offset}) \to \mathit{Value}$, so one simply has to filter by compartment identifier (the first argument).

Finally, to merge two stacks, we look at the top frame of each of the two stacks, observe that they must belong to the same compartment (this is the property we get from the mergeability relation), and keep the one that corresponds to the execution where we keep that compartment.

Armed with these two definitions, we can prove the second property. At each step, the execution is guided by either both programs or both contexts. The two runs synchronize on observable events, which is made possible by the fact that they produce the same trace. This is characterized by the two lemmas. The first one states that taking a silent step in only one of the executions implies taking a step in the merged execution:

Lemma 4.1.1 (Silent step implies silent step in the merged execution) *If*

$$s_1 \sim s''_1 \text{ and } s_1 \to s_2$$

then

$$\mathrm{merge}(s_1, s''_1) \to \mathrm{merge}(s_2, s''_1).$$

The second one states that the two executions producing events in lockstep implies that the merge execution also produces the same event.

Lemma 4.1.2 (Steps producing an event in lockstep) *If*

$$s_1 \sim s''_1, s_1 \xrightarrow{e} s_2, \text{ and } s''_1 \xrightarrow{e} s''_2,$$

then

$$\mathrm{merge}(s_1, s''_1) \xrightarrow{e} \mathrm{merge}(s_2, s''_2).$$

With all of this, we just need to iterate these lemmas and use some symmetry results to finally obtain the recomposition lemma (Definition 4.0.1).

4.2 Discussion

These new definitions are more natural, completely symmetric, and easier to work with than the previous ones about partial semantics [8]. Moreover, using the complete semantics allows us to reuse theorems already proved for these semantics. As a result, the resulting proofs are fully mechanized, at least three times smaller than the original proofs (recomposition is around 2000 lines; meanwhile composition alone was around 6000 lines), and several times faster to check.

[8]: Abate et al. (2018), 'When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise'

It is not surprising that recomposition is easier to prove than composition and decomposition. Indeed, composition and decomposition together directly imply recomposition. This doesn't mean that all uses of composition and decomposition in other works are superfluous. Indeed, many proofs, including the one from section 2.2, use composition and decomposition at two different levels (composition in the source and decomposition in the target). Using recomposition in such proofs would require restructuring the proof entirely, which can be very inconvenient.

Recomposition and fully abstract trace semantics There are two key intuitions behind recomposition: (1) because of *determinacy*, the internal behaviour of a compartment only depends on its internal state, and on all the information it received from other compartments; (2) our trace model is informative enough to capture all information that is exchanged between compartments. This intuition is shared with many other proofs of secure compilation. Often, these results are split into separate composition and decomposition (section 2.2, [6, 40, 42]), and for full abstraction the two are often called soundness and completeness of trace equivalence, which together imply the full abstraction of a trace semantics [43–46].

[6]: Abate et al. (2019), 'Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation' **[Co-authored]**
[40]: Jeffrey et al. (2005), 'Java JR: Fully Abstract Trace Semantics for a Core Java Language'
[42]: Juglaret et al. (2016), 'Beyond Good and Evil: Formalizing the Security Guarantees of Compartmentalizing Compilation'
[43]: Patrignani et al. (2015), 'Fully abstract trace semantics for protected module architectures'
[44]: Patrignani et al. (2015), 'Secure Compilation to Protected Module Architectures'
[45]: El-Korashy et al. (2021), 'CapablePtrs: Securely Compiling Partial Programs Using the Pointers-as-Capabilities Principle'
[46]: Patrignani et al. (2017), 'Secure Compilation and Hyperproperty Preservation'

This intuition suggests that proving recomposition would be impossible in settings where information can leak via side channels. In addition, it suggests that recomposition is *too strong* of a property if, as we are, one is only interested in proving *safety*. We have yet to find a replacement for recomposition that would not imply that the trace semantics are "fully abstract", and that could be used even in the presence of covert communication between the program and the context.

Regarding the absence of the compiler in the recomposition lemma We are able to prove a generalized lemma for arbitrary target programs and contexts in our case, because our enforcement is done at the target level by the abstract machine.

We suspect that it is not possible in the general setting. Indeed, if the compiler plays a bigger role for enforcing security than in our setting, one likely needs to use the fact that the programs are compiled programs. For example, the compiler might be inserting protections in the compiled code at boundaries between components that ensure cross-component calls are well-behaved. In such a case, a recomposition proof would need to also consider the compiler.

Shortcomings of the current proof The acute reader will have noticed two major flaws that prevent applying the proof described above to a more realistic setting.

First, our characterization of mergeable states is very extensional: because two states being "mergeable" only means they are the result of two executions with the same trace prefix, we are forced to rederive more intensional properties of these states in every proof. This doesn't become a problem in our case, because the language is small and simple to reason about; however, when it comes to more realistic languages, doing so might become unwieldy. We would much rather have an *intensional* definition: one that explains exactly what are the properties of the states in question, so that we can use them directly in our proofs.

Second, we are able to define a merge function that way because the language is very simple and designed in a way that makes it easy to merge parts of the state. For instance, the memory can be conveniently split into one partial memory by compartment: a memory is a partial map $\mathsf{M} : (\mathit{CompartmentID} * \mathit{BlockID} * \mathit{Offset}) \to \mathit{Value}$, which makes filtering by compartment easy. In the setting of SECOMP (chapter 5 and chapter 6), the memory is a map $\mathsf{M} : (\mathit{BlockID} * \mathit{Offset}) \to \mathit{Value}$ where some additional information is attached to each block; splitting and merging such a memory is much more difficult, because the allocation pattern of each compartment will influence which block identifier gets assigned to each compartment, and this *will* change even in executions producing the same trace. Hence, in the general setting, one probably wouldn't use a merge function but rather prove that several invariants are preserved.

In section 6.2, we propose a solution to these issues by defining a generic notion of *three-way recomposition simulation*, and generic simulation diagrams that can be reused in a variety of settings to prove recomposition.

SECOMP: A Secure Compiler From Compartmentalized C to RISC-V Assembly

Compiling Compartments from C Down to Assembly

5

In this chapter, we can finally discuss the core contribution of this thesis: SECOMP [2], a secure compilation chain for a compartmentalized variant of C.

[2]: **Thibault** et al. (2024), 'SECOMP: Formally Secure Compilation of Compartmentalized C Programs' **[Co-authored]**

SECOMP allows the programmer to write compartmentalized C programs, where compartments are protected from each other, following the principles discussed in chapter 3. To build SECOMP, we extend the CompCert compiler and its correctness proof to handle isolated compartments that interact only via procedure calls and returns. Although compiler correctness by itself is definitely not enough to prove secure compilation, since it gives up on programs with undefined behaviour, we use it as one key ingredient for such a proof, adopting the high-level proof structure proposed by Abate et al. [8] and described in chapter 3 and chapter 4. We show compiler correctness, as well as the three security-related properties: *back-translation*, *recomposition*, and *blame*. Proving these properties at scale and achieving formally secure compilation for a compiler for a mainstream programming language were open research challenges.

[8]: Abate et al. (2018), 'When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise'

Concretely, we extend the syntax and semantics of all languages of CompCert to deal with compartments, and we adapt all compilation passes and their proofs. This is particularly interesting for the target language, RISC-V, as we need to introduce a novel *shadow stack* that keeps track of calls and returns, in order to characterize the enforcement mechanism. We also develop the security proof for SECOMP, starting at Clight and going all the way down to RISC-V, reusing the proof technique discussed in the previous chapters, but with major adaptations to allow the proof of each step to scale to our setting.

In this chapter, we start by explaining the compartment model that we adopt in SECOMP (section 5.1). We then discuss how we update all languages and passes of CompCert to use compartments (section 5.2), including RISC-V and its novel use of a shadow stack. We conclude by discussing the proof of correctness (section 5.3), which is one of the essential steps of the proof technique we use for security.

The proof of security itself is discussed in the next chapter, chapter 6.

Author contribution I, the author of this thesis, was the lead author of the SECOMP paper [2]. As the lead author of the paper, I participated in most discussions regarding the design of SECOMP, its implementation, and its proof of correctness.

[2]: **Thibault** et al. (2024), 'SECOMP: Formally Secure Compilation of Compartmentalized C Programs' **[Co-authored]**

5.1 Compartment Model

As discussed previously, compartmentalization allows developers to structure large programs into mutually distrustful compartments that have limited privileges, are isolated from each other, and can only interact in a controlled way. For building SECOMP, we adopt a model that statically partitions C programs into compartments. Every C definition of a procedure or a global variable belongs to a single compartment.

Any block of memory belongs to the compartment that allocated it and compartments do not share memory: each block can only be accessed

by the code of the compartment it belongs to.[1] Instead, all interactions between compartments must happen via *cross-compartment calls and returns* that respect the *interfaces* provided by the programmers: each compartment `C` comes with a set of exported procedure declarations (i.e. which procedures it makes available to other compartments), written `C.exports`, and a set of imported procedure declarations (i.e. which procedures it uses from other compartments), written `C.imports`.

1: In chapter 7, we describe an extension of Abate et al. [8] that incorporates memory sharing; because of the complexity of that extension, we choose not to have it in SECOMP.

These interfaces are specified locally with special syntax: `C exports f` means compartment `C` makes `f` available to other compartments; and `C1 imports C2.f` means compartment `C1` imports procedure `f` from compartment `C2`.

Compartments must respect these compartment interfaces; otherwise, they trigger undefined behaviour, whose scope is restricted by our secure compilation property to just the offending compartment. Moreover, in our model, compartments can only pass each other scalar values as procedure call arguments and return values. We introduce this last restriction for two reasons: first, a compartment cannot use pointers from another compartment, which defeats the purpose of passing pointers in most cases. More importantly, passing pointers to other compartments would require recording these pointers on the trace, which would significantly complicate back-translation, recomposition, blame, and, if also done for pointers to dynamically allocated memory, compiler correctness.

The compartments also interact with an external environment using *system calls*.[2] These are special, privileged procedures whose semantics is axiomatized in CompCert and that may generate some events. Examples of these system calls include volatile memory operations, calls to the heap allocator, or input and output (e.g., reading from the console). The system calls do not belong to a real compartment; instead calling them is considered a special kind of internal call that can only change the calling compartment's memory. Following the principle of least privilege, by default compartments do not have access to system calls beyond safe operations like memory allocation and freeing. Instead, the interfaces specify, for each compartment, which system calls they are allowed to use, using syntax `C imports_syscall ef`, meaning compartment `C` imports the system call `ef`.

2: Our "system calls" correspond in CompCert to the "external" functions that do not get resolved to actual C source code during linking. Such functions are implemented in lower-level, trusted libraries (like libc) and may include operating system calls.

Finally, our compartments form a simple lattice with a $\bot$ and a $\top$ element, where inclusion is denoted $\subseteq$, as depicted in Figure 5.1. $\bot$ and $\top$ are special

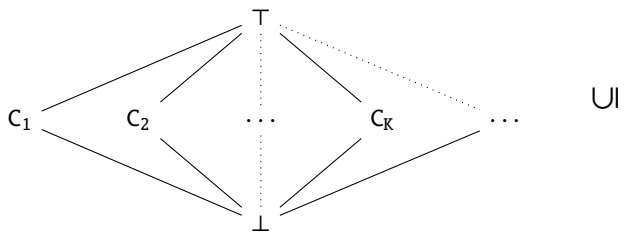


Figure 5.1: Lattice of compartments

logical compartments that play a special role in our implementation: anyone can call and access elements of compartment $\bot$, and $\top$ is used to parametrize operations of the compartmentalization mechanism. For instance, calls to $\bot$ always pass the compartment checks, and $\bot$ is the logical compartment of system calls, so we can treat them the same as standard calls;[3] and some privileged loads or stores that are performed by the compartmentalization mechanism itself are parametrized by $\top$.

3: System calls can still fail if they are not in the interface of the calling compartment, but this failure happens in the semantics of the system calls.

On the other hand, all "concrete" compartments $\mathtt{C}_1$, etc. have the same level of privileges, and are completely independent from each other on this lattice.

5.2 Adding Compartments to CompCert

Following the ideas above, we extend all of CompCert's 10 languages, from C to RISC-V assembly, adding syntax describing the compartment breakup and interfaces and semantic checks to ensure all compartments respect these interfaces. As explained above, failing a check triggers undefined behaviour for the offending compartment, which is represented by the execution becoming stuck.

5.2.1 Memory Model

We reuse the block-based memory model of CompCert and extend it with compartments. A memory `m` is essentially a map from pairs of block identifier and offsets to values, along with some metadata and well-formedness conditions. To this metadata, we add a map from blocks to compartments: each memory block belongs to a single compartment that is assigned at allocation and cannot be changed during the execution after allocation.

Memory operations (reads, writes, frees) are parameterized by the compartment performing the operation, and fail when this compartment does not own the targeted block.

5.2.2 Call and Return Events

To prove secure compilation, just like in section 2.2, we also extend the CompCert trace model with two new events: `Event_call` and `Event_return`. These events are generated by cross-compartment calls and returns (or the equivalent jumps in RISC-V assembly), and record enough information on the traces to be able to prove recomposition and back-translation.

The SECOMP events are described in Figure 5.2.[4] The new events are highlighted: `Event_call C C'.f args` captures a cross-compartment call passing `args` from compartment `C` to the procedure `f` of compartment `C'`. Similarly, `Event_return C' C v` represents returning value `v` from compartment `C'` to compartment `C`.

4: We write $\overline{x}$ to denote a list of x.

$$
\begin{aligned}
a :=\ & \texttt{Event_syscall name } \overline{\texttt{v}}\ \overline{\texttt{bs}}\ \texttt{v}\ \overline{\texttt{bs}} \\
\mid\ & \texttt{Event_vload m ch id ofs v} \\
\mid\ & \texttt{Event_vstore m ch id ofs v} \\
\mid\ & \texttt{Event_annot string } \overline{\texttt{v}} \\
\mid\ & \texttt{Event_call cp cp.id } \overline{\texttt{v}} \\
\mid\ & \texttt{Event_return cp cp v}
\end{aligned}
$$

Figure 5.2: SECOMP events

5.2.3 Compartments at Higher Levels

For all languages except RISC-V, we make use of the existing structure of the semantics to implement these new events, so the only required change is to add the events to the appropriate call and return transitions. We also insert in the semantics dynamic checks to ensure calls and returns conform to the interfaces and do not pass pointers. The dynamic checks act as follows: internal calls (for instance a call from `C.f` to `C.g`) are always allowed. System calls are allowed only if the interface allows it. Lastly, a cross-compartment call from `C.f` to `C'.g` is allowed if

1. `g` ∈ `C'.exports`;
2. `C'.g` ∈ `C.imports`; and
3. all of the arguments are scalar values.

Items 1. and 2. are implemented by the following Rocq predicates:

```
1  Definition allowed_cross_call (ge: t) (cp: compartment) (vf: val) :=
2    match vf with
3    | Vptr b _ =>
4      exists i cp',
5      invert_symbol ge b = Some i /\
6      find_comp_in_genv ge vf = cp' /\
7      In i (interface_export ge.(genv_interface) cp')
8      In (cp', i) (interface_import ge.(genv_interface) cp) /\
9    | _ => False
10   end.
11
12 Definition allowed_call (ge: t) (cp: compartment) (vf: val) :=
13   (find_comp_in_genv ge vf ⊆ cp) \/
14   allowed_cross_call ge cp vf.
```

`allowed_cross_cal` says that `cp` can call the function located at address `v` if and only if `v` is a pointer pointing at a function `i` (line 5), belonging to compartment `cp'` (line 6), that this function is exported by `cp'` (line 7) and imported by `cp` (line 8). Then, the allowed calls are these allowed cross-compartment calls, or the internal ones.

Then, one can look at the rules for calls in each program semantics, for instance for Cminor:

```
1    | step_call: forall f optid sig a bl k sp e m vf vargs fd t,
2        eval_expr sp e m (comp_of f) a vf ->
3        eval_exprlist sp e m (comp_of f) bl vargs ->
4        Genv.find_funct ge vf = Some fd ->
5        funsig fd = sig ->
6        (* Check that the call to the function pointer is allowed *)
7        forall (ALLOWED: Genv.allowed_call ge (comp_of f) vf),
8        forall (NO_CROSS_PTR:
9          Genv.type_of_call (comp_of f) (comp_of fd) = CrossCompartmentCall
           ↪ ->
10         Forall not_ptr vargs),
11       forall (EV: call_trace ge (comp_of f) (comp_of fd) vf vargs sig t),
12       step (State f (Scall optid sig a bl) k sp e m)
13         t (Callstate fd vargs (Kcall optid f sp e k) m (comp_of f))
```

This rule is used for a call from `f` to the function pointed to by `vf`, passing arguments `vargs`. The new conditions are the named ones, from line 7 to 11. The first one is that the call respects the interfaces or is internal, using the previous predicate. Lines 8 to 10 require that if the call crosses compartment boundaries, then no pointer argument is passed. Last, line 11 generates an event (recorded inside `t`) when it's a cross-compartment call.

We refer the interested reader to the Rocq development for more details on the implementation of each predicate.

In all languages before RISC-V control-flow well-bracketedness is ensured by the semantics. What this means is that we do not need to do any interface check when returning: it's impossible to return to the wrong compartment, as calling a `return` instruction uses an abstract stack (or a continuation) to know where to return. However, we still have to check that the returned value is a scalar.

Interestingly, we observe that since calls and returns should not be disturbed by optimization, we must disallow cross-compartment tail-call optimization and inlining, as those would substantially change the way compartments interact. Indeed, consider a call from `C.f` to `C'.g`. Inlining `g` in `f` would eliminate the cross-compartment call from the trace; similarly, tail-call optimization would require merging two stackframes (the one of `f` and the one of `g`) that belong to two different compartments into a single block, which is impossible. We however keep these optimizations for internal calls.

Lastly, to achieve security, we make all registers be caller saved, since on a cross-compartment call we cannot trust the callee compartment to save and restore the caller's registers.[5] We also made the semantics invalidate non-argument registers on cross-compartment calls and non-return registers on cross-compartment returns (by making them undefined values), since recomposition requires all information passed between compartments to be captured by the trace. A better proof technique that wouldn't use recomposition but a weaker property would allow us to dispense with that change.

5: For simplicity our current implementation does this for all calls, not just cross-compartment ones, but it would be good to improve this aspect in the future.

5.2.4 Compartments at the RISC-V Level

At the RISC-V level, implementing secure compartments requires more care. Without proper protection, adversarial code could make use of the unstructured control-flow inherent to the assembly language to break the compartment abstraction. For instance, an attacker could try to jump to code to which it shouldn't have access.

We modify CompCert's RISC-V assembly semantics to prevent this kind of attack, by protecting the compartment abstraction and interfaces. To do so, we observe that calls and returns are only implemented by the compiler using specific instructions—jump-and-link instructions $\texttt{Pjal}_\texttt{s}$ and $\texttt{Pjal}_\texttt{r}$ for calls, and indirect jump instruction $\texttt{Pj}_\texttt{r}$ for returns—so we forbid all other instructions from changing compartments. At the same time, we add a boolean flag to these instructions to mark those that could potentially be used to attempt cross-compartment calls or returns. When this flag is `true`, the instruction can be used to attempt cross-compartment calls or returns, but otherwise instructions are not allowed to change the current compartment. The reason for tagging instructions that way is to give more flexibility for potential implementations of the back-end: with such a tag, one does not necessarily have to perform potentially complex interface checks every time, which should help with performance.

Most importantly, our RISC-V semantics makes use of a *cross-compartment shadow stack*, which is a list of records, each containing a return address, a stack pointer, and a procedure signature, and when an execution encounters such a jump and attempts to switch to another compartment, we make use of the interfaces and this newly added shadow stack to decide whether the switch is allowed.

When a call is encountered, there are two cases, whether it is a cross-compartment call or not.

The case of the internal call is represented in Figure 5.3. When encountering such a case, the compartment allocates a new stack frame in the concrete stack, where it stores the old stack pointer and some other data, the return address is written in the `RA` register, and the `SP` register gets a pointer to the top of the new stack frame. The shadow stack is not updated: it is only used for cross-compartment calls and returns.

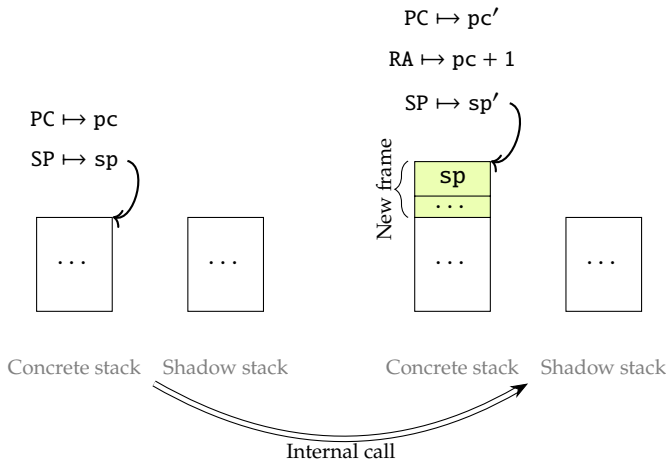


Figure 5.3: Allocation of a new stack frame during an internal call. The shadow stack stays the same.

