



The Quest for Formally Secure Compartmentalizing Compilation

Cătălin Hrițcu

► To cite this version:

Cătălin Hrițcu. The Quest for Formally Secure Compartmentalizing Compilation. Programming Languages [cs.PL]. ENS Paris; PSL Research University, 2019. tel-01995823

HAL Id: tel-01995823

<https://theses.hal.science/tel-01995823>

Submitted on 27 Jan 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License



École Normale Supérieure

Mémoire d'habilitation à diriger des recherches

Specialité Informatique

The Quest for Formally Secure Compartmentalizing Compilation

Cătălin Hrițcu (Inria Paris)

Correspondant HDR :

David Pointcheval (CNRS, DI/ENS, PSL Research University)

Rapporteurs :

Frank Piessens (KU Leuven)

Gilles Barthe (MPI for Security and Privacy and IMDEA Software Institute)

Thomas Jensen (Inria Rennes)

Examineurs :

David Pichardie (ENS Rennes)

Deepak Garg (MPI-SWS)

Karthikeyan Bhargavan (Inria Paris)

Tamara Rezk (Inria Sophia-Antipolis)

Xavier Leroy (Collège de France and Inria Paris)

January 21, 2019

Abstract

Severe low-level vulnerabilities abound in today’s computer systems, allowing cyber-attackers to remotely gain full control. This happens in big part because our programming languages, compilation chains, and architectures too often trade off security for efficiency. The semantics of mainstream low-level languages like C is inherently insecure, and even for safer languages, all guarantees are lost when interacting with low-level code, for instance when using low-level libraries. This habilitation presents my ongoing quest to build formally secure compartmentalizing compilation chains that defend against such attacks. In particular, we propose several formal definitions that characterize what it means for a compartmentalizing compilation chain to be secure, both in the case of safe and of unsafe source languages.

We start by investigating what it means for a compilation chain to provide secure interoperability between a safe source language and linked target-level code that is adversarial. In this model, a secure compilation chain ensures that even linked adversarial target-level code cannot break the security properties of a compiled program any more than some linked source-level code could. However, the precise class of security properties one chooses to preserve crucially impacts not only the supported security goals and the strength of the attacker model, but also the kind of protections the compilation chain has to introduce and the kind of proof techniques one can use to make sure that the protections are watertight. We are the first to thoroughly explore a large space of secure compilation criteria based on the preservation against adversarial contexts of various classes of trace properties such as safety, of hyperproperties such as noninterference, and of relational hyperproperties such as trace equivalence.

We then extend secure compartmentalizing compilation to unsafe languages like C and C++. We propose a new formal criterion for secure compilation schemes from such unsafe languages, expressing end-to-end security guarantees for software components that may become compromised after encountering undefined behavior—for example, by accessing an array out of bounds. Our criterion is the first to model dynamic compromise in a system of mutually distrustful components with clearly specified privileges. It articulates how each component should be protected from all the others—in particular, from components that have encountered undefined behavior and become compromised.

To illustrate this model, we construct a secure compilation chain for a small unsafe language with buffers, procedures, and components, targeting a simple abstract machine with built-in compartmentalization. We give a careful proof (mostly machine-checked in Coq) that this compiler satisfies our secure compilation criterion. We, moreover, show that the protection guarantees offered by the compartmentalized abstract machine can be achieved at the machine-code level using either software fault isolation or a tag-based reference monitor. Finally, we discuss the perspectives of scaling such formally secure compilation to realistic low-level programming languages like C.

Acknowledgements

My recent research, including this, is to a large extent the result of the hard work of a large group of outstanding students, post-docs, and engineers, whom I was very fortunate to supervise, including: Alejandro Aguirre, Arthur Azevedo de Amorim, Ana Nora Evans, Carmine Abate, Clément Pit-Claudel, Danel Ahman, Diane Gallois-Wong, Florian Groult, Guglielmo Fachini, Guido Martínez, Jérémy Thibault, Kenji Maillard, Li-yao Xia, Marco Stronati, Nick Giannarakis, Rob Blanco, Simon Forest, Théo Laurent, Tomer Libal, Victor Dumitrescu, Yannis Juglaret, and Zoe Paraskevopoulou. I am deeply indebted to each of them for their contagious enthusiasm, amazing energy, and all the astonishing things they have taught me.