The case of the cross-compartment call is represented in Figure 5.4. As before, the calling compartment allocates a new stack frame in the concrete stack, where it stores the old stack pointer and some other data, the return address is written in the `RA` register, and the `SP` register gets a pointer to the top of the new stack frame. Then, the interfaces are checked. If the call is allowed, and if the arguments don't contain pointers, a new stack frame is pushed on top of the shadow stack. This stack frame contains the previous stack pointer `sp`, the return address `ra` = `pc` + 1, and the callee's signature `sg`. During this step, all other registers are also cleared, to prevent covert communication between compartments.

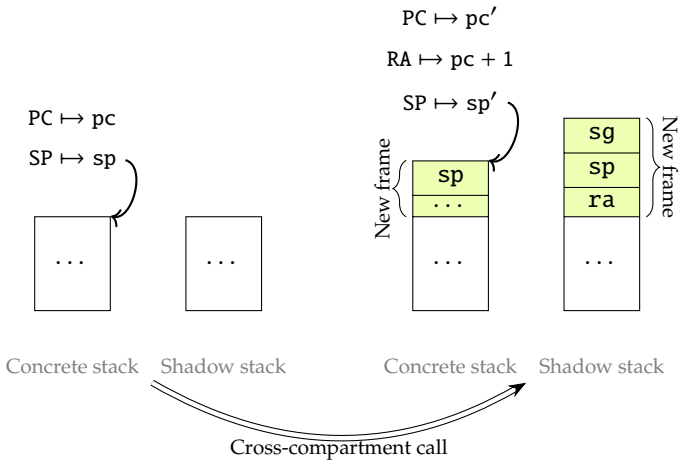


Figure 5.4: Allocation of a new stack frame during a cross-compartment call. The shadow stack gets a new frame.

Returns are the opposite, except that there is no direct way to check that the return follows the interfaces; indeed, there is no guarantee that we are returning from the same function that was called, because of tail-call optimization. Instead, the shadow stack is used. The case of an internal return is simple: we do not perform any check.

The case of a cross-compartment return is very interesting, and depicted in Figure 5.5. The compartment encounters a jump instruction tagged as

a return, that attempts to go to another compartment. Then, we use the shadow stack to perform three checks:

1. the jump targets the right address as recorded on the shadow stack, `ra`
2. the stack pointer register `SP` has been correctly restored: it contains `sp` as recorded on the shadow stack
3. the value returned, which we use the signature `sg` to locate in the registers, is not a pointer.

If all these are true, the return is allowed, the concrete and shadow stack frames are deallocated, and the registers are cleared to avoid covert communication.

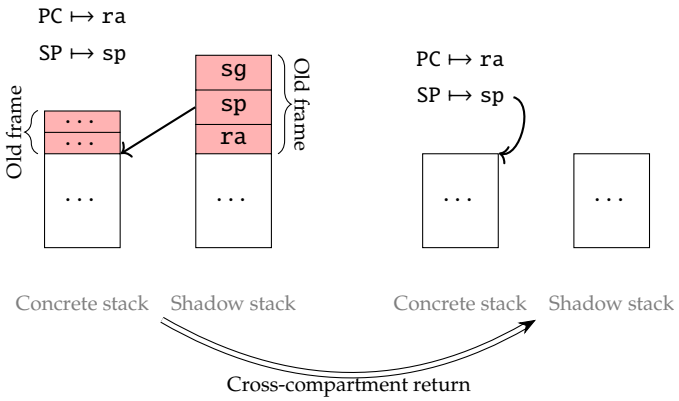


Figure 5.5: Cross-compartment return. After the checks are performed, the top frames are deallocated.

This use of a shadow stack can be seen as abstractly specifying the well-bracketedness [47] of cross-compartment control-flow, and leaves lower-level backends the freedom to enforce this in various ways. Finally, because this shadow stack is not used for calls and returns inside a compartment, backends don't have to do anything special for these.

[47]: Anderson et al. (2023), 'Formalizing Stack Safety as a Security Property'

Additionally, a change we do to the RISC-V semantics for recomposition to hold is to make stack frames used to pass spilled arguments in cross-compartment calls read-only. Indeed, consider what could happen without this protection. A first compartment, $\mathtt{C}_1$, creates a new stack frame $\mathtt{f}_1$ and saves a pointer to it; it then calls $\mathtt{C}_2$ passing enough arguments that some of them are spilled on that stack frame. Then, $\mathtt{C}_2$ calls $\mathtt{C}_1$ back, which is then able to use its saved pointer to the stack frame $\mathtt{f}_1$ to overwrite some arguments. Then, $\mathtt{C}_1$ returns. Now, if $\mathtt{C}_1$ tries accessing an argument on the stack, it will not obtain what was recorded on the trace but what was written by $\mathtt{C}_1$ during the callback. This would break recomposition, hence the need to prevent this kind of attacks.

Making stack frames read-only is our solution: a malicious compartment cannot exploit callbacks to modify the spilled arguments it previously passed anymore.

5.3 Proving Correctness

We update all of CompCert's 19 passes and the simulation proofs showing forward compiler correctness (FCC) (Definition 3.2.2) to account for the addition of compartments and the new trace model. In the setting of CompCert, with determinacy, it implies backward compiler correctness BCC (Definition 3.2.1). The updated proofs mainly rely on the

compartment information being correctly preserved by compilation, e.g. procedures not changing compartment, memory blocks belonging to different compartments not being merged, etc.

Adapting CompCert's compiler correctness Coq proof to account for our changes was a substantial amount of work. We wanted to change the proof as little as possible, in order for the proof to be easier to maintain. But since CompCert is a realistic compiler, it was not always obvious from the start how best to do this. Several times, we made design decisions that seemed adequate, but turned out much later to be inadequate, when we discovered that they interacted poorly with some particular compilation pass (e.g. intra-compartment inlining or tail-call optimization) or language (e.g. RISC-V assembly). These issues often did not affect the correctness of the compiler, but made the proofs much more difficult, so we had to backtrack and find alternative ways to structure the changes so as to keep the proofs simple.

For a concrete example, consider the problem of *when* to check the interface when executing a call. In most of SECOMP's languages, following CompCert's example, calls and returns are done in two steps, using special kinds of states that are called "call-states" and "return-states". An example is depicted in Figure 5.6.

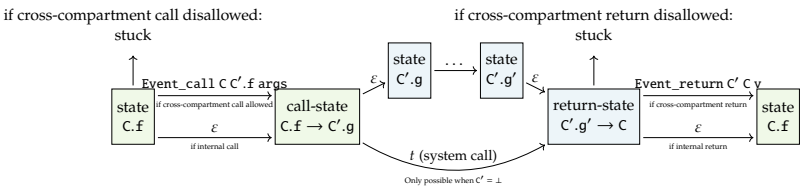


Figure 5.6: Call from compartment `C` (light green states) to compartment `C'` (light blue states), and the corresponding return.

During the execution of a procedure `C.f`, a call instruction to `C'.g` is encountered. The compartmentalized semantics first compare the source and destination of the call. If the call is an internal call (bottom arrow), then the state transitions to a call-state of that compartment without generating an event. If the call is a cross-compartment call, then the semantics perform an additional check: if the call is allowed, then the state transitions to a call-state, but this time generating an event recording the call (top arrow). Otherwise, the execution must stop.

From this call-state, there are two possibilities: If `g` is not a system call, the call-state transitions silently to a standard state of `C'` whose execution continues until reaching a return instruction, upon which the standard state silently transitions to a return-state preparing to return to `C`. If `g` is a system call, then the call-state directly transitions to a return-state preparing to return to `C`, according to the semantics of the system call. Now, from the return-state, and similarly to the transition to the call-state, we must again check whether the return is allowed, and transition to the state of `C` if it is the case. Note that in the example above, it is possible for `C'` to return to `C` from a different procedure `g'`. This may happen when, internally, `g` performs a tail call to `g'`.

The question one could ask is the following: why perform the interface check during the transitions *to* call-states and *from* return-states, instead of during the transitions *from* call-states and *to* return-states? The reason has to do with the structure of the proofs by simulation. Adding the checks at the wrong location would require us to restructure the forward simulation proof from the penultimate language to the last one, RISC-V.

Indeed, the RISC-V semantics are an exception to the structure described above and do not have an explicit call-state, instead merging the two transitions from the upper path into a single one. More explicitly, the existing forward simulation proof matches the first transition to the execution of the jump-and-link, changing the PC to another compartment; and the second transition is "dropped".[6] Hence, given our goal to modify the correctness proofs as little as possible, it was necessary to have the interface check in the first transition rather than the second.

6: CompCert simulations allow relating a source step to zero steps in the target, at the condition that it only happens a finite number of times in a row.

In the end we found elegant ways of adapting CompCert's compiler correctness Coq proof to account for all the changes above and proved FCC and BCC. Yet, while compiler correctness is an important part of our security proof, it is definitely not sufficient by itself. In the next chapter, we will discuss the other components of the proof that together with correctness imply security.

The Secure Compilation Proof of SECOMP

6

With SECOMP, we demonstrate that it is realistic to formally verify a secure compiler for a realistic language. In addition to the proof of correctness that we already discussed in the previous chapter, we also prove all the other components of the proof technique devised by Abate et al. [7], including using recomposition instead of partial semantics.

[7]: Abate et al. (2019), *When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise* **[Co-authored]** (arXiv version with recomposition)

This proof technique relies on several intermediary lemmas, namely *back-translation*, *blame*, and *recomposition*. In this chapter, we will focus our discussion on *back-translation* and *recomposition*.

In section 6.1, we will discuss the *back-translation* step, aiming at proving the following lemma:

Definition 6.0.1 (Back-translation)

$$\begin{aligned}&\forall\, \mathsf{W_T}.\ \mathsf{W_T} \rightsquigarrow m \implies \\ &\quad \exists\, \mathbf{C_S}\ \mathbf{P}.\ (\mathbf{C_S} \bowtie \mathbf{P'}) \rightsquigarrow m\end{aligned}$$

The main idea behind our proof of back-translation is the introduction of slightly more informative events recording so-called *memory deltas*, that we use to reestablish memory injections at key moments during the execution.

Then, in section 6.2, we discuss *recomposition*:

Definition 6.0.2 (Recomposition)

$$\begin{aligned}&\forall\, \mathsf{C_T}\ \mathsf{C'_T}.\ \forall\, \mathsf{P_T}\ \mathsf{P'_T}.\ \forall m. \\ &\quad (\mathsf{C_T} \bowtie \mathsf{P_T}) \rightsquigarrow m \wedge (\mathsf{C'_T} \bowtie \mathsf{P'_T}) \rightsquigarrow m \implies (\mathsf{C'_T} \bowtie \mathsf{P_T}) \rightsquigarrow m\end{aligned}$$

Compared to chapter 4, we simplify our proof considerably by introducing a generic notion of *three-way recomposition simulation*, as well as generic diagrams based on the ideas of having three simulation relations (a strong one, a weak one, and a mixed one), inspired by [3, 45]. We then describe how we instantiate these generic diagrams in our setting.

[3]: El-Korashy et al. (2022), 'SecurePtrs: Proving Secure Compilation with Data-Flow Back-Translation and Turn-Taking Simulation' **[Co-authored]**

[45]: El-Korashy et al. (2021), 'CapablePtrs: Securely Compiling Partial Programs Using the Pointers-as-Capabilities Principle'

Author contribution I, the author of this thesis, was the lead author of the SECOMP paper [2]. As the lead author of the paper, I participated in most discussions regarding the design of the proofs that are discussed here. I was the sole contributor for the proof of recomposition. My contribution to back-translation was purely high-level discussions and I wasn't involved in the technical development, but it is still discussed here for the sake of completeness, and because it's very interesting. I wasn't involved in blame beyond a very high-level view, which explains why it is not discussed here in this thesis.

[2]: **Thibault** et al. (2024), 'SECOMP: Formally Secure Compilation of Compartmentalized C Programs' **[Co-authored]**

6.1 Back-translation

We adapt the back-translation and proof technique described in chapter 3 to our setting, targeting Clight, but to do so, we need to make our events more informative and devise a novel notion of well-formedness

of traces made out of these informative events through an intermediate language.

First, the events we use in SECOMP do not contain enough information to directly convert each trace into a Clight statement. In particular, they do not capture all the information necessary to obtain a back-translated program that produces the same events for system calls:

Unfortunately, among the axioms CompCert gives us, determinacy, which states that executing the system call in the same memory state yields the same result, is not sufficient, because there is indeed little hope of perfectly reproducing the RISC-V memory in Clight.

One could maybe use a technique such as the data-flow back-translation from chapter 7 to obtain a very closely related memory; however, this technique was designed for memory sharing which we explicitly don't support in SECOMP, so it might be too strong for our setting. In addition, it was a very difficult proof even in a simpler setting, and would likely require a lot of work to adapt to CompCert.

Instead, CompCert provides another useful axiom (Axiom 6.1.1): if a system call is executed in some memory $\mathtt{M}_1$ resulting in memory $\mathtt{M}_1'$ while generating a trace, then if executed in some other memory $\mathtt{M}_2$ that $\mathtt{M}_1$ injects into, the same system call with the same arguments results in a memory $\mathtt{M}_2'$ that $\mathtt{M}_1'$ injects into, and crucially generates the same trace.

Axiom 6.1.1 (Commutativity of system calls and memory injections) Given two environments $\mathtt{ge}_1$ and $\mathtt{ge}_2$, two memories $\mathtt{M}_1$ and $\mathtt{M}_2$ and a memory injection j that relates $\mathtt{M}_1$ and $\mathtt{M}_2$ (written $\mathtt{M}_1 \mapsto_j \mathtt{M}_2$) and that relates the symbols of the current compartment in the environments $\mathtt{ge}_1$ and $\mathtt{ge}_2$, then if a system call $\mathtt{ef}$ with arguments $\mathtt{args}$ produces a trace t in $\mathtt{ge}_1$ and $\mathtt{M}_1$, resulting in return value $\mathtt{res}$ and new memory $\mathtt{M}_1'$,

$$\mathtt{ge}_1, \mathtt{M}_1 \vdash \mathtt{ef}(\mathtt{args}) \xrightarrow{t} (\mathtt{res}, \mathtt{M}_1')$$

then:

$$\exists\, j'\ \mathtt{res}'\ \mathtt{M}_2'.\ \mathtt{ge}_2, \mathtt{M}_2 \vdash \mathtt{ef}(\mathtt{args}) \xrightarrow{t} (\mathtt{res}', \mathtt{M}_2')$$

with $\mathtt{M}_1' \mapsto_{j'} \mathtt{M}_2'$.

Yet using this axiom imposes having a memory injection, and while memory injections do not need to be defined on the entire memory, they must be defined at least on the public symbols of the environment (of the calling compartment), which include global buffers. Put simply, this means that the semantics of system calls is allowed to depend on the content of the calling compartment's global buffers, but doesn't need to consider what is not reachable from these global buffers.

This motivates introducing more *informative events* satisfying two requirements: (1) a RISC-V program always produces a well-formed trace of informative events, and (2) each informative event directly translates into a Clight statement. Using these informative events, we define a novel notion of *well-formed informative trace* and a back-translation from such traces to Clight programs, and we significantly extend the technique of Abate et al. [8] to prove the correctness of this back-translation.

[8]: Abate et al. (2018), 'When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise'

Informative events augment each system call event (and also cross-compartment call or return events) with a list of the changes to the global buffers since the last informative event. Each of these changes is called a *memory delta*, written δ, and a list of these is written Δ, and is ordered chronologically. The informative events we use are:

1. $\mathtt{Icall\ f\ t\ g}\ \overline{\mathtt{v}}\ \mathtt{sg}\ \Delta$ represents a cross-compartment call, where $\mathtt{f}$ is the name of the caller, $\mathtt{t}$ is the trace event produced by the call, $\mathtt{g}$ is the name of the callee, $\overline{\mathtt{v}}$ are the arguments, $\mathtt{sg}$ is the signature of the callee, and Δ are the memory deltas.

2. `Ireturn f t v` Δ represents a cross-compartment return, where `f` is the current procedure name, `t` is the trace event produced by the return, `v` is the return value, and Δ are memory deltas.
3. `Isys f t ef` $\overline{\texttt{v}}$ Δ represents a system call, which is similar to the call case except that it requires the system call descriptor `ef` instead of `g` and `sg`.

where Δ represents a list of memory deltas. Formally, the memory deltas are defined as follows:

$$
\begin{aligned}
\delta := \quad & \mid \texttt{delta_storev ch p v C} \\
& \mid \texttt{delta_store ch b o v C} \\
& \mid \texttt{delta_bytes b o } \overline{\texttt{v}} \texttt{ C} \\
& \mid \texttt{delta_alloc C l h} \\
& \mid \texttt{delta_free b l h C}
\end{aligned}
$$

where stores, storevs and bytes are three different kinds of stores, while allocs and frees correspond to the same memory operations.

Our back-translation function uses these deltas to generate Clight code performing the same changes to global buffers before performing the system call, and is defined in the following manner, piecewise: first, the back-translation of memory deltas in compartment `C` $\texttt{bt}_\delta^{\texttt{C}}$ is defined as follows:

$$
\begin{aligned}
\texttt{bt}_\delta^{\texttt{C}}(\texttt{delta_storev ch p v C}) &= \texttt{if symbol_of(p)} = \texttt{id} \wedge \\
&\qquad \texttt{public}^{\texttt{C}}(\texttt{id}) \wedge \texttt{not_pointer(v)} \\
&\quad \texttt{then p} := \texttt{v else skip} \\
\texttt{bt}_\delta^{\texttt{C}}(\texttt{delta_store ch b o v C}) &= \texttt{if symbol_of(b)} = \texttt{id} \wedge \\
&\qquad \texttt{public}^{\texttt{C}}(\texttt{id}) \wedge \texttt{not_pointer(v)} \\
&\quad \texttt{then id[o]} := \texttt{v else skip} \\
\texttt{bt}_\delta^{\texttt{C}}(\texttt{delta_bytes b o } \overline{\texttt{v}} \texttt{ C}) &= \texttt{if symbol_of(b)} = \texttt{id} \wedge \\
&\qquad \texttt{public}^{\texttt{C}}(\texttt{id}) \wedge \texttt{not_pointer}(\overline{\texttt{v}}) \\
&\quad \texttt{then id} := \overline{\texttt{v}} \texttt{ else skip} \\
\texttt{bt}_\delta^{\texttt{C}}(\texttt{delta_alloc C l h}) &= \texttt{skip} \\
\texttt{bt}_\delta^{\texttt{C}}(\texttt{delta_free b l h C}) &= \texttt{skip}
\end{aligned}
$$

where `symbol_of` returns the identifier of its argument, if it exists, $\texttt{public}^{\texttt{C}}(\texttt{id})$ is `true` if and only if `id` is an identifier corresponding to a public global variable of `C`, and `not_pointer(v)` is `true` if and only if `v` is not a pointer.

We define the back-translation of a list of memory deltas in `C`, $\texttt{bt}_\Delta^{\texttt{C}}$ as follows:

$$
\begin{aligned}
\texttt{bt}_\Delta^{\texttt{C}}([]) &= \texttt{skip} \\
\texttt{bt}_\Delta^{\texttt{C}}(\delta :: \Delta) &= \texttt{bt}_\delta^{\texttt{C}}(\delta) \texttt{ ;; } \texttt{bt}_\Delta^{\texttt{C}}(\Delta)
\end{aligned}
$$

Note that we discard the stores to public global variables that are not scalar; to do so, we restrict the semantics so that the public symbols only contain scalars when doing system calls that generate events. Then, our well-formedness conditions (described later) ensure that the last store made to a global variable before issuing an event is necessarily that of a scalar.

We define the back-translation of events in compartment `C`, $\texttt{bt}_\texttt{e}^{\texttt{C}}$ as fol-

lows:

$$\begin{aligned}
\mathtt{bt}_e^{\mathtt{C}}(\mathtt{Icall\ f\ t\ g\ \overline{v}\ sg}\ \Delta) &= \mathtt{bt}_\Delta^{\mathtt{C}}(\Delta)\ ;;\ \mathtt{g}(\overline{\mathtt{v}}) \\
\mathtt{bt}_e^{\mathtt{C}}(\mathtt{Ireturn\ f\ t\ v}\ \Delta) &= \mathtt{bt}_\Delta^{\mathtt{C}}(\Delta)\ ;;\ \mathtt{return\ v} \\
\mathtt{bt}_e^{\mathtt{C}}(\mathtt{Isys\ f\ t\ ef\ \overline{v}}\ \Delta) &= \mathtt{bt}_\Delta^{\mathtt{C}}(\Delta)\ ;;\ \mathtt{ef}(\overline{\mathtt{v}})
\end{aligned}$$

We can finally define, for a given compartment C, the back-translation of a list of events from that compartment, $\mathtt{bt}^{\mathtt{C}}$ in the following manner:

$$\begin{aligned}
\mathtt{bt}_n^{\mathtt{C}}([]) &= \mathtt{return} \\
\mathtt{bt}_n^{\mathtt{C}}(\mathtt{e} :: \mathtt{ls}) &= \mathtt{if\ C.ctr} = \mathtt{n} \\
&\qquad \mathtt{then\ C.ctr} + + ;;\ \mathtt{bt}_e^{\mathtt{C}}(\mathtt{e}) \\
&\qquad \mathtt{else\ bt}_{n+1}^{\mathtt{C}}(\mathtt{ls}) \\
\mathtt{bt}^{\mathtt{C}} &= \mathtt{bt}_0^{\mathtt{C}}
\end{aligned}$$

where C.ctr is a counter local to compartment C. Then, each back-translated procedure is defined by wrapping this in a `while true` loop. One can recognize the structure of the back-translation described in chapter 3, albeit slightly modified.

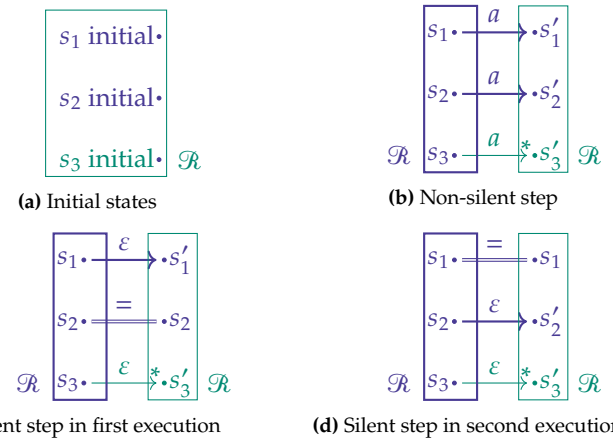
To help with the proof, we define a novel notion of well-formedness of a trace of informative events. To do so, we define a new intermediate language[1] with a semantics that characterizes the well-formedness of such traces. Informally, informative traces can be seen as the programs of this language, and the step relation executes these traces by emitting the first informative event of the trace and updating the state according to the event. More precisely, in this language, states s are triples that record the currently executing procedure, a memory, and a cross-compartment call stack; the step relation $s \xrightarrow{\alpha} s'$ relates the states s and s' and produces an informative event α. The rules of the step relation additionally record the conditions necessary for the back-translated code to be proved correct. In particular, if $s \xrightarrow{\alpha} s'$, then applying the deltas Δ from α to the memory M of s produces the memory M′ of s' (i.e. `mem_delta_apply` Δ M = M′); and the public symbols only contain scalars before system calls.

1: We see our back-translation function as a compiler from traces to Cminor programs.

We say that a trace of informative events is *well-formed* when it is produced by the reflexive transitive closure of this step relation. We prove that for any trace prefix m produced by a RISC-V program, there exists a well-formed informative trace $\overline{\alpha}$ that projects to m by removing the additional data recorded in informative events. This allows us to then completely forget about the RISC-V semantics when proving the correctness of this back-translation.

6.2 Recomposition

As discussed previously, recomposition is essential for the proof technique, as it allows to replace the arbitrary RISC-V context with which we started, with a context that was obtained by back-translation and then compiled. More concretely, starting from two whole target programs $W_1 = C_1 \bowtie P_1$ and $W_2 = C_2 \bowtie P_2$ that can execute to produce the same trace m and that are each split into a *program side* (P_1 respectively P_2) and a *context side* (C_1 respectively C_2), recomposition gives us that the whole program $W_3 = C_1 \bowtie P_2$ produces the same trace m.



(a) Initial states **(b)** Non-silent step

(c) Silent step in first execution **(d)** Silent step in second execution

Figure 6.1: Three-way simulation properties represented graphically

The intuitions behind this kind of proofs are as follows: (1) because of determinacy the internal behaviour of a compartment only depends on its internal state, and on all the information it received from other compartments, or from system calls; (2) our extended trace model is informative enough to capture all information that is exchanged between compartments or obtained from system calls. Because of this, we can relate the execution of W_3 to that of W_1 when executing in a compartment of the context, and to that of W_2 otherwise: the internal states of the compartments of the context part in the execution of W_3 and that of W_1 agree, and the same holds for W_3, W_2, and the internal states of the compartments in the program part. This is preserved by silent steps, because they only depend on that same internal state; and because the trace events record every information exchanged, even when switching side, non-silent steps also preserve this information.

6.2.1 Simulation Diagrams for Recomposition

Formalizing this idea in order to prove recomposition for our RISC-V semantics extended with compartments is highly complex, as it relies on many low-level details of the RISC-V semantics. For this reason, we propose a generic proof technique that elegantly splits recomposition into several self-contained parts. Our technique introduces eight novel CompCert-like simulation diagrams (described in detail in chapter 4) that provide a structured way to think and reason about recomposition, explicitly separating the definition of invariants, the reasoning about internal steps, and the reasoning about events and cross-compartment communication. Together, the diagrams imply the existence of a novel *three-way recomposition simulation* that itself implies the recomposition theorem (Definition 6.0.2).

We define three-way recomposition simulation in the generic setting of labeled transition systems $(S, \rightarrow)$ with *initial* states:

Definition 6.2.1 (Three-way recomposition simulation) *Given three labeled transition systems $L_1 = (S_1, \rightarrow_1)$, $L_2 = (S_2, \rightarrow_2)$ and $L_3 = (S_3, \rightarrow_3)$, we say there exists a three-way recomposition simulation between L_1, L_2, and L_3 when there exists a relation $\mathcal{R}$ between states of L_1, L_2, and L_3 that satisfies the properties depicted in Figure 6.1.*

The Rocq definition, in the style of CompCert simulations, is also given in Figure 6.2.

```
Record tsim_properties (L1 L2 L3: semantics) {...}
    (match_states: state L1 -> state L2 -> state L3 -> Prop) :=
  { tsim_match_initial_states:
      forall s1 s2, initial_state L1 s1 ->
                initial_state L2 s2 ->
      exists s3, initial_state L3 s3 /\
                 match_states s1 s2 s3;
    tsim_simulation_simultaneous:
    forall s1 e s1',
      Step L1 s1 (e :: nil) s1' ->
    forall s2 s2', Step L2 s2 (e :: nil) s2' ->
      forall s3, match_states s1 s2 s3 ->
      exists s3',
         (Plus L3 s3 (e :: nil) s3')
         /\ match_states s1' s2' s3';
    tsim_simulation1:
      forall s1 s1', Step L1 s1 E0 s1' ->
      forall s2 s3, match_states s1 s2 s3 ->
      exists s3',
         Star L3 s3 E0 s3'
      /\ match_states s1' s2 s3';
    tsim_simulation2:
      forall s2 s2', Step L2 s2 E0 s2' ->
      forall s1 s3, match_states s1 s2 s3 ->
       exists s3',
         Star L3 s3 E0 s3'
      /\ match_states s1 s2' s3';
    tsim_public_preserved:
      forall id, Senv.public_symbol (symbolenv L3) id =
        Senv.public_symbol (symbolenv L1) id }.
```

Figure 6.2: Three-way simulation in Rocq

For each property from Figure 6.1, each row represents one of the 3 executions. Arrows represent execution steps, and are annotated with either an event a or silence ε. We denote the reflexive transitive closure of the step relation with $\rightarrow^*$. We use thick, dark purple for the assumptions, and dark green for the conclusions we have to prove (including the existence of states). We denote equality via a double line. We represent the simulation relation as a rectangle around the states it relates.

Property (a) corresponds to `tsim_match_initial_states`, and states that the simulation relation $\mathscr{R}$ is compatible with initial states.

Property (b) corresponds to `tsim_simulation_simultaneous`, and states that whenever the first two executions take a step from related states producing the same observable event a, then so can the third one.

Properties (c) and (d) correspond to `tsim_simulation1` and `tsim_simulation2`, and state that silent steps in either of the first two executions preserve the simulation relation (and the third execution is allowed to take some silent steps too).

We prove that, given such a three-way simulation between the semantics of programs W_1, W_2, and W_3, then $W_1 \rightsquigarrow m \wedge W_2 \rightsquigarrow m \implies W_3 \rightsquigarrow m$ (which directly implies recomposition). For this it is enough to follow both executions in W_1 and W_2, and to apply the appropriate property, (2), (3), or (4) until all of m is produced.[2]

To define simpler to prove simulation diagrams, we will instantiate the relation $\mathscr{R}$ above with the conjunction of three relations, following ideas of El-Korashy et al. [3, 45]: a *strong* relation $\sim$, a *weak* relation $\equiv$, and a *mixed* relation $\mathscr{M}$. Intuitively, the strong relation $\sim$ relates the internal

2: CompCert experts may note that we do not use a notion of decreasing measure or a notion of final states, since we are only concerned with finite execution prefixes.

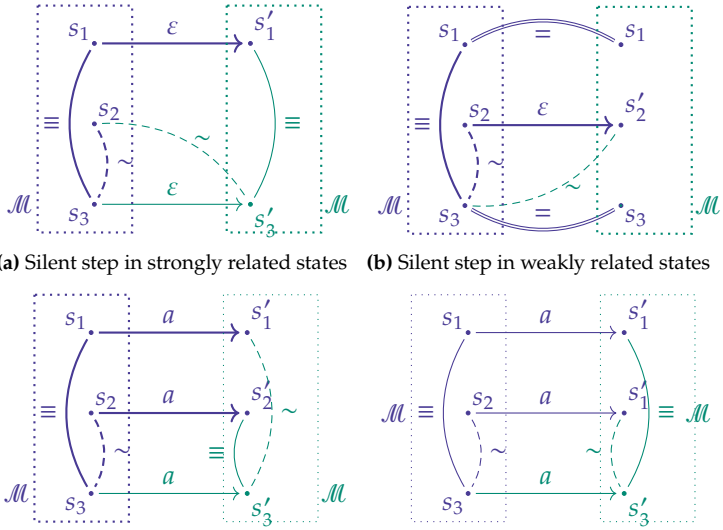
[3]: El-Korashy et al. (2022), 'SecurePtrs: Proving Secure Compilation with Data-Flow Back-Translation and Turn-Taking Simulation' **[Co-authored]**

[45]: El-Korashy et al. (2021), 'CapablePtrs: Securely Compiling Partial Programs Using the Pointers-as-Capabilities Principle'

state of the side being executed, the weak relation ≡ relates the internal state of the side not being executed, and the mixed relation $\mathcal{M}$ relates the parts shared between compartments (such as the stack). Given three states s_1, s_2, s_3 that are executing a compartment that's taken from W_1, the strong relation ≡ relates s_1 to s_3, and the weak relation ∼ relates s_2 to s_3. Symmetrically, when the current compartment is taken from W_2 then $s_1 \sim s_3$ and $s_2 \equiv s_3$. When switching to a compartment taken from the other side the relations are switched as well. These three relations are also parametrized by "metadata", which is allowed to change during the execution, and that we use to capture the relation between memories with memory injections.

In our Rocq development, we call the *strong relation* `strong_equivalence`, the *weak relation* `weak_equivalence`, and the *mixed relation* `common_equivalence`.

From this, we deduce 8 diagrams, 4 of them being depicted in Figure 6.3. The other 4 are symmetric cases that involve switching the role of the first and second executions.



(a) Silent step in strongly related states

(b) Silent step in weakly related states

(c) Non-silent step with swapping relations

(d) Step with an event without swapping relations

Figure 6.3: Recomposition diagrams

As previously, in each diagram, each row represents one of the 3 executions. Arrows represent execution steps, and are annotated with either an event a or silence ε. We use thick, dark purple for the assumptions, and dark green for the conclusions we have to prove (including existence of states). We depict in dashed line the weak relation, and in plain line the strong relation; and we use dotted rectangles to depict the mixed relation. We denote equality via a double line in Figure 6.3b.

The first diagram, Figure 6.3a, corresponds to the following Rocq code:

```
Definition step_silent_strong1:
  forall s1 s1', Step L1 s1 E0 s1' ->
  forall s2 s3 M, strong_equivalence M s1 s3 ->
           weak_equivalence M s2 s3 ->
       common_equivalence M s1 s2 s3 ->
  exists s3' M', Plus L3 s3 E0 s3' /\
           strong_equivalence M' s1' s3' /\
           weak_equivalence M' s2 s3' /\
              common_equivalence M' s1' s2 s3'.
```

and describes the case where s_1 and s_3 are taking a silent step synchronously and can be read as follows: starting from three states s_1, s_2,

and s_3 related by $\mathscr{M}$, such that $s_1 \equiv s_3$ and $s_2 \sim s_3$, and such that s_1 steps silently to s_1', we have to prove that s_3 also steps silently to another state s_3', that the weak and strong relation are reestablished for s_3' (i.e. $s_1' \equiv s_3'$ and $s_2 \sim s_3'$), and that the mixed relation is also reestablished.

Similarly, Figure 6.3b describes the case where s_2 takes a silent step; because it is only weakly related to s_3, we do not require either s_1 or s_3 to take steps as well, but we must still reestablish the relations. It corresponds to the following definition:

```
Definition step_silent_weak1:
  forall s1 s1', Step L1 s1 E0 s1' ->
  forall s2 s3 M, strong_equivalence M s2 s3 ->
           weak_equivalence M s1 s3 ->
           common_equivalence M s1 s2 s3 ->
  exists M', strong_equivalence M' s2 s3 /\
        weak_equivalence M' s1' s3 /\
        common_equivalence M' s1' s2 s3.
```

The last two diagrams correspond to the case where both executions produce an event, and represent the following predicate:

```
Definition step_event:
  forall s1 e s1', Step L1 s1 (e :: nil) s1' ->
  forall s2 s2',   Step L2 s2 (e :: nil) s2' ->
  forall s3 M, match_states M s1 s2 s3    ->
  exists s3' M', Plus L3 s3 (e :: nil) s3' /\
            match_states M' s1' s2' s3'.
```

where `match_states` is defined as:

```
Variant match_states: metadata -> state L1 -> state L2 -> state L3
↪  -> Prop :=
  | match_states_left: forall M s1 s2 s3,
      common_equivalence M s1 s2 s3 ->
      strong_equivalence M s1 s3 ->
      weak_equivalence M s2 s3 ->
      match_states M s1 s2 s3
  | match_states_right: forall M s1 s2 s3,
      common_equivalence M s1 s2 s3 ->
      weak_equivalence M s1 s3 ->
      strong_equivalence M s2 s3 ->
      match_states M s1 s2 s3.
```

This definition captures the two possible cases: for Figure 6.3c and Figure 6.3d, both executions produce an event a. Given $s_1 \equiv s_3$ and $s_2 \sim s_3$, related by $\mathscr{M}$, such that $s_1 \xrightarrow{a} s_1'$ and $s_2 \xrightarrow{a} s_2'$, one must prove the existence of s_3' such that $s_3 \xrightarrow{a} s_3'$, i.e. that the three executions advance in lockstep. If the event constitutes a change of control between the two sides (program and context) weak and strong relations are swapped as illustrated in the first diagram; otherwise, the relations aren't swapped, as represented by the second diagram.

We prove that our 8 diagrams together imply the existence of the above three-way simulation, and hence imply the recomposition theorem. To do so, we simply instantiate $\mathscr{R}$ with the `match_states` predicate.

6.2.2 Instantiating the Diagrams

How do we instantiate these relations in our setting? To answer this question, we must first describe with a few more details the RISC-V language we are working with.

At the RISC-V level, states are the disjoint union of regular states $s = (regs, \mathtt{M}, st)$ and return-states $rs = (regs, \mathtt{M}, st, \mathtt{sg}, \mathtt{C})$ where *regs* is a register set, `M` is a memory, *st* a shadow stack, `sg` a signature, and `C` a compartment name recording which compartment the execution is returning from (Figure 6.4).

```
Inductive state: Type :=
| State: stack -> regset -> mem -> compartment -> state
| ReturnState: stack -> regset -> mem ->
               signature -> compartment -> state.
```

Figure 6.4: RISC-V states in Rocq

Relating the memories is the most interesting part of our instantiation. Indeed, the two executions can have different allocation behaviours, and our relations must reflect that. To do so, we parametrize the relations by two CompCert *memory injections* [22] j_1 and j_2, one for each of the original executions. Their role is to relate the memory of their respective execution to the memory in the recomposed execution.

[22]: Leroy (2009), 'A Formally Verified Compiler Back-end'

A memory injection is a partial map $j : Block \rightarrow Block \times Offset$ that describes how memory blocks are transformed; for instance, in our case, if $j_1 b = (b', o)$, it means that the memory region located at block b in the first execution will be found starting at block b' and offset o in the recomposed execution.

The injections are built iteratively during the simulation proof, starting from initial memory injections relating the public symbols of their side, and adding new mappings after each allocation. In particular, we make sure that they do not relate the memory locations of the side that doesn't correspond to their execution. We define in Rocq a relation `mem_rel (j: meminj) (delta: side) (m1 m3: mem)` that captures all the properties we want of the memory injections relating `m1` and `m3`, where `m3` is the memory of the recomposed execution and `m1` is that of either the first or the second, depending on the `delta` parameter. In addition, we also define a predicate `meminj_preserves_globals` inspired from the one in `Unusedglobproof.v`, that states that our memory injections are defined over and preserve the global symbols of the compartments of their side. This allows us to show that system calls commute with the memory injections.

Then, we define a weak and a strong relation using those injections, named `weak_equivalence` and `strong_equivalence`. We leave the details of these definitions to the Rocq development; the important part is that the weak relation relates the memories of the runs according to an injection, and the strong relation does the same, but additionally relates the content of the registers.

The mixed relation (or common equivalence) relates the content of the cross-compartment stacks in the three executions. A simplified definition of the relation between the stacks can be seen below:

```
Inductive stack_rel (j1 j2: meminj) m1 m2 m3:
  stack -> stack -> stack -> Prop :=
| stack_rel_empty:
    stack_rel j1 j2 m1 m2 m3 nil nil nil
| stack_rel_cons: forall st1 st2 st3 f1 f2 f3,
    stack_relj1 j2 m1 m2 m3 st1 st2 st3 ->
    stackframe_rel j1 j2 m1 m2 m3 f1 f2 f3 ->
    stack_rel j1 j2 m1 m2 m3 (f1 :: st1) (f2 :: st2) (f3 :: st3)
.
```

The important part is `stackframe_rel`: we define this predicate to use either `j1` or `j2`, depending on which execution the stack frame belongs to.

Proofs Only the proofs of the 4 diagrams from Figure 6.3 are interesting, since the remaining 4 follow by exploiting symmetries of the diagrams and of the relations.

The first two diagrams require that silent steps don't affect other compartments. To prove them, we show several intermediary lemmas that state that memory operations (allocation, store, free) preserve the relations. Because of the large number of different instructions, we discard most goals with an automation tactic that discards contradictory cases, and applies preservation lemmas automatically. The (simplified) statement of Figure 6.3a in our setting is given in Figure 6.5. Note that the injection

```
Lemma step_E0_strong: forall (s1 s1': state),
  Step (semantics W1) s1 E0 s1' ->
  forall (s2 s3: state) j1 j2,
    meminj_preserves_globals j1 ->
    meminj_preserves_globals j2 ->
    stack_rel j1 j2
      (mem_of_state s1) (mem_of_state s2) (mem_of_state s3)
      (stack_of_state s1) (stack_of_state s2) (stack_of_state s3) ->
    strong_equivalence j1 s1 s3 ->
    weak_equivalence j2 s2 s3 ->
    exists (s3': state) j1',
      Plus (semantics W3) s3 E0 s3' /\
        meminj_preserves_globals j1' /\
        meminj_preserves_globals j2' /\
        stack_rel j1' j2
          (mem_of_state s1') (mem_of_state s2) (mem_of_state s3')
          (stack_of_state s1') (stack_of_state s2) (stack_of_state s3') /\
        strong_equivalence j1' s1' s3' /\
        weak_equivalence j2 s2 s3'.
```

Figure 6.5: Rocq formalization of the lemma for a silent step in strongly related states

for the strongly related side is allowed to change, because an allocation might have happened, which requires adding a mapping to the injection. The lemma `step_E0_weak` corresponding to Figure 6.3b is similar, but doesn't require the existence of a transition of the third execution (`Plus`) as a conclusion.

Finally, the last diagrams, where an event is produced, are proved by re-establishing the three relations after producing the correct event. They are proved by a single lemma called `step_t`, reproduced in Figure 6.6. One can see the two parallel steps in the assumptions, and the possible switching of relations in the conclusion.

Proving these lemmas is made possible by the fact that all information that is shared between the two compartments on a call or return appears in the trace event. This wouldn't hold if stack frames storing spilled arguments were still writable by the caller, since otherwise a compartment could surreptitiously communicate by changing previously passed arguments in an old stack frame. We discovered this issue while trying to complete the proof of these three diagrams and failing, as we couldn't prove that a compartment reading an argument from the stack would get related values according to the memory injection of its side (it could instead get a pointer from the other side that's not in the memory injection). This required us to step back and change the semantics of RISC-V assembly to make old arguments passed on the stack read only, as described in subsection 5.2.4.

```
Lemma step_t: forall (s1 s1': state) (s2 s2': state) e,
  Step (semantics W1) s1 (e :: nil) s1' ->
  Step (semantics W2) s2 (e :: nil) s2' ->
  forall (s3: state) j1 j2,
    meminj_preserves_globals j1 ->
    meminj_preserves_globals j2 ->
    stack_rel j1 j2
      (mem_of_state s1) (mem_of_state s2) (mem_of_state s3)
      (stack_of_state s1) (stack_of_state s2) (stack_of_state s3) ->
    strong_equivalence j1 s1 s3 ->
    weak_equivalence j2 s2 s3 ->
    exists s3' j1' j2',
      Plus (semantics W3) s3 (e :: nil) s3' /\
        meminj_preserves_globals j1' /\
        meminj_preserves_globals j2' /\
        stack_rel sj1' j2'
          (mem_of_state s1') (mem_of_state s2') (mem_of_state s3')
          (stack_of_state s1') (stack_of_state s2') (stack_of_state s3')
          ↪ /\
        ((strong_equivalence j1' s1' s3' /\
            weak_equivalence j2' s2' s3') \/
          (weak_equivalence j1' s1' s3' /\
             strong_equivalence j2' s2' s3')).
```

Figure 6.6: Rocq formalization of the lemma for parallel steps producing an event

While our proof structure is similar to the turn-taking simulation of El-Korashy et al. [3], we greatly benefit from the proof diagrams above, as they give us a clear structure to organize the proof. Moreover, because we rely on CompCert's memory injections instead of ad-hoc memory renamings, we are able to reuse much of the machinery that already exists for the compiler correctness proof.