Together with the other members of the Prosecco research team they have made my time at Inria Paris fantastic and have served as a second family. In addition, Karthik Bhargavan gave me insightful advice on many occasions and the freedom I needed to grow as an independent researcher. Bruno Blanchet always had an open door and was ready to help me on countless occasions. Graham Steel and Ben Smyth taught me the importance of having a beer. Anna Bednarik and Mathieu Mourey, the world's best research team assistants, provided excellent support on a daily basis.

My amazing external collaborators helped put my research in a broader perspective: Benjamin Pierce served as an inspiring role model from whom I have learned a great deal, and most importantly to strive for clarity, simplicity, thoroughness, honesty, and kindness. Nik Swamy got me thrilled about F^* and since then has been a great companion in the quest of turning interesting ideas into a tool others can use. Andrew Tolmach was always ready to give me advice and a helping hand. André DeHon taught me that efficiency and thorough experiments matter as much as thorough proofs. Deepak Garg is an endless fountain of intuition and good ideas that I was lucky to recently discover.

Finally, I am eternally grateful to my family for their endless love, care, and compassion. To my life partner Beate, who never ceases to amaze me with her kindness and selflessness and who never gave up on trying to make the world a better place and me a better person. To my mother Stela, who taught me the most important things in life and made any imaginable sacrifice to help me succeed. To my father Ioan, who was a great first moral compass and role model. To my sister Gabriela, with whom I shared both the happiness of childhood and the pain of growing up. Thank you everyone for being there!

Paris, September 7, 2018

Cătălin Hrițcu

Contents

Preface	6
Contribution 1: Tag-Based Security Monitoring	6
Contribution 2: Formally Secure Compilation	7
Contribution 3: F^* – A Language for Program Verification	7
Contribution 4: Dependable Property-Based Testing	8
1 Introduction	9
2 Secure Interoperability with Lower-Level Code	13
2.1 Overview	13
2.2 Robustly Preserving Classes of Trace Properties	18
2.2.1 Robust Trace Property Preservation (<i>RTP</i>)	18
2.2.2 Trace Model with Finite and Infinite Traces and Its Impact on Safety and Liveness	21
2.2.3 Robust Safety Property Preservation (<i>RSP</i>)	24
2.2.4 Robust Dense Property Preservation (<i>RDP</i>)	25
2.3 Robustly Preserving Classes of Hyperproperties	26
2.3.1 Robust Hyperproperty Preservation (<i>RHP</i>)	26
2.3.2 Robust Subset-Closed Hyperproperty Preservation (<i>RSCHP</i>)	27
2.3.3 Robust Hypersafety Preservation (<i>RHSP</i>)	28
2.3.4 Where is Robust Hyperliveness Preservation?	29
2.4 Robustly Preserving Classes of Relational Hyperproperties	30
2.4.1 Relational Hyperproperty Preservation (<i>RrHP</i>)	30
2.4.2 Relational Trace Property Preservation (<i>RrTP</i>)	32
2.4.3 Robust Relational Safety Preservation (<i>RrSP</i>)	33
2.4.4 Robust Non-Relational Preservation Doesn't Imply Robust Relational Preservation	34
2.5 Where is full abstraction?	35
2.6 Proving Secure Compilation	36
2.6.1 Source and Target Languages	37
2.6.2 The Compiler	38
2.6.3 Proving Robust Relational Hyperproperty Preservation	38
2.6.4 Proving Robust Finite-Relational Safety Preservation	40
2.7 Related Work	42
2.8 Conclusion and Future Work	44

3	Secure Compilation for Unsafe Languages	46
3.1	Overview	46
3.2	<i>RSCC</i> By Example	49
3.3	Formally Defining <i>RSCC</i>	56
3.3.1	RSC^{DC} : Dynamic Compromise	56
3.3.2	RSC_{MD}^{DC} : Mutually Distrustful Components	58
3.3.3	Formalizing <i>RSCC</i>	59
3.3.4	A Generic Proof Technique for <i>RSCC</i>	60
3.3.5	Class of safety properties preserved by RSC^{DC}	63
3.3.6	Comparison to Static Compromise	65
3.4	Secure Compilation Chain	65
3.4.1	Source Language	67
3.4.2	The Compartmentalized Machine	68
3.4.3	<i>RSCC</i> Proof in Coq	69
3.4.4	Software Fault Isolation Back End	72
3.4.5	Micro-policies Tagged Architecture	74
3.4.6	Tag-based Reference Monitor	76
3.5	Related Work	77
3.6	Conclusion	80
4	Research Plan for the Next 4 Years	81
	Bibliography	85
	Appendix	96