[3]: El-Korashy et al. (2022), 'SecurePtrs: Proving Secure Compilation with Data-Flow Back-Translation and Turn-Taking Simulation' **[Co-authored]**

6.3 Discussion

We have shown how to prove the individual pieces of the proof technique to prove $\mathsf{RSC}^{\mathsf{DC}}_{\mathsf{MD}}$ of our compiler, SECOMP.[3] Building and verifying SECOMP was a long venture that took many man-months of work. SECOMP adds ~43k LoC on top of CompCert, mostly in proofs. In more detail, our extensions to CompCert 3.12 and its correctness comprise ~7k LoC of specs and ~11k of proofs, an increase of 7.2% and 22.3% respectively. In addition, back-translation involves ~5k LoC of specs and ~6k of proofs; recomposition ~1k LoC of specs and ~8k of proofs; and blame ~1k LoC of specs and ~4k of proofs.

3: Blame is discussed in the paper, but we leave it out of this thesis.

Limitations First, the back-translation can only handle traces up to a certain length, that is 2^{64} events per compartment. This is because the number of already executed events is stored in a single 64-bit integer. One could imagine schemes to handle more events, using arrays of integers, however it remains unclear whether there would still always be a limit or not.

Then, the proof technique designed by Abate et al. [8] and that we use for SECOMP does not directly apply. Indeed, the proof technique assumes that compilation is total (it always succeeds), so that there is always a compiled back-translated program. CompCert does not satisfy this assumption: compilation can fail. Partiality comes mainly from several sources in CompCert: first, some steps can fail: two steps are not verified but instead use translation-validation, while the data-flow analyzer used in some optimizations can fail if the analysis does not converge fast enough. Some passes use errors to rule out code that couldn't have been

[8]: Abate et al. (2018), 'When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise'

generated by the compiler, to avoid having to carry invariants through several passes. Finally, some passes constrain the size of the generated stack frames, which could make compilation fail.

None of these issues seem unsolvable on their own, but making CompCert total would amount to a considerable amount of work. Alternatively, one could attempt to prove that CompCert is total for a certain subset of programs in which our back-translated programs belong. We do not do any of these, and instead opt for systematically testing the compilation of the back-translation, with property-based testing. This gives us reasonable confidence that our back-translated programs can always be compiled.

Capability backend Finally, one thing we didn't mention in this thesis but is touched upon in the full paper is our prototype implementation of a capability-based backend. While still extremely basic, this prototype suggests that the compartmentalization mechanism can effectively be implemented on a realistic machine, at a lower-level than CompCert's RISC-V.

Extensions

7 Memory Sharing

The various works presented in the previous chapters, including SECOMP, do not allow for memory sharing between compartments. In these works, compartments can only communicate via functions call and returns, and are prevented from passing pointers directly. However, passing pointers and having other functions read and write to them is a common programming pattern in any C-like language.

In this chapter, we discuss SecurePtrs [3], an extension of [7] that integrates memory sharing. Pointers are seen as unforgeable capabilities that can be passed to other compartments to grant them access to a memory region.

This new source of interactions between the context and the program makes the definition of traces [45, 48] much more complex, and as a result the back-translation and the overall proof of secure compilation significantly more complex. It is also concerning that similar paper proofs (for full abstraction) that consider memory sharing [45] end up running way over one hundred pages, and formalizing them in a proof assistant would likely be impossible in a reasonable amount of time.

Our contribution is SecurePtrs: a verified secure compiler that allows memory sharing via capability-like pointers. To build and verify SecurePtrs, we reuse the proof technique of [8], but we introduce improvements to the intermediary lemmas:

- *data-flow back-translation*, a form of back-translation that uses more informative events to record changes in the memory
- *turn-taking simulation*, an improvement of recomposition that, similarly to SECOMP (section 6.2), makes use of novel "strong" and "weak" simulation relations.

In this chapter, we focus on the first contribution, the *data-flow back-translation*. To understand why we need a new back-translation technique, consider the following example: suppose a compartment shared one of its pointers to another compartment. That other compartment might then have stashed this pointer somewhere, before returning to the first compartment. Later in the execution, the context may recover this shared pointer from where it stashed it, and write through it; because this is a write to shared memory, this event must be recorded in the trace, and the back-translated compartment will have to reproduce that exact write. However, the back-translated compartment cannot forge a pointer, and does not have access to the stashed pointer because it lives in private memory (which shall never be recorded in the trace).

Prior work solved this issue in the back-translation by maintaining in the back-translated program a store of *all* previously shared pointers, including those that were shared transitively (pointers found in shared memory, for instance). However, this leads to extremely complex invariants that would be hard if not impossible to mechanize in a proof assistant. Our solution, *data-flow back-translation*, enriches the traces with new *data-flow events* recording *every* memory operation. The back-translation then reexecutes all these operations one-by-one, essentially simulating the whole of the target memory in the source.

In the rest of this chapter, we will expand upon this new data-flow back-translation. In section 7.1, we discuss the memory sharing model

[3]: El-Korashy et al. (2022), 'SecurePtrs: Proving Secure Compilation with Data-Flow Back-Translation and Turn-Taking Simulation' **[Co-authored]**

[7]: Abate et al. (2019), *When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise* **[Co-authored]** (arXiv version with recomposition)

[45]: El-Korashy et al. (2021), 'CapablePtrs: Securely Compiling Partial Programs Using the Pointers-as-Capabilities Principle'

[48]: Laird (2007), 'A Fully Abstract Trace Semantics for General References'

[45]: El-Korashy et al. (2021), 'CapablePtrs: Securely Compiling Partial Programs Using the Pointers-as-Capabilities Principle'

[8]: Abate et al. (2018), 'When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise'

we consider in detail, and we highlight the major difficulties of this setting when it comes to back-translation. Then, in section 7.2, we discuss our data-flow back-translation technique, explaining how we solved the issues we raised before. Finally, in section 7.3, we describe how we put everything together, using data-flow back-translation, turn-taking simulation, and the proof technique from [8] to build SecurePtrs and verify it in the Rocq proof assistant.

[8]: Abate et al. (2018), 'When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise'

Author contribution I, the author of this thesis, am one of the authors of [3]. My main contribution was the implementation and part of the design of the data-flow traces and the data-flow back-translation. I have not participated in the design or implementation of the turn-taking simulation, which is why it is not discussed here.

7.1 Memory Sharing Model

We place ourselves in the setting described in chapter 3, the same as [8]. The major difference we introduce is that we allow compartments to pass pointers to each other, via function call.

[8]: Abate et al. (2018), 'When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise'

Pointers are treated as capabilities [49–51]: the mere possession of a pointer grants the authority to read and write in the memory block it points to. Importantly, this extends transitively: if one has a pointer to a memory block, and this block contains other pointers, then they also have access to the pointers and hence to the blocks they point to. Once another compartment gets access to a pointer from another compartment, the block it points to must now be considered *shared*; and, because compartments can easily store pointers in their private memory, memory cannot be unshared.

[49]: Fabry (1974), 'Capability-based addressing'
[50]: Carter et al. (1994), 'Hardware support for fast capability-based addressing'
[51]: Woodruff et al. (2014), 'The CHERI capability model: Revisiting RISC in an age of risk'

In such a setting, it is natural and important that one can state properties about this shared memory, and the obvious way to do it is on traces, by recording the state of the shared memory on the traces. Moreover, for the proof of security, one needs to track this shared memory on the traces, or recomposition wouldn't hold (there would be covert communication through the shared memory between compartments). Because the setting is sequential, the way we do it, following the example of El-Korashy et al. [45], is to extend the events from [8] to record a *shared memory* `M`: a partial view of the memory that only looks at shared blocks, as depicted in Figure 7.1.

[45]: El-Korashy et al. (2021), 'CapablePtrs: Securely Compiling Partial Programs Using the Pointers-as-Capabilities Principle'

$$
\begin{aligned}
a := {} & \ldots \\
& \mid \texttt{Event_call cp cp.id M v} \\
& \mid \texttt{Event_return cp cp M v}
\end{aligned}
$$

Figure 7.1: Trace events recording cross-compartment calls and returns with a shared memory

To apply the proof technique from [8], we would need to update their back-translation to also back-translate the changes to the memory between two events. For instance, consider the very simple trace:

$$m = \texttt{Event_call C1 C2.f M 0; Event_return C2 C1 M' 0}.$$

When back-translating the second event, the back-translation would have to *compute* the difference between `M'` and `M` and generate source expressions to apply it. Even disregarding the issue of computing the difference, the back-translation must ensure that `C2` still has access to all locations it must modify in this step; this means that `C2` must have

stashed every single pointer it ever had access to, including transitively, just in case it needs them again later in the trace.

This is exactly what prior work [45, 52] has been doing: they built in the back-translated program a custom data structure whose role is to stash all pointers obtained through a graph traversal of the memory. We attempted proving such a construction correct in a proof assistant, but it is far from trivial, and even just proving termination of the graph traversal is a challenge (the memory may contain cyclic data-structures).

[45]: El-Korashy et al. (2021), 'CapablePtrs: Securely Compiling Partial Programs Using the Pointers-as-Capabilities Principle'
[52]: Patrignani et al. (2021), 'Robustly Safe Compilation, an Efficient Form of Secure Compilation'

The key observation that leads to our data-flow back-translation is that one shouldn't have to stash *all possible* pointers at each step; instead, if one knew exactly what pointers were used and when, for instance because it's recorded on a trace, back-translation would become much simpler.

7.2 Extending the Back-Translation Proof with Data-flow Traces

The key idea for the data-flow back-translation is to enrich the trace events to capture all *data flows*, even those internal to the compartments. We call these enriched traces *data-flow traces*, and the new type of data-flow events is described in Figure 7.2. The new events, $\mathscr{E}$, extend the regular ones to capture all possible data flows in the target language. The events `dfCall` and `dfRet` are just the `Call` and `Ret` events of Figure 7.1. The remaining events correspond to target language instructions that cause data flows: loading a constant to a register (`Const`), copying from a register to another (`Mov`), binary operations (`BinOp`), copying from a register to memory or vice-versa (`Store`, `Load`) and allocating a fresh location (`Alloc`). Importantly, in a data-flow trace, every event records the entire state—both shared state and state private to individual components. Accordingly, in the data-flow events, `M` also includes locations that were not shared to other components, and `R` is the state of the register file.

$$
\begin{aligned}
\mathscr{E} ::= {} & \texttt{dfCall M R cp cp.id v} \\
\mid {} & \texttt{dfRet M R cp cp v} \\
\mid {} & \texttt{Const M R cp v } \mathtt{r}_{\mathtt{dest}} \\
\mid {} & \texttt{Mov M R cp } \mathtt{r}_{\mathtt{src}}\ \mathtt{r}_{\mathtt{dest}} \\
\mid {} & \texttt{BinOp M R cp op } \mathtt{r}_{\mathtt{src1}}\ \mathtt{r}_{\mathtt{src2}}\ \mathtt{r}_{\mathtt{dest}} \\
\mid {} & \texttt{Load M R cp } \mathtt{r}_{\mathtt{addr}}\ \mathtt{r}_{\mathtt{dest}} \\
\mid {} & \texttt{Store M R cp } \mathtt{r}_{\mathtt{addr}}\ \mathtt{r}_{\mathtt{src}} \\
\mid {} & \texttt{Alloc M R cp } \mathtt{r}_{\mathtt{ptr}}\ \mathtt{r}_{\mathtt{size}}
\end{aligned}
$$

Figure 7.2: Events of data-flow traces

7.2.1 Outline of data-flow back-translation proof

Data-flow traces simplify the proof of back-translation by splitting it into two key lemmas: enriching interaction traces to data-flow traces and back-translation of data-flow traces, both of which are shown below. Recall that T denotes a data-flow trace. We define `remove_df`(T) to denote the interaction trace obtained by removing all internal data-flow events from T, i.e., by retaining only `Call` and `Return` events.

Lemma 7.2.1 (Enrichment)

$$\forall \mathtt{P}\, t.\, \mathtt{P} \rightsquigarrow t \implies$$
$$\exists T.\, \mathtt{P} \rightsquigarrow_{DF} T \wedge t = \mathtt{remove_df}(T)$$

Proof. Immediate from the definition of the target-language semantics. □

Enrichment is reminiscent of the first lemma from section 2.2 that we used to go from "standard" traces to "informative" traces.

Lemma 7.2.2 (Data-flow back-translation)

$$\forall \mathtt{P}\, T.\, \mathtt{P} \rightsquigarrow_{DF} T \implies$$
$$\exists \mathbf{P}\, t.\, \mathbf{P} \rightsquigarrow_{t} \wedge\, t \sim \mathtt{remove_df}(T)$$

This lemma is a standard back-translation lemma, but it also includes a component to go back to "standard" traces, much like one of the lemmas from section 2.2. The $\sim$ relation captures the fact that a renaming of the pointers occurs, as they are offset by one when going from the target to the source. The reason for this renaming is discussed in subsection 7.2.2.

7.2.2 Data-flow Back-Translation, in Practice

We provide some details on how we implement the data-flow back-translation, written $\uparrow$, and how we prove Lemma 7.2.2.

$\uparrow$ takes as input a data-flow trace T and outputs a whole program **W** that produces the standard trace $\mathtt{remove_df}(T)$. As for the proof of Abate et al. [8], each compartment in **W** maintains an *event counter* to keep track of which trace event the component is currently mimicking. As part of the main loop, this counter is incremented, and the appropriate branch in order to emit the next event is chosen based on this counter. This counter, as well as a small amount of other metadata used by the back-translation, is stored inside the statically allocated buffers of each component of **P**, which are accessed using the **local** construct.

[8]: Abate et al. (2018), 'When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise'

Mimicking register operations A technical difficulty in the back-translation is that, unlike the target, the source language does not have registers. In order to mimic data-flow events involving registers, **P** simulates these registers and operations on them within the static buffer of the active component. For instance, a $\mathtt{Mov}\ \mathtt{M}\ \mathtt{R}\ \mathtt{c}_{\mathtt{cur}}\ \mathtt{r}_{\mathtt{src}}\ \mathtt{r}_{\mathtt{dest}}$ event (which copies a value from register $\mathtt{r}_{\mathtt{src}}$ to register $\mathtt{r}_{\mathtt{dest}}$) is simulated by the expression $(\mathbf{local} + \mathbf{OFFSET}(\mathtt{r}_{\mathtt{dest}})) := !(\mathbf{local} + \mathbf{OFFSET}(\mathtt{r}_{\mathtt{src}}))$, where $\mathbf{OFFSET}(\mathtt{r})$ is a macro that is statically expanded to the offset corresponding to register $\mathtt{r}$ in the simulated register file, and $!$ is dereferencing.

Mimicking memory operations Because the source and target memory models coincide, we are able to back-translate memory events quite easily. That is, a `Store` event is back-translated using assignment ($:=$) and a `Load` event is back-translated using dereferencing ($!$). Since the static buffer (whose block number is 0 in our semantics) is already used by the back-translation to store metadata and simulated registers, the back-translated program's memory shifts by one block relative to the memory in the

target: for each component, block b in the target corresponds to block $b+1$ in the source. This explains why we use the relation $\sim$ in the statement of the correctness of the data-flow back-translation.

The memory layout of the back-translated program ($\mathbf{P} = T\uparrow$) relative to the given target program P is shown in Figure 7.3.

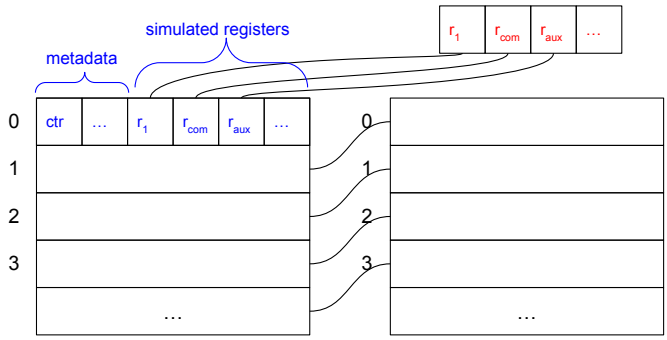


Figure 7.3: Memory layout of a back-translated component (left) compared to a target component (right) witnessing the increment-by-1 block-id-renaming relation

Mimicking calls and returns The target language's semantics enforces a calling convention: calls and returns store the argument or return value in $\mathsf{r}_{\mathsf{COM}}$, and set all other registers to zero. Therefore, calls and returns in **P** need extra administrative steps to mimic this convention. For example, mimicking a call event requires two administrative steps:

1. in the caller, dereference the content of the location simulating $\mathsf{r}_{\mathsf{COM}}$ to get the argument and pass it to the function;
2. in the callee, assign the function argument **arg** to the location simulating $\mathsf{r}_{\mathsf{COM}}$, and set all other registers to 0.

Similar administrative steps are needed for mimicking a return event.

Proof of back-translation To prove back-translation (Lemma 7.2.2), we use a simulation lemma that ensures a relation `mimicking_state` holds between the state of **P** and the prefix mimicked so far. Intuitively, `mimicking_state` T_{pref} T_{suff} **s** means that **s** is the state reached after mimicking all the data-flow events in T_{pref}, and that the starting state of the remaining trace T_{suff} matches **s**.

Lemma 7.2.3 (Trace-prefix mimicking)

$$
\begin{aligned}
&\forall\, \mathsf{P}\; T\; T_{pref}\; T_{suff}.\\
&\quad \mathsf{P} \rightsquigarrow_{DF} T \implies\\
&\quad T = T_{pref} +\!\!+ T_{suff} \implies\\
&\quad \exists\, \mathbf{s}\; t'_{pref}.\\
&\qquad T\uparrow \xrightarrow{t'_{pref}}{}^{*} \mathbf{s} \wedge t'_{pref} \sim \texttt{remove_df}(T_{pref}) \wedge\\
&\qquad \texttt{mimicking_state}\; T_{pref}\; T_{suff}\; \mathbf{s}
\end{aligned}
$$

Because Lemma 7.2.3 ensures the relation `mimicking_state` holds for every prefix, it effectively states that the memory of the back-translation is in lock-step with the `M` and `R` components appearing in each data-flow event $\mathscr{E}$ from T. `mimicking_state` is also strong enough to ensure that the trace relation holds between the projection of the prefix mimicked

so far `remove_df`(T_{pref}) and the corresponding prefix t'_{pref} that the back-translation emits.

Comparison with code-based back-translation It is legitimate to compare our data-flow back-translation with both traditional trace-based back-translation and code-based back-translations.

Indeed, a data-flow trace can be interpreted as the sequence of instructions that were executed in the code of the target program, during a particular execution. In that sense, the data-flow back-translation borrows from code-based back-translation in that it also *simulates* the target code in the source.

However, we want to note that our data-flow back-translation only simulates a target program *along a certain execution path*, while traditional code-based approaches need to simulate the code *along all execution paths*. For this reason, our data-flow back-translation is still easier to implement and to verify and is likely applicable to more settings than code-based back-translation.

7.3 Verification of SecurePtrs

Using these ideas, we implemented and verified SecurePtrs, a verified secure compiler extending the compiler from Abate et al. [8] with memory sharing.

[8]: Abate et al. (2018), 'When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise'

We reuse their proof technique, but we replaced their back-translation with our novel data-flow back-translation, and recomposition with a novel turn-taking simulation notion that is described in full detail in the paper [3].

[3]: El-Korashy et al. (2022), 'SecurePtrs: Proving Secure Compilation with Data-Flow Back-Translation and Turn-Taking Simulation' **[Co-authored]**

7.3.1 Verifying the Data-Flow Back-Translation

We verify the data-flow back-translation following the ideas highlighted in the previous section, in particular Lemma 7.2.3. The proof is highly complex, around 15k lines for the proof of back-translation alone.

There are two main reasons for this complexity: first, we work in a custom memory model where we cannot reuse already existing theorems from CompCert. In particular, reasoning about the offset-by-1 block renaming is difficult, because we are not able to reuse all the machinery already developed in CompCert for *memory injections*, like we are able to do for SECOMP. In hindsight, it might have been a better idea to focus on developing a better theory of memory renamings, or straight up reuse the CompCert memory injections, which would have saved us a lot of work.

The second reason comes from a particularity of the back-translation: compartments do not start in a state where they are ready to execute right away. Instead, our back-translated compartments must, when they get control for the first time, initialize themselves, which is essentially writing some default metadata in the right locations and allocating a copy of the target initial local buffer. Proving this initialization correct, though not the biggest part of our proof, is still a non-trivial amount of work.

7.3.2 Rocq Mechanization

Just like Abate et al. [8], we mechanize the entirety of the proof technique, with the exception of forward and backward compiler correctness.

We only need one additional axiom, which is that our compiler preserves the privacy of the local buffer, meaning that source programs that don't share their local buffers also do not share it once compiled. We expect this axiom to hold, because our compiler does not merge memory blocks, but we would also expect it to hold for compilers that merge blocks, as long as they don't merge private and shared blocks together.

The entire development amounts to around 30k lines of code, of which a bit more than half are dedicated to the data-flow back-translation and a bit less than half to the updated recomposition lemma.

[8]: Abate et al. (2018), 'When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise'

8 Nanopass Back-Translation

Through this thesis, we've built secure compilers that help protect source programs against low-level adversarial contexts. However, with the exception of the compiler from section 2.2, all these compilers were protecting the safety properties of the source program: properties whose violation is witnessed by a finite trace prefix (a "bad prefix"). However, there is value in protecting stronger properties, such as hypersafety properties and relational hypersafety properties. Indeed, those can be used to encode standard security properties such as noninterference [27] and observational equivalences [53].

[27]: Goguen et al. (1982), 'Security Policies and Security Models'

[53]: Abadi (1999), 'Protection in Programming-Language Translations'

Achieving secure compilation is challenging, and as we've shown in this thesis, even more challenging is it when trying to prove it formally. Proofs such as the ones we showcased but also proofs from the literature rely on a notion of *back-translation*: given a target context performing an attack on a compiled program, often evidenced by execution traces, one must build a source context performing the same attack against the source program. Such back-translations and their corresponding proofs can be highly non-trivial, especially when the attacks are against hyperproperties and relational hyperproperties. For such classes of security properties, attacks are witnessed by several traces that the generated context must be able to produce, depending on the observable behaviour of the program with which it is linked.

In this chapter, as an important step towards large-scale secure compilation proofs for hyperproperties and relational hyperproperties, we propose a novel and general technique for back-translating a finite set of finite trace prefixes into a single source context, and a convenient way to prove the correctness of this back-translation in a proof assistant. We observe that the set of finite trace prefixes recording the calls and returns between the program and the context can be represented as a *call-return tree* branching on calls and returns from the program to the context. During the execution, the context produced by our back-translation has to deduce, from the events that have already been executed, which branch it is running (i.e. what subset of programs it could still be linked with), in order to know what event to emit next. To do so, the context produced by our back-translation maintains private state storing its last *location* in the tree. Every time the context gets control, it uses this information as well as the argument or the return value from the program, to decide what event to produce next, and how to update its own location.

This raises two main challenges: (1) the generated context must correctly handle *reentrancy*, since the program can perform arbitrary *callbacks*; and (2) because these callbacks can be arbitrarily nested, returns may occur arbitrarily far from their corresponding call. So the generated context manipulates a complex state that is difficult to reason about.

As we have experienced first-hand in previous failed attempts to produce a mechanized proof, a monolithic approach leads to writing a complex back-translation function and to dealing with many invariants at the same time trying to prove this complex back-translation correct. To address these challenges, we instead structure the back-translation as *a nanopass compiler* [54–56]. The back-translation is done in several small steps, each performing a simple transformation of the tree, adding to each node new information describing how the location in the tree changes depending

[54]: Keep (2020), 'The Nanopass Framework as a Nanopass Compiler (ELS keynote)'

[55]: Sarkar et al. (2005), 'A Nanopass framework for compiler education (educational pearl)'

[56]: Keep et al. (2013), 'A nanopass framework for commercial compiler development'

on how the context regains control. This information is then used to generate the source context that produces the right trace prefixes.

We prove this back-translation correct by adapting compiler correctness techniques. We give semantics to each intermediate tree language and prove several small CompCert-style forward simulations [22, 23] between these representations. These small-step operational semantics reduce a tree by following a trace: for each event of the trace, the semantics follow the corresponding branch, and emit the event. The semantics also explicitly enforce invariants of the execution, using *ghost state* to store information, such as what events have already been executed, or an abstract call stack. In the last compilation step, the simulation proofs link this ghost state to a *concrete state* from the source language semantics.

[22]: Leroy (2009), 'A Formally Verified Compiler Back-end'
[23]: Tan et al. (2019), 'The verified CakeML compiler backend'

Thanks to this modular structure, the effort of proving these small transformations from trees to trees and eventually to source code is lower, and we have fully mechanized these transformations and their correctness proofs. The back-translation targets the compilation chain of Abate et al. [8] (the one described in chapter 3 and chapter 4), and is designed to replace the existing back-translation step of the existing proof. While we have not yet mechanized this part, it should be relatively easy to show that this compilation chain satisfies a much stronger secure compilation criterion—i.e. that it robustly preserves a class of relational hyperproperties including hypersafety (e.g. noninterference) and relational hypersafety properties (e.g. trace equivalence, thus also obtaining full abstraction)—by simply replacing the previous simplistic back-translation handling only one finite trace prefix with ours that back-translates any finite set of finite trace prefixes.

[8]: Abate et al. (2018), 'When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise'

Finally, we believe that this general technique should be reusable for other secure compilation proofs. In particular, the nanopass approach allows us to structure the back-translation into two distinct phases: a tree-manipulating phase that only depends on the trace model, but on no other language features, and a code generation phase. Even the code generation phase and the proofs pertaining to it should be mostly reusable, as they only rely on programming language features that are common to many languages (e.g. procedures, private state). For this reason, we see this work as an important step in devising secure compilation proof techniques that can be scaled up to real-world languages, such as the recent compartmentalized version of CompCert, SECOMP [2], that was described in chapters chapter 5 and chapter 6.

[2]: **Thibault** et al. (2024), 'SECOMP: Formally Secure Compilation of Compartmentalized C Programs' **[Co-authored]**

The remainder of the chapter is structured as follows: section 8.1 first gives a higher-level account of our back-translation and proof technique, explaining the key ideas. section 8.2 then presents the design of the semantics of the intermediate languages and of the back-translation passes. section 8.3 then outlines how we proved correctness for this back-translation using forward simulations. Finally, section 8.4 describes our Rocq formalization.

Author contribution I, the author of this thesis, am the lead author of the paper [1] upon which this chapter is based. I devised the different steps and the main ideas behind the proofs, and did most of the technical work when it comes to the Rocq implementation of the ideas presented in this chapter.

[1]: **Thibault** et al. (2025), 'Nanopass Back-Translation of Call-Return Trees for Mechanized Secure Compilation Proofs' **[Co-authored]**

8.1 Key Ideas

We set our work in the setting of Abate et al. [8]. We refer the reader to chapter 3 which contains a detailed description of the setting and, in particular, of their proof technique (section 3.2). To simplify the exposition throughout the chapter, we make the simplifying assumption that each compartment is made of only one procedure, and we equate the compartments and the procedure each contains.

[8]: Abate et al. (2018), 'When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise'

At the heart of this chapter is the following observation: by strengthening the back-translation step, it is easy to extend the proof of Abate et al. [8] to show a much stronger secure compilation criterion.

8.1.1 Towards a proof of a stronger criterion

While Abate et al. [8] proved that their compilation chain satisfies RSP, this is still a weak secure compilation criterion, one of the weakest they study. Indeed, RSP only considers attacks represented by a single finite trace prefix. What this means is that a secure compilation chain satisfying RSP only robustly preserves safety properties. In general, safety properties cannot capture data confidentiality and integrity, which are essential security properties that one would want their compilation chain to preserve.

Thankfully, Abate et al. [6] also propose stronger criteria that consider several trace prefixes, which can also represent attacks against data confidentiality and integrity (see also chapter 2). One such criterion is the one named *Robust Finite-relational Safety Preservation* (RfrSP):

[6]: Abate et al. (2019), 'Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation' **[Co-authored]**

Definition 8.1.1 (Robust Finite-relational Safety Preservation) *A compilation chain is said to satisfy* RfrSP *when:*

$$\begin{aligned}&\forall K.\ \forall \mathbf{P_1} \ldots \mathbb{P}\mathbb{K}\ \mathbf{C_T}\ m_1 \ldots m_K.\\ &\quad (\mathbf{P_1}{\downarrow} \bowtie \mathbf{C_T} \rightsquigarrow m_1 \wedge \ldots \wedge \mathbf{P_K}{\downarrow} \bowtie \mathbf{C_T} \rightsquigarrow m_K) \implies\\ &\quad \exists \mathbf{C_S}.\ (\mathbf{P_1} \bowtie \mathbf{C_S} \rightsquigarrow m_1 \wedge \ldots \wedge \mathbf{P_K} \bowtie \mathbf{C_S} \rightsquigarrow m_K).\end{aligned}$$

RfrSP states that any attack against a finite-relational safety property of compiled programs $\mathbf{P_1} \ldots \mathbf{P_K}$ characterized by the finite trace prefixes $m_1 \ldots m_K$, can also be mounted against the source programs (i.e. there exists a source context $\mathbf{C_S}$ that induces the same prefixes). In [6] and section 2.2, we also proved that a simple compiler that erases types satisfies a variant of RfrSP.

This criterion is particularly interesting, as it implies the robust preservation of all hypersafety properties [26], a large class that includes not just safety but also most variants of noninterference. In our context, it also implies a variant of (the security-relevant direction of) the well-studied criterion of fully abstract compilation [6, 53]. Finally, this is the strongest criterion of Abate et al. [6] that can be proved by back-translating trace prefixes.

[6]: Abate et al. (2019), 'Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation' **[Co-authored]**
[53]: Abadi (1999), 'Protection in Programming-Language Translations'

The RfrSP criterion is, however, harder to prove than RSP: while for RSP the context generated by back-translation only needs to produce a single trace prefix when linked with a single known program, for RfrSP the produced context needs to be able to produce finitely many trace prefixes when linked with several possibly different programs.

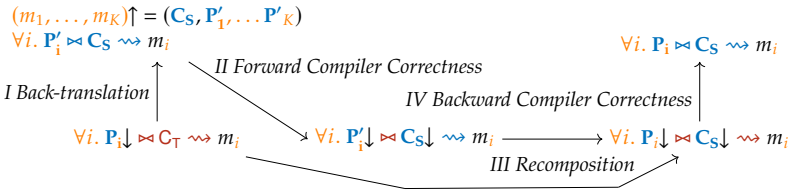


Figure 8.1: Outline of Abate et al. [8]'s proof technique adapted for RfrSP (parts changed in orange)

The first insight is that it is possible to prove RfrSP using the same proof technique, with the sole difference in the back-translation. Indeed, suppose one is able to provide a back-translation $(m_1, \ldots, m_K)\uparrow = (\mathbf{C_S}, \mathbf{P'_1}, \ldots, \mathbf{P'_K})$ such that $\forall\, i.\ \mathbf{P'_i} \bowtie \mathbf{C_S} \rightsquigarrow m_i$. Then, it is possible to plug this back-translation in place of the single-trace back-translation and apply all the other steps pointwise. One then obtains a proof of RfrSP. The updated proof technique is given in Figure 8.1, with the differences highlighted in red.

8.1.2 Back-translating multiple traces

How can one back-translate a finite set of finite trace prefixes though? Consider an example compartmentalized application with compartment $\mathtt{C}_\mathbb{C}$ in the context and compartment $\mathtt{C}_\mathbb{P}$. in the program. Suppose that the context is linked with three other programs so that we obtain the traces m_1, m_2, m_3 in Figure 8.2. A key idea is that such prefixes can be represented as a call-return tree whose branches are labeled with events, as seen on the left of Figure 8.2. In the languages we consider branching can only occur when the program has control, and two different programs decide to produce different events.

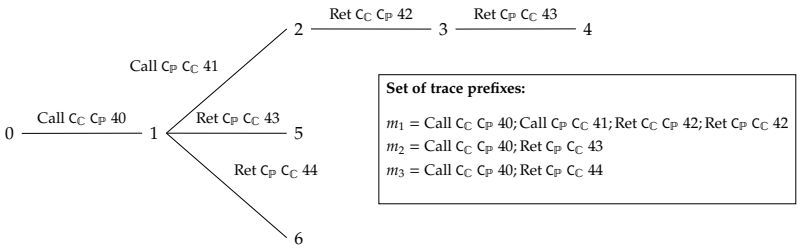


Figure 8.2: Example of a call-return tree representing a set of finite prefixes

In Figure 8.2, we give each node a unique identifier. These identifiers generalize the scheme of Abate et al. [8]: instead of relying on a counter that is always incremented, the context is now able to encode what events have already been executed by simply recording the *location* of the last executed event in a private variable `loc`. The compartment being back-translated can then use this information to determine which event to perform next and how to update `loc`. Determining how to update the `loc` counter and which event to produce is, however, far more involved than in the simple single-trace back-translation, and the code for the compartment $\mathtt{C}_\mathbb{C}$ that we generate from the traces in Figure 8.2 is given in Figure 8.3.

The core idea of the back-translation is for the generated compartment $\mathtt{C}_\mathbb{C}$ to maintain a private variable `is_call` that will be set to 1 whenever $\mathtt{C}_\mathbb{C}$ is about to pass control to another compartment (and generally when it doesn't have control). That way, it knows that a call it receives when `is_call` = 1 comes from another compartment (line 2), and that it must update its location `loc` based on the current value `loc` and the argument it received (lines 5–7).

[8]: Abate et al. (2018), 'When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise'

```
1  C_C(arg):
2  if (is_call) {
3    /* Receiving a cross-compartment callback; we update the location based
4       on the current location and the received argument */
5    if (loc = 1) {
6      if (arg = 41)
7        loc = 2;
8    }
9  } else {
10   /* this is not an incoming call, but rather a recursive call to handle
11      a return value. We update the location based on the current location
12      and the returned value (stored in res) */
13   if (loc = 1) {
14     if (res = 43)
15       loc = 5;
16     else if (res = 44)
17       loc = 6;
18   } else if (loc = 3) {
19     if (res = 43)
20       loc = 4;
21   } };
22 // Now we generate the appropriate event based on the current location
23 if (loc = 0) {
24   is_call = 1;
25   loc = 1;
26   res = C_P(40);
27   is_call = 0;
28   C_C();
29 } else if (loc = 2) {
30   is_call = 1;
31   loc = 3;
32   return 42; }
```

Figure 8.3: Code for $\mathbb{C}_\mathbb{C}$ generated by the back-translation

Following the code in Figure 8.3, $\mathtt{C}_\mathbb{C}$ goes directly to line 23, and starts branching on its current location (that just got updated). Depending on this value, it will produce either a cross-compartment call (lines 23–28) or a cross-compartment return (lines 29–33). In the case of a call, the compartment sets `is_call` to 1 (line 24) and updates its location (line 25), before calling the other compartment (line 26). When this new call returns, the return value will be stored in a third private variable `res` (line 26), the compartment will set `is_call` to 0 (line 27), and call itself recursively (line 28). Then, the `else` path of the first `if` will be taken (line 9), and the return value will be used, together with `loc`, to set a new location (lines 13–21). One may wonder why not set the new location right after the call, between lines 26 and 28? This is because $\mathtt{C}_\mathbb{C}$ might have received callbacks that updated `loc` in the meantime. Because `loc` might have changed, it is necessary for correctness to branch on it again; and it is simpler in terms of reasoning to gather all this branching at the same place in the code instead of it being dispersed among all the calls.

Finally, what happens when doing a return (lines 30–32)? Because it is about to pass control back, $\mathtt{C}_\mathbb{C}$ sets `is_call` to 1, updates its location, and starts returning. Then, the return pops the stack and reaches another compartment, which if also generated by our back-translation, will handle its return value in the way described above (lines 24–29).

Note that in our example, we only consider two compartments, and each compartment has only one procedure. In the general case, we support an arbitrary number of distinct compartments; we consider one tree per compartment: for a compartment `C`, it is the tree obtained by filtering out all events that do not involve `C` from the traces. We are able to do this filtering because a compartment cannot observe the interactions between two other compartments directly. This generalization does not affect the structure of the back-translated programs.

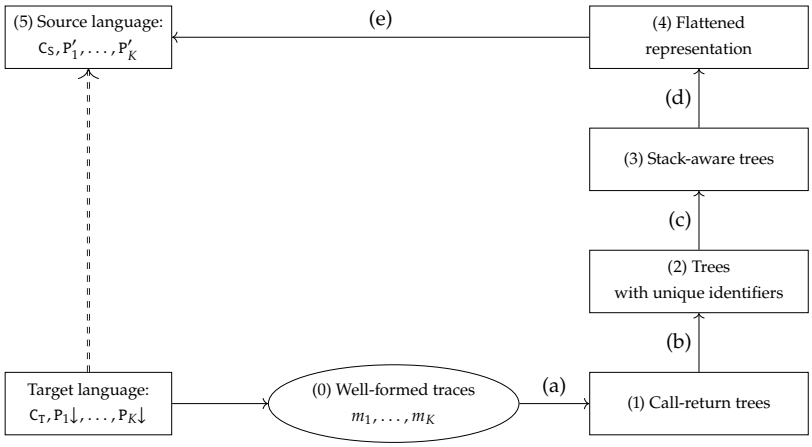


Figure 8.4: Steps of our nanopass back-translation

8.1.3 Nanopass back-translation

The back-translation described above is complex; to tame this complexity, we divide the back-translation into a series of small steps, each improving upon the previous one in a minimal way, in the style of a nanopass compiler [54–56].

[54]: Keep (2020), 'The Nanopass Framework as a Nanopass Compiler (ELS keynote)'
[55]: Sarkar et al. (2005), 'A Nanopass framework for compiler education (educational pearl)'
[56]: Keep et al. (2013), 'A nanopass framework for commercial compiler development'

For this we define several *intermediary tree languages* with more and more detailed semantics as we progress through the back-translation. Each step makes explicit one *invariant* that was previously implicit, or introduces a new piece of *ghost state* that will later be related to concrete state in the generated source code.

Figure 8.4 gives an overview of the different passes of our back-translation. Starting from the target language on the bottom left, the compiled programs produce a well-formed set of traces when linked with the target context. Our notion of well-formedness captures the notion of determinacy necessary for them to be representable as trees. This well-formed set of traces is the starting point of our formally verified back-translation:

(a) we prove they can be represented as call-return trees, as in Figure 8.2
(b) to each node of the trees we assign a unique identifier, which will be mapped to the variable `loc` in the code;
(c) we add an abstract cross-compartment call stack to the semantics of the trees;
(d) we flatten the trees into lists of rules that describe how each compartment should update its private state and emit events, depending on its current private state and the last event that was executed. For instance, a rule of the form $(n, \text{Call } C_1\ C_2\ 40, n')$ can mean—depending on whether it's a rule for C_1 or C_2—either that when at location n, compartment C_1 must update its location to n' and call C_2 with argument 40, or that when at location n, if compartment C_2 receives a call with argument 40 then it must update its location to n';
(e) for each such rule, we generate a source statement; for instance, the previous rule would be translated either to `loc = n'; C2(40); ...` (for C_1) or to `if(loc = n && arg = 40){loc = n'}` (for C_2). We then assemble the final source programs from these sub-statements.

We write $\uparrow_{(a-e)}$ to refer to the whole back-translation, and $\uparrow_{(a)}, \ldots, \uparrow_{(e)}$ for each back-translation step. We see verifying this back-translation as proving *compiler correctness*. We first equip the call-return trees with simple semantics of the form $T \rightsquigarrow_n m$ meaning that the call-return tree

T at level n produces the trace m. A step in this semantics works by plucking and emitting the event at the root and using the event to choose one of the child branches to reduce next. We use this to state the main theorem, saying that for any prefix m in the set of well-formed traces S we consider, the back-translation of S to a source whole program $\mathsf{S}\uparrow_{(\mathrm{a-e})}$ can produce m in the source semantics, $\rightsquigarrow_5$:

Theorem 8.1.1 (Correctness of call-return tree back-translation) *For any well-formed set of traces* S*, if* $m \in \mathsf{S}$ *then:* $\mathsf{S}\uparrow_{(\mathrm{a-e})} \rightsquigarrow_5 m$.

To formally prove this theorem in Rocq, we rely on the nanopass structure of the back-translation, and first prove an initialization lemma and then a compiler correctness lemma for each pass of the back-translation from levels (1) to (5).

Lemma 8.1.2 (Initialization of the back-translation) *Let* S *be a well-formed set of traces. Then there exists* T *a program at level* (1) *such that for any* $m \in \mathsf{S}$, $\mathsf{T} \rightsquigarrow_{(1)} m$.

Lemma 8.1.3 (Correctness of a back-translation step) *Let* T *be a program at level* (i) *where* $i > 0$ *and* (α) *the step from* (i) *to* $(i+1)$*. If* $\mathsf{T} \rightsquigarrow_{(i)} m$ *then* $\mathsf{T}\uparrow_{(\alpha)} \rightsquigarrow_{(i+1)} m$.