Preface

My research is primarily focused on developing rigorous formal techniques for solving security problems. My contributions span formal methods for computer and network security (memory safety, compartmentalization, dynamic monitoring, integrity, information flow, security protocols, privacy, anonymity), programming-languages techniques (type systems, verification, proof assistants, property-based testing, semantics, formal metatheory, certified tools), and the design and verification of security-critical systems (tag-based reference monitors, secure compilation chains, secure hardware). My research combines practice and theory: On the practical side, I design and build innovative software for solving real security problems, I experiment with this software, and I make it available for everybody to use. On the theoretical side, I make sure that each technique I propose is correct by coming up with appropriate attacker models and formal security definitions and then distilling the main ideas into a formalism that I prove correct, usually with the help of program verification systems and proof assistants.

This report presents research I have done since defending my PhD thesis in January 2012. This preface outlines my 4 main research contributions since 2012, while the rest of the report focuses for the most part on contributions 2 and 1, presenting them through the lens of my ongoing quest for achieving efficient formally secure compilation for realistic programming languages. The 5 research papers on which this thesis builds are reproduced in the appendix.

Contribution 1: Tag-Based Security Monitoring

During my postdoc at University of Pennsylvania I helped propel and steer the very ambitious DARPA CRASH/SAFE project, a large academia-industry collaboration (40+ people) that has undertaken the clean-slate co-design of a secure network host, including the design of novel hardware [Dhawan and DeHon 2013, Dhawan et al. 2012], operating/runtime system [Sullivan et al. 2013], programming language [Hrițcu et al. 2013a, Montagu et al. 2013], and the systematic testing [Hrițcu et al. 2013b, 2016] and verification of key components [Azevedo de Amorim et al. 2014, 2016]. I was actively involved in most of the design activities of CRASH/SAFE. I was a main designer and implementer of Breeze, a new high-level language with fine-grained dynamic information-flow control (IFC) and label-based access control (clearance). In particular, I was responsible for the novel security and exception handling mechanisms of Breeze [Hrițcu et al. 2013a]. I also took part in the design of the SAFE hardware and runtime system, which dynamically enforce type and memory safety, IFC and access control all the way down to the lowest level [Dhawan et al. 2012], and I played a leading role in designing, formalizing, testing,

and verifying the low-level IFC mechanisms of the SAFE system [Azevedo de Amorim et al. 2014, 2016, Hrițcu et al. 2013b, 2016].

Relatively late in the design of SAFE we realized that the tag-based monitoring mechanism, which was originally meant for enforcing IFC and access control, is a lot more general than we first thought, and can replace most of the other protection mechanisms of SAFE. This idea has led to a follow up Micro-Policies project between UPenn, Inria, and Portland State. In this project we have shown that a large number of critical safety and security *micro-policies* can be expressed as tag-based security monitors and efficiently enforced using hardware caching and sophisticated micro-architectural optimizations [Dhawan et al. 2015b]. Moreover, I have led the effort of devising a formal verification methodology for micro-policies and applying it to prove the security of our compartmentalization, control-flow integrity, and memory safety micro-policies [Azevedo de Amorim et al. 2015]. In parallel, our industrial partners at Draper Labs have continued working on a hardware platform for micro-policies based on the RISC-V ISA, and have created Dover Microsystems, a startup aimed at developing and commercializing this technology. This continued academic and industrial interest in micro-policies has recently led to a new DARPA-funded SSITH/HOPE project, in which I am also involved.

Contribution 2: Formally Secure Compilation

This new research project started around 2015 with my realization that micro-policies would be very well-suited for devising more secure compilation chains [Juglaret et al. 2015], and that this problem is very interesting in general, even irrespective of micro-policies. In particular, even defining what secure compilation means was a big open problem at that point. So we started by devising a variant of full abstraction that supports protecting mutually distrustful components written in an unsafe low-level language like C [Juglaret et al. 2016]. In the process we realized, however, that full abstraction is difficult to achieve and prove and it does not well match our intuitive attacker model; in particular, it does not interact well with undefined behavior and is limited to a static compromise model. So we investigated other security criteria based on preserving trace properties, hyperproperties, and relational hyperproperties against adversarial contexts [Abate et al. 2018b]. This solves the issues with full abstraction and in particular allows us to support a model of dynamic component compromise [Abate et al. 2018a]. This line of research forms the foundation of my ongoing ERC SECOMP project and is also the main focus of this report.

Contribution 3: F^* – A Language for Program Verification

Since early 2014, I am actively involved in the design and continuous evolution of F^* , a general-purpose functional programming language with effects targeted at program verification. The current F^* design is aimed at combining SMT-based automation with the power and expressiveness of proof assistants like Coq, which enables users to prove arbitrarily complex properties

manually. To achieve this we have introduced full dependent types and tracking of side-effects, while isolating a core language of pure total functions that can be used to write specifications and proof terms [Swamy et al. 2016]. A key insight in F^* was that verification conditions can be computed generically for any effect using so called *Dijkstra monads*, and that these Dijkstra monads can be derived “for free” from the monadic model of the effect [Ahman et al. 2017]. Moreover, carefully exposing the monadic representation of effects can be used to verify relational properties that characterize many useful notions of security, program refinement, and equivalence [Grimm et al. 2018]. This general treatment of side-effects allowed us to recently implement support for meta-programming and tactics in F^* simply as a user defined effect [Martínez et al. 2018]. Finally, we added to F^* convenient support for the verification of programs whose state evolves monotonically [Ahman et al. 2018]. The main idea is that a property witnessed in a prior state can be soundly recalled in the current state, provided (1) state evolves according to a given preorder, and (2) the property is preserved by this preorder. All these innovations of F^* are used to verify an efficient HTTPS stack in Project Everest [Bhargavan et al. 2017a,b, Protzenko et al. 2017, Zinzindohoué et al. 2017].

Contribution 4: Dependable Property-Based Testing

This project, which I initiated in 2013 and lead until 2015, investigates the integration of property-based testing [Claessen and Hughes 2000] and of formal verification in the Coq proof assistant in order to lower the costs of verification and increase the thoroughness of testing. For this we investigated realistic case studies [Hrițcu et al. 2013b, 2016], the integration of testing in Coq [Paraskevopoulou et al. 2015], the use of formal verification to improve the quality of testing [Paraskevopoulou et al. 2015], domain-specific languages for property-based generators [Lampropoulos et al. 2017], and a novel variant of mutation testing. This project has been successful at producing the QuickChick plugin for Coq, which is now maintained and further improved in the DeepSpec NSF expedition [Lampropoulos et al. 2018].

1 Introduction

Today’s computer systems are distressingly insecure. This affects the foundation upon which today’s information society is built and makes everyone potentially vulnerable. Visiting a website, opening an email, or serving a client request is often enough to cause a computer to be compromised by a cyber-attack that allows remote attackers to gain full control. This often results in the disclosure or destruction of information and the use of the machine in further cyber-attacks. Hundreds of thousands of compromised computers are hoarded into “botnets” that are used to send spam, mount distributed denial of service attacks, or mine cryptocurrency. Botnets are increasingly rented out by cyber-criminals as commodities and according to Symantec and Kaspersky Labs they are currently the biggest threat to the Internet. Given their cyber-attack power, previously unknown (“0-day”) exploitable low-level vulnerabilities in widely-used software are often sold to intelligence agencies or botnet “controllers” for tens to hundreds of thousands of dollars.

The causes for this dissatisfying state of affairs are complex, but at this point mostly historical: our programming languages, compilers, and architectures were designed in an era of scarce hardware resources and far too often trade off security for efficiency. Today’s mainstream low-level languages, C and C++, give up even on the most basic safety checks for the sake of efficiency, which leaves programmers bearing all the burden for security: the smallest mistake in widely-deployed C and C++ code can cause security vulnerabilities with disastrous consequences [Durumeric et al. 2014]. Four of the top 25 most dangerous software error types (<https://cwe.mitre.org/top25/>) would be prevented or effectively mitigated by ensuring memory safety alone, including #3 in the top: “Classic Buffer Overflow.” The C and C++ languages do not guarantee memory safety and their compilation chains do not enforce it because currently deployed hardware provides no good support for it and software checks would incur 70-80% overhead on average [Nagarakatte et al. 2010, 2015]. Instead, much weaker low-overhead mitigation techniques are deployed and routinely circumvented by practical attacks [Conti et al. 2015, Evans et al. 2015, Szekeres et al. 2013]. Unfortunately, just ensuring memory safety would in fact not be enough to make C and C++ safe, as the standards and compilers for these languages call out a much larger number of undefined behaviors [Hathhorn et al. 2015, Krebbers 2015], for which compilers produce code that behaves arbitrarily, often leading to security vulnerabilities, including for instance invalid unchecked type casts [Duck and Yap 2018, Haller et al. 2016], data races, and sometimes even integer overflows.