We verify each pass after the initialization by proving a simulation. Because each transformation is simple, proving each of these simulations is considerably simpler than proving the entire back-translation correct at once; and while this requires more upfront setup (defining each tree language), we found this technique much easier overall.

Proving these lemmas raises two questions that have to be answered together:

i How do we define the semantics of each intermediate level?
ii How can we prove that the final source program obtained by back-translation is correct? In particular, what kind of *invariants* are needed in this proof?

First, we identify the core invariants that must hold of the generated source code: (i) the location stored in the private state of each compartment must always be up-to-date; and (ii) completeness properties: all events on the trace have a corresponding statement in the back-translated code, all events have a corresponding statement that "consumes" it and updates the state, and all these statements should be reachable when in the right state.

Then, we build up incrementally to the source level by introducing these new invariants in the semantics, little by little. More concretely, the small-step operational semantics of our languages enforce these invariants by adding them as requirements of the small-step transitions. For instance, when we add the abstract cross-compartment stack, the semantics only allows returns if they are indeed returns from and to the compartments recorded on that stack. This enforcement is only a tool for the proof: indeed, we later prove forward simulations between the languages, showing that these invariants do not really restrict the behaviours of programs. Each of these small additions incurs a proof burden, but instead of having to deal with them all at once, our structure allows us to do it one by one, in isolation. Once an invariant has been established in the semantics of one level, the fact that each step is small makes it easy to prove that it is preserved up to the source level. Once we reach the final pass, we only have to reason about the semantics of

the source language, but all of the more complex invariants have already been dealt with in earlier passes. Moreover, all the levels except the very last one are pretty much independent of the source language we target, so we expect one would be able to reuse these parts of the proof fairly easily.

In the following sections, we describe in more detail our methodology to build the semantics of the languages and how we use them in the proofs.

8.2 Design of the nanopass back-translation

8.2.1 The tree languages in the back-translation

Structure and invariants The idea at the core of our definition of all tree languages is the following: to each compartment, we associate one tree representing the set of traces it produces. The type `tree` we use to represent all of our trees is given below in Figure 8.5. It is parameterized by the type `A`, which differs between each tree language and represents the data stored in each node. The branches are labeled by `events`.

```
1 Inductive tree (A : Type) : Type :=
2 | node : (* content of the node *) A ->
3          (* children *) branches A -> tree A
4 with branches (A : Type) : Type :=
5 | Bnil : branches A
6 | Bcons : (* label *) event ->
7           (* tree corresponding to that label *) tree A ->
8           (* rest of the children *) branches A -> branches A
```

Figure 8.5: Basic type used to represent trees

There is an important asymmetry in this setting: while compartments $C \in \mathbb{C}$ (in the context) depend on the entire set of traces, compartments $C \in \mathbb{P}_i$ (in the program) only depend on one of the traces, m_i. What this means is that, for compartments in the program, the tree that is associated to each compartment is entirely linear, and is a simple embedding of the trace in the type of tree (i.e. they look like this: `node (Bcons e1 (node (Bcons e2 ... Bnil)) Bnil)`). Conversely, for compartments in the context, the tree associated to them is branching; yet, this branching can only occur in restricted manners. Indeed, our languages are deterministic: this means that the only way for a tree associated to a compartment of the context to branch is if a compartment of the program behaved differently.

This allows us to define *determinacy invariants* that we use to prove the completeness properties discussed in the previous section. First, we define two determinacy invariants[1] that apply at all levels. The first such invariant is `unique_current_tree` (Figure 8.6): This invariant captures the intuition that, if a compartment has control (line 9), then it can only execute one single next event (captured by `match br' with | Bnil => true | _ => false end`). The rest of the invariant corresponds to the recursive cases: for the subtree `tr` (line 8) and the remaining branches `br'` (line 11).

1: In our Rocq development, we use predicates in either `bool` or `Prop` without distinction, depending on what is more practical.

The second determinacy invariant, `deterministic_tree`, is partially listed in Figure 8.7:

This determinacy invariant captures the intuition that executing the same event always leads to the same state; i.e. there are no two branches at the

```
Fixpoint unique_current_tree {A : Type} (C : Compartment.id) (tr : tree A)
↪  : bool :=
  match tr with
  | node _ br => unique_current_branches C br
  end
with unique_current_branches {A : Type} (C : Compartment.id) (br :
↪  branches A): bool :=
       match br with
       | Bnil => true
       | Bcons e tr br' => (unique_current_tree C tr) &&
             (if cur_comp_of_event e == C then
                match br' with | Bnil => true | _ => false end
              else unique_current_branches C br')
       end.
```

Figure 8.6: Invariant capturing the uniqueness of the next event for the current compartment

```
Inductive deterministic_tree {A : Type} : tree A -> Prop := ...
with deterministic_branches {A : Type} : branches A -> Prop := ...
| deterministic_branches_cons : forall (e : event) tr brs,
    deterministic_branches brs ->
    deterministic_tree tr ->
    (* once we find a given event, it doesn't appear in the rest *)
    not_in e brs ->
    deterministic_branches (Bcons e tr brs)
with not_in {A : Type} : event -> branches A -> Prop := ...
```

Figure 8.7: Invariant capturing the determinacy of trees

same level in the tree that are labeled by the same event. This idea can be seen in the use of a predicate `not_in` (line 7), which enforces that an event cannot appear (anymore) in the rest of the children labels.

Finally, we introduce a third determinacy invariant (that we do not list here), during step (b). In that step, we associate to each node in the trees a location identifier and this determinacy invariant captures the uniqueness of locations. It is very similar to `deterministic_tree`, except that it is about the content of the nodes instead of the labels of the branches.

Semantics As described above, programs in these languages are maps from compartments to such trees. We give a small-step operational semantics to each of the tree languages that works by reducing states. As a common basis, states contain the remaining trace that is to be produced, and, for each compartment, a tree capturing the rest of the execution. Then, the semantics are built around a reduction rule such as the one described in Figure 8.8. The idea of this rule is that one can follow the execution of a given trace by replaying all of its events one by one, each time reducing the trees associated with the two compartments. In this rule, the notation $\mathsf{T}_i \xrightarrow{e} \mathsf{T}'_i$ means that T'_i is a child of T_i through a branch labeled e, and $\mathsf{compartments}(e) = (\mathtt{C_i}, \mathtt{C_j})$ means that the event e is a call or a return from $\mathtt{C_i}$ to $\mathtt{C_j}$.

Moreover, the semantics maintains some amount of *ghost state* gs, which stores information about what happened previously during the execu-

$$\textsc{S-Step}\quad \frac{\begin{array}{c} s.\mathsf{trace} = e :: t \qquad \mathsf{compartments}(e) = (\mathtt{C_i}, \mathtt{C_j}) \\ s.\mathsf{trees}(\mathtt{C_i}) = \mathsf{T}_i \qquad s.\mathsf{trees}(\mathtt{C_j}) = \mathsf{T}_j \qquad \mathsf{T}_i \xrightarrow{e} \mathsf{T}'_i \\ \mathsf{T}_j \xrightarrow{e} \mathsf{T}'_j \qquad s'.\mathsf{trees} = s.\mathsf{trees}[\mathtt{C_i} \mapsto \mathsf{T}'_i, \mathtt{C_j} \mapsto \mathsf{T}'_j] \qquad s'.\mathsf{trace} = t \end{array}}{s \xrightarrow{e} s'}$$

Figure 8.8: Basic reduction rule for trees

tion. This information contains the current location gs.loc (a map from compartments to natural numbers added by step (a)), and an abstract stack gs.stack (added in step (b)) recording cross compartment calls, i.e. a sequence of `(Compartment.id * Compartment.id)`. Then, the semantics contain restrictions on the ghost state that capture the invariants we want to prove for the back-translated program. For instance, below one can see what we added to the rules to capture the well-bracketedness of the stack: calls add frames to the call stack, and returns are only allowed when the top frame of the stack matches that return.

$$\frac{\text{S-Step-Stack-Call}\quad \ldots \qquad \mathsf{gs'.stack} = (\mathsf{C_i}, \mathsf{C_j}) :: \mathsf{gs.stack}}{s, \mathsf{gs} \xrightarrow{\text{Call } \mathsf{C_i}\ \mathsf{C_j}\ \mathsf{z}} s', \mathsf{gs'}}$$

$$\frac{\text{S-Step-Stack-Return}\quad \ldots \qquad \mathsf{gs.stack} = (\mathsf{C_j}, \mathsf{C_i}) :: \mathsf{s} \qquad \mathsf{gs'.stack} = \mathsf{s}}{s, \mathsf{gs} \xrightarrow{\text{Ret } \mathsf{C_i}\ \mathsf{C_j}\ \mathsf{z}} s', \mathsf{gs'}}$$

This ghost state is an abstraction of the concrete state of the generated source code. For instance, the abstract location stored in trees corresponds to the private variable `loc` of each generated compartment, and the abstract stack corresponds to the concrete stack of the source language in the sense that it captures the same cross-compartment behaviours.

We build the semantics iteratively, by taking the semantics of the previous level, and adding the new conditions in addition to the previous ones.

Initialization We do not start from trees, but actually from sets of traces that satisfy a notion of well-formedness. For this reason, the first step of our back-translation is to constructively prove that such sets of traces can be represented as trees satisfying the predicates described previously. To do so, we define `add_trace_to_tree` (Figure 8.9) that adds a trace to a tree, and `tree_of_trace_list` that iterates this function over a sequence of traces. The core of this function is actually `add_trace_to_branch`:

```
1  Fixpoint add_trace_to_tree (tra: trace) (tr: tree unit): tree unit :=
2    match tra with
3    | nil => tr
4    | cons e q =>
5        match tr with
6        | node tt br => node tt (add_trace_to_branch e q br)
7        end
8    end
9  with add_trace_to_branch (e: event) (tra: trace) (br: branches unit): tree
   ↪  unit :=
10   match br with
11   | Bnil => branch_of_trace (e :: tra)
12   | Bcons e' tr br' => if (e == e') then
13                         Bcons e (add_trace_to_tree tra tr) br'
14                       else
15                         Bcons e' tr (add_trace_to_branch e tra br')
16   end.
```

Figure 8.9: Construction of trees from traces

this function takes an event and a trace following this event, traverses the branches until it finds one that is already labeled by this event, and merges the remaining trace with that branch (lines 12-13). If it doesn't find such a branch, it creates a new one (line 11). This ensures there is no duplication between events, as required by `deterministic_tree`.

We do not reproduce our notion of well-formedness here, but it captures the determinacy conditions for proving `unique_current_tree`: that is, if the history is the same and the context has control, then it can't be that two different events appear at this point on the traces.

Tree transformations Steps (b) and (c) are simple tree transformations. Pass (b) adds a unique identifier and pass (c) a stack snapshot to each node. The stack snapshot records what the state is supposed to be when reaching that node, which simplifies the statement of the properties and the reasoning in the simulation proof. These two transformations compile trees by adding new information to the nodes of the trees: pass (b) goes from `tree unit` (level 1) to `tree nat` (level 2); and pass (c) goes from this to `tree (nat * stack)` where `stack = list (Compartment.id * Compartment.id)`. These transformations are simple (they map a transformation over all nodes), and do preserve the determinacy invariants described previously. The more involved transformations are described in the next subsection.

8.2.2 Flattening and code generation

The trees used so far cannot be easily mapped to source compartments in one go, as described in subsection 8.1.2. Instead, we go through two steps, (d) and (e), that manipulate a flattened representation of the trees, in order to generate source statements.

Flattening Step (d) flattens trees into a list of rules describing how compartments should update their location when they get control, and what event they need to produce at a given location. Concretely, a program at level (4) is a map from compartments to `list (nat * event * nat)`. Elements of such a list are rules on how to update location and what events to emit. As an additional invariant, these lists of rules satisfy uniqueness and determinism properties that derive from the uniqueness and determinism properties of the tree languages. For example, consider a call Call $\mathtt{C}_1$ $\mathtt{C}_2$ 40 that appears in a trace. Then:

- the list corresponding to $\mathtt{C}_1$ will contain some rule $(n_1, \text{Call } \mathtt{C}_1\ \mathtt{C}_2\ 40, n_1')$ meaning that when $\mathtt{C}_1$ is in location n_1, then it must update its location to n_1' and call $\mathtt{C}_2$ with argument 40. Additionally, this list will not contain any other rule starting at n_1 (because $\mathtt{C}_1$ has control, it can only do one next event);
- the list corresponding to $\mathtt{C}_2$ will contain some rule $(n_2, \text{Call } \mathtt{C}_1\ \mathtt{C}_2\ 40, n_2')$ meaning that when $\mathtt{C}_2$ is called with argument 40 and its current location is n_2, it must update its location to n_2'. Additionally, the list may contain other rules starting at n_2, however none of those will also be for a call with the specific argument 40.

We give a semantics to this language in the same style as for the trees. The main difference is that a step can be taken iff a corresponding rule appears in the mapping for each compartment. Unlike the tree representation, the list of rules is *not* reduced during execution: this is similar to the source semantics in that the source code of a source compartment does not "disappear" when it is executed. Finally, the ghost state still keeps track of the abstract call stack.

Code generation The code generation step (e) transforms the previous lists of rules to executable source code. For each rule, this step generates two statements: one for the compartment giving up control, and the other for the one gaining control. The statements of each compartment are then concatenated so as to obtain source code akin to Figure 8.2, by filtering and folding over the list of rules and statements.

8.3 Proving the back-translation correct using forward simulations

In this section, we discuss the core of our proof: how we prove forward simulations between all the tree languages to obtain a correctness lemma for each of these steps.

8.3.1 Forward simulations

To prove Lemma 8.1.3 between tree languages, we use *forward simulations* in the style of CompCert. For each step α between levels (i) and $(i+1)$, we define a simulation relation $\sim_{(\alpha)}$ between the states of two consecutive languages. Then, we prove the following lemmas:

Lemma 8.3.1 (Initial states are related) *Let s_i be an* initial *state at level i and s_{i+1} an* initial *state at level (i+1), then $s_i \sim_{(\alpha)} s_{i+1}$.*

Lemma 8.3.2 (Forward simulation) *Let s_i be a state at level i and s_{i+1} a state at level (i+1).*
Suppose $s_i \sim_{(\alpha)} s_{i+1}$ and $s_i \xrightarrow{e}_i s'_i$. Then there exists a state s'_{i+1} at level (i+1) such that:

$$s_{i+1} \xrightarrow{e}{}^*_{i+1} s'_{i+1} \qquad \textit{and} \qquad s'_i \sim_\alpha s'_{i+1}$$

It is well known [22, 23] that, together, these two lemmas imply Lemma 8.1.3.

[22]: Leroy (2009), 'A Formally Verified Compiler Back-end'
[23]: Tan et al. (2019), 'The verified CakeML compiler backend'

8.3.2 Simulation relations and proofs

We now describe how to define these simulation relations for each pass.

Simulations between tree languages Recall that for the tree languages, states contain a map from compartment identifier to trees storing what is left to be executed. For passes between such tree languages, the basic idea is simple, and can be summarized as follows: we always maintain one essential property: if s_i is a state at level i and s_{i+1} a state at level $i+1$ that ought to be related by the simulation relation, then for any compartment $\mathtt{C_i}$, $s_{i+1}.\text{trees}(\mathtt{C_i}) = f(s_i.\text{trees}(\mathtt{C_i}))$ where f is the transformation applied by the pass. In addition, when we introduce the abstract stack, the simulation relation captures the fact that this stack is always well-bracketed. We use the predicate in Figure 8.10 in the simulation relation between the states at level 2 and 3. This predicate relates the trace remaining to be executed and the current stack: if the next event is a call (line 4), then the rest of the trace is related to the same stack, with a new frame corresponding to the call added. On the other hand, if the next event is a return (line 5), then the stack must have a top

```
Fixpoint wf_stack_trace (tra: trace) (st: stack): bool :=
  match tra with
  | [] => true
  | ECall Ccur P z Cnext :: tra' => wf_stack_trace tra' ((Ccur, P, Cnext)
  ↪  :: st)
  | ERet Ccur z Cnext :: tra' =>
      match st with
      | (Cnext', _, Ccur') :: st' =>
          (Ccur == Ccur') && (Cnext == Cnext') && wf_stack_trace tra' st'
      | _ => false
      end
  end.
```

Figure 8.10: Relation for the stack used in the simulation proof between levels 2 and 3

stackframe that corresponds to this return and the rest of the trace must be related to the rest of the stack. We use this predicate in the simulation to prove the well-bracketedness of the stack required by the semantics.

Note that we don't need to manipulate this predicate after step (c), because after that the semantics of the languages enforce the well-bracketedness of the stack, as in subsection 8.2.1.

Flattening Recall that after flattening, we do not reduce the list of rules anymore; hence, the list of rules in the flattened representation is a superset of the trees from the previous level. Instead, the invariant we maintain states the following: any rule that can be derived from the source trees in the current state can be found in the list of rules we keep.

In addition, we prove the determinacy invariants stated in subsection 8.2.2 explicitly from the start, and make it part of a notion of well-formedness of our programs. We use this to prove the notion of uniqueness from Figure 8.11: This says that in the list of incoming calls for compartment `C`,

```
Definition uniqueness_in {A B: Type} (l: list (B * A)) :=
  forall b a a', In (b, a) l -> In (b, a') l -> a = a'.

Definition unique_incoming_calls: forall C es,
  prog_rules p C = Some es ->
  uniqueness_in (incoming_calls C es).
```

Figure 8.11: Uniqueness for the flattened representation

there is at most one rule for each key, and here keys are of type `(nat * Z)`: the natural number corresponding to the location, and the integer argument for this rule. We have similar definitions for incoming returns and for outgoing events. Proving that the properties of the trees imply these uniqueness properties is quite involved, but only because of a complex induction on trees.

Code generation Step (e) goes from the flattened representation to source code and could have been the most complex to reason about. However, since the flattened representation is very close in its structure to the actual code produced by the back-translation, most of the reasoning to prove this step correct has already been done. This proof step goes as follows:

1. We define the simulation invariant that holds at each synchronization point of the simulation (i.e. when we have to prove the two states are related).
2. We prove individual properties of the back-translations of each rule: they produce the right event, they update the location correctly, etc.

3. We prove that the generated code chooses the right branches in the branching structure, using the determinism and uniqueness properties of the flattened representation.
4. We assemble the full programs, and prove the forward simulations, using all the above properties and the determinism and uniqueness properties of the flattened representation.

The core of our simulation invariants is reproduced in Figure 8.12. This simulation invariant states that: two states are related if the con-

```
1  Inductive match_concrete_stacks: Component.id -> stack -> CS.stack -> Prop
   ↪  :=
2  | match_concrete_stacks_nil: forall C, match_concrete_stacks C [] []
3  | match_concrete_stacks_int: forall C s st old_arg,
4      match_concrete_stacks C s st ->
5      match_concrete_stacks C s (CS.Frame C old_arg Kstop :: st)
6  | match_concrete_stacks_ext: forall C C' s st old_arg,
7      C <> C' ->
8      match_concrete_stacks C' s st ->
9      match_concrete_stacks C
10       ((C', C) :: s)
11       (CS.Frame C' old_arg (Kafter_call C' 0) :: st) .
12
13 Definition match_mem (locs: NMap nat) (mem: Memory.t) :=
14   forall C n, locs C = Some n ->
15     Memory.load mem (C, Block.local, 0%Z) = Some (Int (Z.of_nat n))
16     /\ Memory.load mem (C, Block.local, 1%Z) = Some (Int 1%Z)
17     /\ (exists v, Memory.load mem (C, Block.local, 2%Z) = Some v).
18
19 Inductive match_states (p: Flattened.prg) (i: nat): Flattened.state ->
   ↪  CS.state -> Prop :=
20 | match_states_cons: forall t locs st C cst mem e v es,
21     forall (MS_ST: match_concrete_stacks C st cst)
22       (MS_MEM: match_mem locs mem)
23       (ES: Flattened.genv_code (globalenv (Flattened.sem p)) C = Some es),
24       e = (switch_outgoing C es E_exit) ->
25       match_states p i (t, locs, st) (CS.State C cst mem Kstop e v) .
```

Figure 8.12: Flattened to source simulation invariant

crete program is currently executing a compartment's code (lines 23-24: `switch_outgoing` is a function that unfolds to the code displayed in the second half of the code in Figure 8.2); the abstract and the concrete stack are related; and the memory in the source state is related to the map of locations in the flattened state. The stacks are related (`match_concrete_stacks`) when they represent the same cross-compartment call-stack. The rule `match_concrete_stacks_int` allows the source to perform internal calls, which are then unfolded during the simulation proof when returning. The memory must agree with the location map (line 15), and the other variables must be accessible.

While the proof of this pass is the longest in our development, it consists mostly in step-by-step reasoning about the behaviour of source statements, which is made much easier because all the harder invariants have already been established in previous steps.

8.4 Formalization in Rocq

We have implemented and verified this back-translation in the Rocq proof assistant. Starting from a well-formed set of traces, we define each step of the back-translation from (a) to (e) and the semantics of each intermediate language. We prove an initialization lemma converting well-formed sets of traces to trees, and for each remaining step we prove a CompCert-style forward simulation result, following the ideas described in the

previous sections. Our Rocq development is, however, done in a slightly more complex setting [8] than the simplified one we used for aiding the exposition above. In the Rocq development we consider multiple procedures per compartment, that are mediated through an interface, which is interesting, since the compartments are able to distinguish when receiving calls to different procedures. Therefore, we must generate slightly different statements for handling calls for each procedure of a compartment. We do this by recording the procedure being called in the call events.