Safer languages such as Java, C#, ML, Haskell, or Rust provide memory safety and type safety by default as well as many useful abstractions for writing more secure code (e.g., modules, interfaces, parametric polymorphism, etc). Unfortunately, these languages are still not immune

to low-level attacks. All the safety guarantees of these source languages are lost when interacting with low-level code, for instance when using low-level libraries. This interaction is useful but dangerous because the low-level code can be malicious or compromised (e.g., by a buffer overflow). Currently, not only is the low-level code trusted to be safe, but also to preserve all the complex abstractions and internal invariants of the high-level language semantics, compiler, and runtime system. So even if some critical code is secure with respect to the semantics of a high-level language, any low-level code with which it interacts can break its security.

Verification languages such as Coq and F^{*} [Swamy et al. 2016] provide additional abstractions, such as dependent types, logical pre- and postconditions, and tracking the precise effects of computations, distinguishing between pure and stateful computations or computations that can raise exceptions. Such abstractions are crucial for making the verification effort more tractable in practice, but they also make the final verification result only valid in these very abstract languages. In order for a Coq or F^{*} program to be executed it is first compiled all the way down to machine code. Even if the compilation is correct [Kumar et al. 2014, Leroy 2009a], this is usually not enough to ensure the security of the verified code, since usually not all the code can be written and verified in the abstract verification language.

For a concrete example, consider the miTLS^{*} implementation of the TLS standard, the most widely-used security protocol framework on the Internet. miTLS^{*} is being written and formally verified in Low^{*}, a safe subset of C embedded in F^{*} [Bhargavan et al. 2017b, Protzenko et al. 2017, Zinzindohoué et al. 2017]. miTLS^{*} includes tens of thousands lines of Low^{*} code, and even when all this code will be formally verified, it will just be a tiny library linked from large unverified applications such as web browsers, web servers, and operating systems, which have millions of lines of C, C++, and ASM code. Not only are these applications not verified and can thus break the verified security properties of the Low^{*} code, but these applications are not even memory safe, and any error can allow remote attackers to take complete control, disclose the memory of the process stealing the TLS private keys, etc. A correct compilation chain is not enough in this case, since (1) a correct compilation chain [Leroy 2009a] for an insecure language like C still produces insecure code and leaves the burden of avoiding undefined behaviors to the programmer, and (2) a correct compilation chain does not protect the interaction between high-level and low-level code and does not enforce the abstractions of each language against faulty or malicious code written in the lower-level languages. In order for miTLS^{*} to be secure in practice we don't need only correct compilation, but also *secure compilation*.

In the ERC SECOMP project we study the use of *compartmentalization* to practically defend against low-level attacks and achieve secure compilation. Widely deployed compartmentalization technologies include process-level privilege separation [Bittau et al. 2008, Gudka et al. 2015, Kilpatrick 2003] (used in OpenSSH [Provos et al. 2003] and for sandboxing plugins and tabs in web browsers [Reis and Gribble 2009]), software fault isolation [Tan 2017, Wahbe et al. 1993] (e.g., Google Native Client [Yee et al. 2010]), WebAssembly modules [Haas et al. 2017] in modern web browsers, and hardware enclaves (e.g., Intel SGX [Intel]); many more are on the drawing boards [Azevedo de Amorim et al. 2015, Chisnall et al. 2015, Skorstengaard et al. 2018a, Watson et al. 2015b]. These compartmentalization mechanisms offer an attractive base for building more secure compilation chains that prevent or at least mitigate low-level attacks [Gollamudi

and Fournet 2018, Gudka et al. 2015, Juglaret et al. 2015, Patrignani et al. 2016, Tsampas et al. 2017, van Ginkel et al. 2016, Van Strydonck et al. 2018].

However, what does it mean for a compartmentalizing compilation chain to be secure? This thesis provides several formal security definitions that can answer this question for both safe and unsafe source languages.

Secure interoperability with lower-level code In chapter 2 we investigate what it means for a compilation chain to provide secure interoperability between a *safe* source language and linked target-level code that is adversarial. In this model, a secure compilation chain protects source-level abstractions all the way down, ensuring that even an adversarial target-level context cannot break the security properties of a compiled program any more than some source-level context could. However, the precise class of security properties one chooses to preserve crucially impacts not only the supported security goals and the strength of the attacker model, but also the kind of protections the compilation chain has to introduce and the kind of proof techniques one can use to make sure that the protections are watertight. Since efficiently achieving and proving secure compilation at scale are challenging open problems, designers of secure compilation chains have to strike a pragmatic balance between security and efficiency that matches their application domain.

To inform this difficult design decision, we thoroughly explore a large space of formal secure compilation criteria based on the preservation of properties that are *robustly* satisfied against arbitrary adversarial contexts. We study robustly preserving various classes of trace properties such as safety, of hyperproperties such as noninterference, and of relational hyperproperties such as trace equivalence. For each of the studied classes we propose an equivalent “property-free” characterization of secure compilation that is generally better tailored for proofs. We, moreover, order the secure compilation criteria by their relative strength and prove several separation results.

Finally, we show that even the strongest of our secure compilation criteria, the robust preservation of all relational hyperproperties, is achievable for a simple translation from a statically typed to a dynamically typed language. We prove this using a universal embedding, a context back-translation technique previously developed for fully abstract compilation. We also illustrate that for proving the robust preservation of most relational safety properties including safety, noninterference, and sometimes trace equivalence, a less powerful but more generic technique can back-translate a finite set of finite execution prefixes into a source context.

The presentation in chapter 2 closely follows a research paper draft [Abate et al. 2018b] that I have recently co-authored and which is also included in the appendix.

Secure compartmentalization for unsafe languages In chapter 3 we extend secure compartmentalizing compilation to unsafe languages like C and C++. We propose a new formal criterion for evaluating secure compilation schemes for such unsafe languages, expressing end-to-end security guarantees for software components that may become compromised after encountering *undefined behavior*—for example, by accessing an array out of bounds.

Our criterion is the first to model *dynamic* compromise in a system of mutually distrustful components with clearly specified privileges. It articulates how each component should be protected from all the others—in particular, from components that have encountered undefined behavior and become compromised. Each component receives secure compilation guarantees—in particular, its internal invariants are protected from compromised components—up to the point when this component itself becomes compromised, after which we assume an attacker can take complete control and use this component’s privileges to attack other components. More precisely, a secure compilation chain must ensure that a dynamically compromised component cannot break the safety properties of the system at the target level any more than an arbitrary attacker-controlled component (with the same interface and privileges, but without undefined behaviors) already could at the source level.

To illustrate the model, we construct a secure compilation chain for a small unsafe language with buffers, procedures, and components, targeting a simple abstract machine with built-in compartmentalization. We give a careful proof (mostly machine-checked in Coq) that this compiler satisfies our secure compilation criterion. Finally, we show that the protection guarantees offered by the compartmentalized abstract machine can be achieved at the machine-code level using either software fault isolation or a tag-based reference monitor.

The presentation in chapter 3 closely follows a recent research paper [Abate et al. 2018a]. The main exceptions are: §3.3.6, which illustrates the more restrictive static compromise model of an earlier paper that served as a stepping stone for the current work [Juglaret et al. 2016]; and §3.4.5, which introduces micro-policies [Azevedo de Amorim et al. 2015], the mechanism for tag-based reference monitors that we use for one of the back ends of our prototype compilation chain and that we hope will allow us to achieve efficient formally secure compilation at scale. Micro-policies generalize a tagging mechanism we originally devised to efficiently enforce information flow control [Azevedo de Amorim et al. 2016], and which also served as a stepping stone for the work presented here. I have substantially contributed to these research papers and they are included in the appendix.

Longer-term perspectives While chapters 2 and 3 use simple secure compilation chains for illustrating the main ideas, scaling this up to realistic languages and compilation chains is still an open challenge. The final goal of the ERC SECOMP project is to build the first formally secure compilation chains for realistic programming languages. In particular, we are planning a secure compilation chain starting from programs written in a combination of C and Low* [Protzenko et al. 2017] and targeting a RISC-V architecture [Asanović and Patterson 2014] extended with micro-policies [Azevedo de Amorim et al. 2015], building up on the simple prototype from chapter 3. In order to ensure high confidence in the security of our compilation chains, we plan to thoroughly test them using property-based testing and then formally verify their security using Coq. For measuring and optimizing efficiency we plan to use standard benchmark suites [Henning 2006] and realistic source programs, with miTLS* as the main end-to-end case study. Chapter 4 further explains this research plan.