[8]: Abate et al. (2018), 'When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise'

The development is ~7800 lines of Rocq code long (comments excluded), with 1200 lines of specifications and 6600 of proofs. This figure includes the definition of the source language, but neither that of the trace model, nor that of some libraries and existing files that we reuse from Abate et al. [8]'s development. The largest part of the development is spent reasoning about the determinacy invariants, in particular proving that they imply the uniqueness from Figure 8.11. This part should be fully reusable with minor adaptation, because it takes place between levels 3 and 4, before we start reasoning about the source language.

Benefits of our approach We found approaching the proof in nanopass style beneficial, since it allowed us to partition proof concerns relatively easily. In particular, we were able to parallelize the work between the first steps and the latter steps without too much effort, because these steps were not directly related. On the contrary, when we previously attempted a monolithic proof, any modification at any level of the proof did affect the entire development and made the verification of the many invariants complex and more time-consuming.

Discussion

9 Related Work

9.1 Compiler Correctness and CompCert

The study of compilers and their correctness can be traced back to the 1960s, when McCarthy and Painter [20] noticed that compilers need to be proven correct; they proceed with the first proof of correctness of a compiler, for compiling arithmetic expressions. Soon after, in 1973, Morris proposes [21] the modern approach to structuring proofs of compiler correctness: the source and target languages should be given semantics, and the semantics of the compiled program should be "decodable" to retrieve the semantics of the source program. While Morris doesn't go into more details concerning what "decodable" should mean in the general case, this is essentially the notion of "refinement": all behaviours of the compiled program should also be behaviours of the source program.

[20]: McCarthy et al. (1967), *Correctness of a compiler for arithmetic expressions*

[21]: Morris (1973), 'Advice on structuring compilers and proving them correct'

While research in compiler correctness continues to thrive [57], it is not before the mid 2000s that the next major achievement in compiler correctness is reached: CompCert [58]. CompCert shows for the first time that it is indeed possible to build a complex optimizing compiler for a realistic language, C, and to verify it in a proof assistant. The theorem proved in CompCert is not unlike that of Morris [21], and, informally, states that every behaviour of a compiled program is also a behaviour of its source. To prove it, CompCert uses a proof structure using a uniform memory model (the CompCert memory model), and proof steps using simulations, which became a basis for proving compiler correctness in various settings.

[57]: Dave (2003), 'Compiler verification: a bibliography'

[58]: Leroy (2006), 'Formal certification of a compiler back-end or: programming a compiler with a proof assistant'

[21]: Morris (1973), 'Advice on structuring compilers and proving them correct'

Following CompCert come new verified compiler projects, the most well-known being CakeML [23], many extensions [59–61], and many tools and frameworks building on top of it [62, 63].

[23]: Tan et al. (2019), 'The verified CakeML compiler backend'

[59]: Ševčík et al. (2013), 'CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency'

[60]: Besson et al. (2018), 'CompCertS: A Memory-Aware Verified C Compiler Using a Pointer as Integer Semantics'

[61]: Kim et al. (2025), 'Archmage and CompCertCast: End-to-End Verification Supporting Integer-Pointer Casting'

However, despite this success, CompCert in its first iteration suffers from some limitations that prevent its use in many domains. One main area of focus is that it lacks any form of compositionality: CompCert (version 1) only deals with whole programs that can be run on their own; CompCert gives no guarantee to partial programs that still need to be linked with other partial programs before being run. It is hence no surprise that a multitude of works come and claim to solve the problem of compositionality of CompCert. We refer the interested reader to [64], which gives a detailed overview of the various attempts at solving compositional compiler correctness proposed until 2019.

[62]: Gu et al. (2015), 'Deep Specifications and Certified Abstraction Layers'

[63]: Appel (2012), 'Verified Software Toolchain'

[64]: Patterson et al. (2019), 'The next 700 compiler correctness theorems (functional pearl)'

9.2 Secure Compilation

9.2.1 Full Abstraction and Proof Techniques

Full abstraction is an important concept in the study of programming languages that was first developed in the context of denotational semantics [65–67]. Then, the goal was to give "faithful" denotational semantics to programming languages, in the sense that they capture exactly the differences in the operational semantics. At this time, the concern was not yet on compilation or security.

[65]: Milner (1975), 'Processes: A Mathematical Model of Computing Agents'
[66]: Milner (1977), 'Fully abstract models of typed λ-calculi'
[67]: Plotkin (1977), 'LCF considered as a programming language'

The idea of using full abstraction for security can be traced back to Abadi [53], who propose the use of full abstraction to characterize what it means for a compiler to be secure. He argues, quite informally, that the forward direction of full abstraction (equivalent programs are mapped by the compiler to equivalent programs) captures that there are no information leaks introduced by the compiler. He then proceeds to show how full abstraction can be used to capture the (absence of) security of two transformations: one for compiling Java to Java bytecode (expanded upon in [68]), and one for compiling π-calculus, more precisely implementing secure private channels as encrypted communication over public channels (expanded upon in [69]).

[53]: Abadi (1999), 'Protection in Programming-Language Translations'

[68]: Stata et al. (1998), 'A Type System for Java Bytecode Subroutines'

[69]: Abadi et al. (), 'Secure implementation of channel abstractions'

Since then, full abstraction has been considered the gold standard for secure compilation, and it has been used in many works, either to assert the security of a transformation ([33, 40, 43, 45, 70–75]), or to identify security vulnerabilities ([76]).

[33]: New et al. (2016), 'Fully abstract compilation via universal embedding'
[40]: Jeffrey et al. (2005), 'Java JR: Fully Abstract Trace Semantics for a Core Java Language'
[43]: Patrignani et al. (2015), 'Fully abstract trace semantics for protected module architectures'
[45]: El-Korashy et al. (2021), 'CapablePtrs: Securely Compiling Partial Programs Using the Pointers-as-Capabilities Principle'
[70]: Fournet et al. (2013), 'Fully abstract compilation to JavaScript'
[71]: Ahmed et al. (2011), 'An equivalence-preserving CPS translation via multi-language semantics'
[72]: Agten et al. (2012), 'Secure Compilation to Modern Processors'
[73]: Ahmed et al. (2008), 'Typed closure conversion preserves observational equivalence'
[74]: Devriese et al. (2017), 'Modular, Fully-abstract Compilation by Approximate Back-translation'
[75]: Busi et al. (2021), 'Securing Interruptible Enclaved Execution on Small Microprocessors'
[76]: Kennedy (2006), 'Securing the .NET programming model'

An interesting aspect of these works is the proof technique they use. In general, proving full abstraction requires back-translating a low-level context that distinguishes two compiled programs into a source one that distinguishes the source programs. Back-translation techniques can then be broadly categorized in two ways: *trace-based backtranslation* and *code-based back-translation*, similarly to section 2.2.

Trace-based back-translation works in a similar way as the back-translations we presented in this thesis. These back-translations typically consider a trace (often finite), representing the common execution prefix after which the traces of the two executions start being different. They then construct a context based on this trace that will do two different things depending on which program it is linked with. Examples of this kind of back-translation include [40, 43, 45, 72, 75].

On the other hand, code-based back-translation attempts to construct a source context based simply on the target context's code. This is often made possible by the fact that the source language is not actually more restrictive than the target [33, 70, 71]. One exception is the work by Devriese et al. [74]. There, they only build back-translated contexts that can emulate the target contexts for an arbitrary but finite number of steps. However, since termination, by definition, occurs in a finite number of steps, this is enough to detect the difference in behaviour, and thus can they prove full abstraction.

Full Abstraction for Identity Compilers Finally, recent years have seen the use of full abstraction for identity compilers. In that context, the goal of the full abstraction theorem is to argue that two semantics, one more concrete than the other, are exactly as secure: typically, the

abstract semantics satisfy some properties, and the goal is to show that the concrete semantics also satisfy these properties.

For instance, some recent work uses semantics called "overlay semantics". There, the compilation step goes from an *overlay semantics*, a semantics that is secure *by construction*, to a more concrete semantics ("regular semantics") where the security comes from actual enforcement by actual features. In this setting, the overlay semantics is usually constructed to be as close as possible to the regular semantics, but with more ghost state to enforce invariants.

Skorstengaard, Devriese, and Birkedal [77] design a calling convention based on linear capabilities[1] that guarantees two properties, *local state encapsulation*[2] and *well-bracketed control flow*. To prove that their calling convention does what they want, they build, on top of their assembly language's semantics (regular semantics), an overlay semantics that guarantees by construction the two properties. The overlay semantics is built by taking the regular semantics and modifying it so that the properties are guaranteed, using a built-in abstract stack and replacing concrete objects such as linear capabilities with abstract objects such as so-called stack capabilities that can only be used to manipulate this abstract stack. They then prove full abstraction between the overlay semantics and the regular semantics, using a logical relation-based proof technique to relate the behaviours of the same program in the two semantics. Interestingly, they do not use a back-translation, because their contexts are syntactically the same. Georges, Trieu, and Birkedal [78] follow the same approach, but this time considering *stack safety*, adding to the properties of local state encapsulation and well-bracketed control flow the new property of *temporal stack safety*, which basically prevents access to former stack frames (stack frames for calls that have already been returned from). They can again prove their result directly, for the same reason as Skorstengaard, Devriese, and Birkedal [77].

[77]: Skorstengaard et al. (2019), 'StkTokens: enforcing well-bracketed control flow and stack encapsulation using linear capabilities'

1: capabilities that cannot be duplicated by the hardware

2: basically the secrecy and integrity of a caller's private variables

[78]: Georges et al. (2022), 'Le temps des cerises: efficient temporal stack safety on capability machines using directed capabilities'

In another setting, Busi et al. [75] also use full abstraction to argue for the security of an enclave on a microprocessor, with interrupts. More precisely, they prove a full abstraction result between the semantics of a microprocessor with an enclave without interrupts, and the same semantics with an additional interrupt mechanism that is in control of the adversary. However, they do not prove full abstraction directly, instead relying on a trace-based back-translation; that is, they show that attackers exploiting the interrupt mechanism can be simulated by attackers that do not use it.

[75]: Busi et al. (2021), 'Securing Interruptible Enclaved Execution on Small Microprocessors'

9.2.2 Other Kinds of Secure Compilers

Even if we ignore chapter 2 and the work of Abate et al. [6], other characterizations of what secure compilation means have obviously been proposed and applied.

Patrignani and Garg [52] propose *robustly safe compilation*, a concept that is not too different from the criteria described in chapter 2. Just like RSP, Robustly safe compilers map source programs satisfying safety properties robustly[3] to target programs satisfying the same safety properties robustly. The difference lies in the fact that Patrignani and Garg [52] allow for a translation of the safety property between the two languages.[4] They also show how different enforcement mechanisms and proof techniques, some simpler and some more complex, can be used to build and prove compilers secure according to this criterion; they compare robustly safe

[6]: Abate et al. (2019), 'Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation' **[Co-authored]**

[52]: Patrignani et al. (2021), 'Robustly Safe Compilation, an Efficient Form of Secure Compilation'

3: Against any adversarial context.

4: See also [4] for an account of how different representations can be accounted for in correct and secure compilation criteria.

compilation to full abstraction, and reach basically the same conclusions as us (chapter 2).

Patrignani and Garg [46], again, propose *trace-preserving compilation*, a criterion that implies the robust preservation of all hyperproperties and is stronger than full abstraction, in the setting of reactive programs. The idea behind trace-preserving compilation is that compiled programs should be able to output at least the same traces as the source; in addition, if they receive invalid input (inputs that aren't related to anything in the source), they should respond by outputting values that aren't related to anything in the source as well. They remark that the easiest way to obtain a trace-preserving compiler is to take a correct compiler, and add a wrapper that detects invalid inputs and reacts by either terminating the program or ignoring the input.

[46]: Patrignani et al. (2017), 'Secure Compilation and Hyperproperty Preservation'

Another line of work is compilation that preserves particular security properties, such as constant-time [28, 79]. Operational semantics are instrumented with a leakage model that captures the information that leaks at each step of execution. Using this, the goal is to prove that compilation preserves leakage equivalence, which directly implies the preservation of noninterference in that setting. This is done by proving that there exists a "constant-time simulation" between source and target, often depicted as a cube diagram: two source states that leak equally are simulated by two target states that also leak equally. This technique can be used to prove that CompCert preserves leakage equivalence [80].

[28]: Barthe et al. (2018), 'Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time"'
[79]: Almeida et al. (2017), 'Jasmin: High-Assurance and High-Speed Cryptography'
[80]: Barthe et al. (2019), 'Formal verification of a constant-time preserving C compiler'

9.3 Formalization of Secure Compilers

Formally verified secure compilers are few and far between.

The first kind of formalized compilers are the ones that show secure compilation criterion such as the ones described in chapter 2, including full abstraction. Devriese et al. [74] prove secure a compiler between a simply-typed lambda calculus and an untyped one. To work around the absence of universal type in the source, they develop a notion of "approximate back-translation" that only simulates the target context for a finite number of steps. Devriese, Martin, and Patrignani [81] give three secure compilation proofs using the same technique, in order to study the expressive power of some lambda calculi. Interestingly, these approximate back-translations are similar to our own trace-based back-translations, because both kinds of back-translations can only emulate the target for a finite number of steps.

[74]: Devriese et al. (2017), 'Modular, Fully-abstract Compilation by Approximate Back-translation'
[81]: Devriese et al. (2024), 'On the Semantic Expressiveness of Iso- and Equi-Recursive Types'

For other mechanized works in the field of secure compilation, one has to look at other notions of secure compilation. For instance, Besson et al. [82] verify a sandboxing mechanism using source-level software fault isolation for CompCert. The works discussed in the previous section related to the preservation of cryptographic constant time [28, 79, 80] were also mechanized.

[82]: Besson et al. (2019), 'Compiling Sandboxes: Formally Verified Software Fault Isolation'
[28]: Barthe et al. (2018), 'Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time"'
[79]: Almeida et al. (2017), 'Jasmin: High-Assurance and High-Speed Cryptography'
[80]: Barthe et al. (2019), 'Formal verification of a constant-time preserving C compiler'

10 Conclusion and Future Work

In this thesis, we set out to improve the world of secure compilation by designing novel proof techniques that simplify the work of proving the security of a compiler in a proof assistant.

In chapter 2, we observed that *full abstraction*, the gold-standard secure compilation criterion, is inadequate for our goal of building a formally verified secure compiler, since its security guarantees aren't clear, and it is often too difficult to enforce and too difficult to prove. Following these observations, we described a large class of security criteria, and we showed with an example, how some security criteria are simpler to enforce and prove, and more prone to formal verification.

After summarizing prior work on defining and proving an adequate security criterion for a compiler in the presence of undefined behaviour and compartmentalization (chapter 3), we presented our first contribution (chapter 4): *recomposition*, a novel step that replaces the use of partial semantics in secure compilation proofs. The benefit is huge, as partial semantics are complex and difficult to reason about, and it showed in our implementation, with the new step being three times smaller than the ones it replaced.

We then described SECOMP (chapter 5): a secure compartmentalizing compilation chain for C that allows programmers to annotate their code with compartment information and reason soundly about the security of their programs, even in the presence of foreign and potentially adversarial code, and undefined behaviours. SECOMP is implemented in the Rocq proof assistant, as an extension of the already existing CompCert C compiler, and we discuss our design decisions as well as how we were able to update the compiler correctness proof with little changes.

Then, in chapter 6, we discussed how we proved SECOMP secure. Reusing the technique from (chapter 3), we had to innovate in terms of proof engineering and proof technique in order to be able to scale up to a realistic language, in particular with a new structure for the recomposition proof using CompCert-style simulation diagrams.

Finally, in the last part of this thesis, we described two additional improvements and extensions. First, we showed that the compiler from chapter 3 can be extended to support *memory sharing* (chapter 7), something that is still lacking in SECOMP. We prove the compiler secure with novel *data-flow traces*, traces that record fine-grained memory operations as they happen during the execution. Then, we showed one can obtain much stronger criteria while keeping the bulk of the proof the same (chapter 8), the idea being to structure the back-translation as a *nanopass compiler*, i.e., a series of small steps instead of a massive single transformation.

Future work We do not claim to have solved secure compilation. On the contrary, there are still many research directions that can be taken in the field of secure compilation.

First, SECOMP is still incomplete. SECOMP has only been evaluated on very simple examples in practice, and a major obstacle to realism is the current inability for compartments to communicate beyond passing scalar data. The ideal solution would be to implement the memory sharing facilities that we presented in chapter 7. However, the proof in that

work is done in a much simpler setting; and yet, it is already extremely long and complicated. Small subtleties can often make implementing and proving a secure compiler much harder, and so adding memory sharing to SECOMP would likely be a considerable amount of work, and would probably necessitate the creation of new techniques and new methodologies. In fact, current CompCert doesn't even support passing arbitrary pointers to system calls, so one would first have to significantly change the trace model to allow such behaviour, and fix all the correctness proofs first.

The other extension presented in this thesis, in chapter 8, would also be a welcome addition to SECOMP, as it would result in a much stronger criterion that captures hypersafety properties, and in particular noninterference. While this seems more reasonable than allowing memory sharing, it is still unclear how difficult such extension would be in practice.

Another area of possible improvement is the backend: currently, SECOMP's backend is still relatively high-level. Its memory model is still CompCert's block-based memory model, but a more realistic memory model for such a language would be a flat memory model. It is unclear how one can go from a block-based memory model to such a flat memory-model in a security proof; indeed, our technique would likely fail, as it uses recomposition, which is very sensitive to observations in the target language. For instance, if a context can observe the size of the code, recomposition will not hold.

Many different compositional compiler correctness theorems have been proposed for CompCert-like compilers (chapter 9). So far, they do not allow linking C code with arbitrary assembly code. One line of work that would be interesting to explore is how one can use these compositional correctness results to build secure compilers.

Beyond SECOMP, a very interesting future work would be the enforcement of the robust preservation of hypersafety in the presence of side-channels, including micro-architectural attacks such as Spectre [83]. Such side-channels completely annihilate the hope for proving recomposition, because they immediately leak information about the code being executed, so new techniques are necessary.

[83]: Kocher et al. (2019), 'Spectre Attacks: Exploiting Speculative Execution'

Finally, there is still a large gap to bridge before we can hope that the use of secure compilers becomes more common. In particular, for compartmentalization, a major issue is that of compartmentalizing existing software; it is unlikely that one can easily find nice interfaces along which to split legacy software into compartments, when this legacy software was not written with compartmentalization in mind. It is possible that the future is not in large, general purpose secure compilers like SECOMP, but instead in the design of smaller, purpose-specific secure compilers, for instance compilers that are good at securely compiling cryptographic code, in the presence of some side-channels.

We hope that our work in this thesis will encourage people to improve upon our techniques and design and verify their own secure compilers.

Bibliography

[1] **Jérémy Thibault**, Joseph Lenormand, and Catalin Hritcu. 'Nanopass Back-Translation of Call-Return Trees for Mechanized Secure Compilation Proofs'. In: *16th International Conference on Interactive Theorem Proving, ITP 2025, September 28 to October 1, 2025, Reykjavik, Iceland*. Ed. by Yannick Forster and Chantal Keller. Vol. 352. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025, 29:1–29:20. DOI: 10.4230/LIPICS.ITP.2025.29 (cited on pages 7, 11, 79).

[2] **Jérémy Thibault**, Roberto Blanco, Dongjae Lee, Sven Argo, Arthur Azevedo de Amorim, Aïna Linn Georges, Cătălin Hriţcu, and Andrew Tolmach. 'SECOMP: Formally Secure Compilation of Compartmentalized C Programs'. In: *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. CCS '24. ACM, Dec. 2024, pp. 1061–1075. DOI: 10.1145/3658644.3670288 (cited on pages 7, 11, 33, 49, 58, 79).

[3] Akram El-Korashy, Roberto Blanco, **Jérémy Thibault**, Adrien Durier, Deepak Garg, and Cătălin Hriţcu. 'SecurePtrs: Proving Secure Compilation with Data-Flow Back-Translation and Turn-Taking Simulation'. In: *2022 IEEE 35th Computer Security Foundations Symposium (CSF)*. IEEE, Aug. 2022, pp. 64–79. DOI: 10.1109/csf54842.2022.9919680 (cited on pages 7, 58, 63, 68, 71, 72, 76).

[4] Carmine Abate, Roberto Blanco, Ştefan Ciobâcă, Adrien Durier, Deepak Garg, Catalin Hritcu, Marco Patrignani, Éric Tanter, and **Jérémy Thibault**. 'An Extended Account of Trace-relating Compiler Correctness and Secure Compilation'. In: *ACM Trans. Program. Lang. Syst.* 43.4 (2021), 14:1–14:48. DOI: 10.1145/3460860 (cited on page 96).

[5] Carmine Abate, Roberto Blanco, Ștefan Ciobâcă, Adrien Durier, Deepak Garg, Cătălin Hriţcu, Marco Patrignani, Éric Tanter, and **Jérémy Thibault**. 'Trace-Relating Compiler Correctness and Secure Compilation'. In: *Programming Languages and Systems*. Springer International Publishing, 2020, pp. 1–28. DOI: 10.1007/978-3-030-44914-8_1.

[6] 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: *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. IEEE, June 2019. DOI: 10.1109/csf.2019.00025 (cited on pages 6, 14, 26, 27, 32, 46, 80, 96).

[7] Carmine Abate, Arthur Azevedo de Amorim, Roberto Blanco, Ana Nora Evans, Guglielmo Fachini, Catalin Hritcu, Théo Laurent, Benjamin C. Pierce, Marco Stronati, **Jérémy Thibault**, and Andrew Tolmach. *When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise*. Extended version of a paper previously published at CCS'18. Updates the proof technique to use *recomposition*. 2019. URL: https://arxiv.org/abs/1802.00588 (cited on pages 7, 43, 58, 71).

[8] Carmine Abate, Arthur Azevedo de Amorim, Roberto Blanco, Ana Nora Evans, Guglielmo Fachini, Catalin Hritcu, Théo Laurent, Benjamin C. Pierce, Marco Stronati, and Andrew Tolmach. 'When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise'. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. ACM, Oct. 2018. DOI: 10.1145/3243734.3243745 (cited on pages 5–7, 33–35, 37–41, 43, 46, 49, 50, 59, 68, 71, 72, 74, 76, 77, 79–81, 92).

[9] Cameron Boozarjomehri. *Rapidly Leveling up Firefox Security*. Archived version: https://archive.is/hqIeu. 2024. URL: https://blog.mozilla.org/security/2024/04/04/rapidly-leveling-up-firefox-security/ (cited on page 2).

[10] Aleph One. 'Smashing the Stack for Fun and Profit'. In: *Phrack* 7.49 (Nov. 1996) (cited on page 2).

[11] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. 'The Matter of Heartbleed'. In: *Proceedings of the 2014 Conference on Internet Measurement Conference*. IMC '14. ACM, Nov. 2014. DOI: 10.1145/2663716.2663755 (cited on page 2).

[12] Chromium Security Team. *Memory safety in Chromium*. Archived version: https://archive.is/obxm8. 2022. URL: https://www.chromium.org/Home/chromium-security/memory-safety/ (cited on page 2).

[13] Catalin Cimpanu. *Microsoft: 70 percent of all security bugs are memory safety issues*. Archived version: https://archive.is/I5vda. 2019. URL: https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/ (cited on page 2).

[14] Thomas Claburn. *Malicious SSH backdoor sneaks into xz, Linux world's data compression library*. Archived version: https://archive.is/RMeY3. 2024. URL: https://www.theregister.com/2024/03/29/malicious_backdoor_xz/ (cited on page 3).

[15] Douglas Kilpatrick. 'Privman: A Library for Partitioning Applications'. In: *Proceedings of the FREENIX Track: 2003 USENIX Annual Technical Conference, June 9-14, 2003, San Antonio, Texas, USA*. USENIX, 2003, pp. 273–284 (cited on page 5).

[16] Charles Reis and Steven D. Gribble. 'Isolating web programs in modern browser architectures'. In: *Proceedings of the 4th ACM European conference on Computer systems*. EuroSys '09. ACM, Apr. 2009, pp. 219–232. DOI: 10.1145/1519065.1519090 (cited on page 5).

[17] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 'Efficient software-based fault isolation'. In: *Proceedings of the fourteenth ACM symposium on Operating systems principles - SOSP '93*. SOSP '93. ACM Press, 1993, pp. 203–216. DOI: 10.1145/168619.168635 (cited on page 5).

[18] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 'Native Client: A Sandbox for Portable, Untrusted x86 Native Code'. In: *2009 30th IEEE Symposium on Security and Privacy*. IEEE, May 2009, pp. 79–93. DOI: 10.1109/sp.2009.25 (cited on page 5).

[19] Andreas Rossberg, Ben L. Titzer, Andreas Haas, Derek L. Schuff, Dan Gohman, Luke Wagner, Alon Zakai, J. F. Bastien, and Michael Holman. 'Bringing the web up to speed with WebAssembly'. In: *Communications of the ACM* 61.12 (Nov. 2018), pp. 107–115. DOI: 10.1145/3282510 (cited on page 5).

[20] John McCarthy and James Painter. *Correctness of a compiler for arithmetic expressions*. 1967. DOI: 10.1090/psapm/019/0242403. URL: http://dx.doi.org/10.1090/psapm/019/0242403 (cited on pages 6, 94).

[21] F. Lockwood Morris. 'Advice on structuring compilers and proving them correct'. In: *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '73*. POPL '73. ACM Press, 1973, pp. 144–152. DOI: 10.1145/512927.512941 (cited on pages 6, 94).

[22] Xavier Leroy. 'A Formally Verified Compiler Back-end'. In: *Journal of Automated Reasoning* 43.4 (Nov. 2009), pp. 363–446. DOI: 10.1007/s10817-009-9155-4 (cited on pages 6, 16, 37, 38, 66, 79, 89).

[23] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. 'The verified CakeML compiler backend'. In: *Journal of Functional Programming* 29 (2019). DOI: 10.1017/S0956796818000229 (cited on pages 6, 79, 89, 94).

[24] Leslie Lamport and Fred B. Schneider. 'Formal Foundation for Specification and Verification'. In: *Distributed Systems: Methods and Tools for Specification, An Advanced Course, April 3-12, 1984 and April 16-25, 1985, Munich, Germany*. Ed. by Mack W. Alford, Jean-Pierre Ansart, Günter Hommel, Leslie Lamport, Barbara Liskov, Geoff P. Mullery, and Fred B. Schneider. Vol. 190. Lecture Notes in Computer Science. Springer, 1984, pp. 203–285. DOI: 10.1007/3-540-15216-4_15 (cited on pages 16–18, 37).

[25] Bowen Alpern and Fred B. Schneider. 'Defining liveness'. In: *Information Processing Letters* 21.4 (Oct. 1985), pp. 181–185. DOI: 10.1016/0020-0190(85)90056-0 (cited on page 16).

[26] Michael R. Clarkson and Fred B. Schneider. 'Hyperproperties'. In: *2008 21st IEEE Computer Security Foundations Symposium*. IEEE, 2008, pp. 51–65. DOI: 10.1109/csf.2008.7 (cited on pages 16, 19, 80).

[27] J. A. Goguen and J. Meseguer. 'Security Policies and Security Models'. In: *1982 IEEE Symposium on Security and Privacy*. IEEE, Apr. 1982. DOI: 10.1109/sp.1982.10014 (cited on pages 16, 19, 78).

[28] Gilles Barthe, Benjamin Gregoire, and Vincent Laporte. 'Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time"'. In: *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, July 2018, pp. 328–343. DOI: 10.1109/csf.2018.00031 (cited on pages 16, 97).

[29] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 'CakeML: a verified implementation of ML'. In: *ACM SIGPLAN Notices* 49.1 (Jan. 2014), pp. 179–191. DOI: 10.1145/2578855.2535841 (cited on page 16).

[30] Orna Kupferman and Moshe Y. Vardi. 'Robust Satisfaction'. In: *CONCUR'99 Concurrency Theory*. Springer Berlin Heidelberg, 1999, pp. 383–398. DOI: 10.1007/3-540-48320-9_27 (cited on page 17).

[31] David Swasey, Deepak Garg, and Derek Dreyer. 'Robust and compositional verification of object capability patterns'. In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (Oct. 2017), pp. 1–26. DOI: 10.1145/3133913 (cited on pages 17, 18).

[32] John McLean. *Proving Noninterference and Functional Correctness Using Traces*. Jan. 1992 (cited on page 19).

[33] Max S. New, William J. Bowman, and Amal Ahmed. 'Fully abstract compilation via universal embedding'. In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. ICFP'16. ACM, Sept. 2016, pp. 103–116. DOI: 10.1145/2951913.2951941 (cited on pages 20, 25, 95).

[34] Brigitte Pientka. 'POPLMark reloaded: mechanizing logical relations proofs (invited talk)'. In: *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs - CPP 2018*. CPP 2018. ACM Press, 2018, pp. 1–1. DOI: 10.1145/3176245.3167102 (cited on page 26).

[35] Carmine Abate and Matteo Busi. 'The Fox and the Hound: Comparing Fully Abstract and Robust Compilation'. In: *CoRR* abs/2006.14969 (2020) (cited on page 32).

[36] Carmine Abate, Matteo Busi, and Stelios Tsampas. 'Fully Abstract and Robust Compilation: And How to Reconcile the Two, Abstractly'. In: *Programming Languages and Systems - 19th Asian Symposium, APLAS 2021, Chicago, IL, USA, October 17-18, 2021, Proceedings*. Ed. by Hakjoo Oh. Vol. 13008. Lecture Notes in Computer Science. Springer, 2021, pp. 83–101. DOI: 10.1007/978-3-030-89051-3_6 (cited on page 32).

[37] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 'Compositional CompCert'. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '15. ACM, Jan. 2015, pp. 275–287. DOI: 10.1145/2676726.2676985 (cited on page 33).

[38] Jérémie Koenig and Zhong Shao. 'CompCertO: compiling certified open C components'. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI '21. ACM, June 2021, pp. 1095–1109. DOI: 10.1145/3453483.3454097 (cited on pages 33, 43).

[39] Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. 'Lightweight verification of separate compilation'. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. ACM, Jan. 2016, pp. 178–190. DOI: 10.1145/2837614.2837642 (cited on page 37).

[40] Alan Jeffrey and Julian Rathke. 'Java JR: Fully Abstract Trace Semantics for a Core Java Language'. In: *Programming Languages and Systems*. Springer Berlin Heidelberg, 2005, pp. 423–438. DOI: 10.1007/978-3-540-31987-0_29 (cited on pages 39, 46, 95).

[41] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. *The CompCert Memory Model, Version 2*. Research Report RR-7987. INRIA, June 2012, p. 26 (cited on page 40).

[42] Yannis Juglaret, Catalin Hritcu, Arthur Azevedo De Amorim, Boris Eng, and Benjamin C. Pierce. 'Beyond Good and Evil: Formalizing the Security Guarantees of Compartmentalizing Compilation'. In: *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*. IEEE, June 2016, pp. 45–60. DOI: 10.1109/csf.2016.11 (cited on page 46).

[43] Marco Patrignani and Dave Clarke. 'Fully abstract trace semantics for protected module architectures'. In: *Computer Languages, Systems & Structures* 42 (July 2015), pp. 22–45. DOI: 10.1016/j.cl.2015.03.002 (cited on pages 46, 95).

[44] Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. 'Secure Compilation to Protected Module Architectures'. In: *ACM Trans. Program. Lang. Syst.* 37.2 (2015), 6:1–6:50. DOI: 10.1145/2699503 (cited on page 46).

[45] Akram El-Korashy, Stelios Tsampas, Marco Patrignani, Dominique Devriese, Deepak Garg, and Frank Piessens. 'CapablePtrs: Securely Compiling Partial Programs Using the Pointers-as-Capabilities Principle'. In: *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. Vol. 13. IEEE, June 2021, pp. 1–16. DOI: 10.1109/csf51468.2021.00036 (cited on pages 46, 58, 63, 71–73, 95).

[46] Marco Patrignani and Deepak Garg. 'Secure Compilation and Hyperproperty Preservation'. In: *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. IEEE, Aug. 2017, pp. 392–404. DOI: 10.1109/csf.2017.13 (cited on pages 46, 97).

[47] Sean Noble Anderson, Roberto Blanco, Leonidas Lampropoulos, Benjamin C. Pierce, and Andrew Tolmach. 'Formalizing Stack Safety as a Security Property'. In: *2023 IEEE 36th Computer Security Foundations Symposium (CSF)*. IEEE, July 2023, pp. 356–371. DOI: 10.1109/csf57540.2023.00037 (cited on page 55).

[48] James Laird. 'A Fully Abstract Trace Semantics for General References'. In: *Automata, Languages and Programming*. Springer Berlin Heidelberg, 2007, pp. 667–679. DOI: 10.1007/978-3-540-73420-8_58 (cited on page 71).

[49] R. S. Fabry. 'Capability-based addressing'. In: *Communications of the ACM* 17.7 (July 1974), pp. 403–412. DOI: 10.1145/361011.361070 (cited on page 72).

[50] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. 'Hardware support for fast capability-based addressing'. In: *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*. ASPLOS94. ACM, Nov. 1994, pp. 319–327. DOI: 10.1145/195473.195579 (cited on page 72).

[51] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 'The CHERI capability model: Revisiting RISC in an age of risk'. In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, June 2014, pp. 457–468. DOI: 10.1109/isca.2014.6853201 (cited on page 72).

[52] Marco Patrignani and Deepak Garg. 'Robustly Safe Compilation, an Efficient Form of Secure Compilation'. In: *ACM Transactions on Programming Languages and Systems* 43.1 (Feb. 2021), pp. 1–41. DOI: 10.1145/3436809 (cited on pages 73, 96).

[53] Martín Abadi. 'Protection in Programming-Language Translations'. In: *Secure Internet Programming*. Springer Berlin Heidelberg, 1999, pp. 19–34. DOI: 10.1007/3-540-48749-2_2 (cited on pages 78, 80, 95).

[54] Andy Keep. 'The Nanopass Framework as a Nanopass Compiler (ELS keynote)'. In: *Proceedings of the 13th European Lisp Symposium (ELS 2020), Zurich, Switzerland, April 27-28, 2020*. Ed. by Ioanna Matilde Dimitriou Henríquez. ELSAA, 2020 (cited on pages 78, 83).

[55] Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. 'A Nanopass framework for compiler education (educational pearl)'. In: *Journal of Functional Programming* 15.5 (June 2005), pp. 653–667. DOI: 10.1017/s0956796805005605 (cited on pages 78, 83).

[56] Andrew W. Keep and R. Kent Dybvig. 'A nanopass framework for commercial compiler development'. In: *ACM SIGPLAN Notices* 48.9 (Sept. 2013), pp. 343–350. DOI: 10.1145/2544174.2500618 (cited on pages 78, 83).

[57] Maulik A. Dave. 'Compiler verification: a bibliography'. In: *ACM SIGSOFT Software Engineering Notes* 28.6 (Nov. 2003), pp. 2–2. DOI: 10.1145/966221.966235 (cited on page 94).

[58] Xavier Leroy. 'Formal certification of a compiler back-end or: programming a compiler with a proof assistant'. In: *ACM SIGPLAN Notices* 41.1 (Jan. 2006), pp. 42–54. DOI: 10.1145/1111320.1111042 (cited on page 94).

[59] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 'CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency'. In: *Journal of the ACM* 60.3 (June 2013), pp. 1–50. DOI: 10.1145/2487241.2487248 (cited on page 94).

[60] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. 'CompCertS: A Memory-Aware Verified C Compiler Using a Pointer as Integer Semantics'. In: *Journal of Automated Reasoning* 63.2 (Nov. 2018), pp. 369–392. DOI: 10.1007/s10817-018-9496-y (cited on page 94).

[61] Yonghyun Kim, Minki Cho, Jaehyung Lee, Jinwoo Kim, Taeyoung Yoon, Youngju Song, and Chung-Kil Hur. 'Archmage and CompCertCast: End-to-End Verification Supporting Integer-Pointer Casting'. In: *Proceedings of the ACM on Programming Languages* 9.POPL (Jan. 2025), pp. 1326–1354. DOI: 10.1145/3704881 (cited on page 94).

[62] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 'Deep Specifications and Certified Abstraction Layers'. In: *ACM SIGPLAN Notices* 50.1 (Jan. 2015), pp. 595–608. DOI: 10.1145/2775051.2676975 (cited on page 94).

[63] Andrew W. Appel. 'Verified Software Toolchain'. In: *NASA Formal Methods*. Springer Berlin Heidelberg, 2012, pp. 2–2. DOI: 10.1007/978-3-642-28891-3_2 (cited on page 94).

[64] Daniel Patterson and Amal Ahmed. 'The next 700 compiler correctness theorems (functional pearl)'. In: *Proceedings of the ACM on Programming Languages* 3.ICFP (July 2019), pp. 1–29. DOI: 10.1145/3341689 (cited on page 94).

[65] Robin Milner. 'Processes: A Mathematical Model of Computing Agents'. In: *Logic Colloquium '73, Proceedings of the Logic Colloquium*. Elsevier, 1975, pp. 157–173. DOI: 10.1016/s0049-237x(08)71948-7 (cited on page 95).

[66] Robin Milner. 'Fully abstract models of typed λ-calculi'. In: *Theoretical Computer Science* 4.1 (Feb. 1977), pp. 1–22. DOI: 10.1016/0304-3975(77)90053-6 (cited on page 95).

[67] G.D. Plotkin. 'LCF considered as a programming language'. In: *Theoretical Computer Science* 5.3 (Dec. 1977), pp. 223–255. DOI: 10.1016/0304-3975(77)90044-5 (cited on page 95).

[68] Raymie Stata and Martín Abadi. 'A Type System for Java Bytecode Subroutines'. In: *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*. Ed. by David B. MacQueen and Luca Cardelli. ACM, 1998, pp. 149–160. DOI: 10.1145/268946.268959 (cited on page 95).

[69] M. Abadi, C. Fournet, and G. Gonthier. 'Secure implementation of channel abstractions'. In: *Proceedings. Thirteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.98CB36226)*. LICS-98. IEEE Comput. Soc, pp. 105–116. DOI: 10.1109/lics.1998.705647 (cited on page 95).

[70] Cedric Fournet, Nikhil Swamy, Juan Chen, Pierre-Evariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. 'Fully abstract compilation to JavaScript'. In: *ACM SIGPLAN Notices* 48.1 (Jan. 2013), pp. 371–384. DOI: 10.1145/2480359.2429114 (cited on page 95).

[71] Amal Ahmed and Matthias Blume. 'An equivalence-preserving CPS translation via multi-language semantics'. In: *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*. ICFP '11. ACM, Sept. 2011. DOI: 10.1145/2034773.2034830 (cited on page 95).

[72] Pieter Agten, Raoul Strackx, Bart Jacobs, and Frank Piessens. 'Secure Compilation to Modern Processors'. In: *2012 IEEE 25th Computer Security Foundations Symposium*. IEEE, June 2012, pp. 171–185. DOI: 10.1109/csf.2012.12 (cited on page 95).

[73] Amal Ahmed and Matthias Blume. 'Typed closure conversion preserves observational equivalence'. In: *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*. ICFP08. ACM, Sept. 2008. DOI: 10.1145/1411204.1411227 (cited on page 95).

[74] Dominique Devriese, Marco Patrignani, Frank Piessens, and Steven Keuchel. 'Modular, Fully-abstract Compilation by Approximate Back-translation'. In: *Logical Methods in Computer Science* Volume 13, Issue 4 (Oct. 2017). DOI: 10.23638/lmcs-13(4:2)2017 (cited on pages 95, 97).

[75] Matteo Busi, Job Noorman, Jo Van Bulck, Letterio Galletta, Pierpaolo Degano, Jan Tobias Mühlberg, and Frank Piessens. 'Securing Interruptible Enclaved Execution on Small Microprocessors'. In: *ACM Transactions on Programming Languages and Systems* 43.3 (Sept. 2021), pp. 1–77. DOI: 10.1145/3470534 (cited on pages 95, 96).

[76] Andrew Kennedy. 'Securing the .NET programming model'. In: *Theoretical Computer Science* 364.3 (Nov. 2006), pp. 311–317. DOI: 10.1016/j.tcs.2006.08.014 (cited on page 95).

[77] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 'StkTokens: enforcing well-bracketed control flow and stack encapsulation using linear capabilities'. In: *Proceedings of the ACM on Programming Languages* 3.POPL (Jan. 2019), pp. 1–28. DOI: 10.1145/3290332 (cited on page 96).

[78] Aïna Linn Georges, Alix Trieu, and Lars Birkedal. 'Le temps des cerises: efficient temporal stack safety on capability machines using directed capabilities'. In: *Proceedings of the ACM on Programming Languages* 6.OOPSLA1 (Apr. 2022), pp. 1–30. DOI: 10.1145/3527318 (cited on page 96).

[79] 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. ACM, Oct. 2017, pp. 1807–1823. DOI: 10.1145/3133956.3134078 (cited on page 97).

[80] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. 'Formal verification of a constant-time preserving C compiler'. In: *Proceedings of the ACM on Programming Languages* 4.POPL (Dec. 2019), pp. 1–30. DOI: 10.1145/3371075 (cited on page 97).

[81] Dominique Devriese, Eric Mark Martin, and Marco Patrignani. 'On the Semantic Expressiveness of Iso- and Equi-Recursive Types'. In: *Logical Methods in Computer Science* Volume 20, Issue 4 (Nov. 2024). DOI: 10.46298/lmcs-20(4:14)2024 (cited on page 97).

[82] Frédéric Besson, Sandrine Blazy, Alexandre Dang, Thomas Jensen, and Pierre Wilke. 'Compiling Sandboxes: Formally Verified Software Fault Isolation'. In: *Programming Languages and Systems*. Springer International Publishing, 2019, pp. 499–524. DOI: 10.1007/978-3-030-17184-1_18 (cited on page 97).

[83] 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: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2019, pp. 1–19. DOI: 10.1109/sp.2019.00002 (cited on page 99).