2 Secure Interoperability with Lower-Level Code: Journey Beyond Full Abstraction

2.1 Overview

Good programming languages provide helpful abstractions for writing secure code. For example, the HACL* [Zinzindohoué et al. 2017] and miTLS* [Bhargavan et al. 2017b] verified cryptographic libraries are written in Low* [Protzenko et al. 2017], a language that provides many different kinds of abstractions: from low-level abstractions associated with safe C programs (such as structured control flow, procedures, and a block-based memory model inspired by CompCert [Leroy and Blazy 2008]), to higher-level abstractions associated with typed functional languages like ML (such as modules, interfaces, and parametric polymorphism), to features associated with verification systems like Coq and Dafny (such as effects, dependent types, and logical pre- and post-conditions), and, finally, to patterns specific to cryptographic code (such as using abstract types and restricted interfaces to rule out certain side-channel attacks). Such abstractions are crucial in making the effort required to reason about the correctness and security properties of realistic code tractable.

However, such abstractions are not enforced all the way down by today’s compilation chains. In particular, the security properties a program has in the source language are generally not preserved when compiling the program and linking it with adversarial low-level code. HACL* and miTLS* are *libraries* that get linked into real applications such as web browsers [Beurdouche et al. 2018, Erbsen et al. 2019], which include millions of lines of legacy C/C++ code. Even if we formally proved, say, that because of the way the miTLS* library is structured and verified, no Low* application embedding miTLS* can cause it to leak a private decryption key, this guarantee is completely lost when compiling miTLS* [Leroy 2009a, Protzenko et al. 2017] and linking it into a C/C++ application that can get compromised via a buffer overflow and simply read off the private key from memory [Durumeric et al. 2014, Szekeres et al. 2013]. More generally, a compromised or malicious application that links in the miTLS* library can easily read and write the data and code of miTLS*, jump to arbitrary instructions, or smash the stack, blatantly violating any source-level abstraction and breaking any guarantee obtained by source-level reasoning.

An idea that has been gaining increasing traction recently is that it should be possible to build *secure compilation chains* that protect source-level abstractions all the way down, ensuring that an adversarial target-level context cannot break the security properties of a compiled program any more than some source-level context could [Abadi 1999, Abadi and Plotkin 2012, Abadi

et al. 2002, Abate et al. 2018a, Ahmed 2015, Ahmed and Blume 2008, 2011, Devriese et al. 2016a, 2017, Fournet et al. 2013, Jagadeesan et al. 2011, Juglaret et al. 2016, Larmuseau et al. 2015, New et al. 2016, Patrignani and Garg 2018, Patrignani et al. 2015, 2016, 2019]. Such a compilation chain enables reasoning about the security of compiled code with respect to the semantics of the source programming language, without having to worry about “low-level” attacks from the target-level context. In order to achieve this, the various parts of the secure compilation chain—including for instance the compiler, linker, loader, runtime, system, and hardware—have to work together to provide enough protection to the compiled program, so that any security property proved against all source contexts also holds against all target contexts.

However, the precise class of security properties one chooses to preserve is crucial. Full abstraction [Abadi 1999], currently the most well-known secure compilation criterion [Abadi and Plotkin 2012, Abadi et al. 2002, Ahmed 2015, Ahmed and Blume 2008, 2011, Devriese et al. 2016a, 2017, Fournet et al. 2013, Jagadeesan et al. 2011, Juglaret et al. 2016, Larmuseau et al. 2015, New et al. 2016, Patrignani et al. 2015, 2016, 2019], would, for instance, not be very well-suited to preserving the confidentiality of miTLS*’s private key. First, while a fully abstract compilation chain preserves (and reflects) observational equivalence, the confidentiality of miTLS*’s private key is a noninterference property that is not directly captured by observational equivalence. Second, even if one was able to encode noninterference as an observational equivalence [Abadi 1999, Patrignani et al. 2019], the kind of protections one has to put in place for preserving observational equivalence will likely be overkill if all one wants to preserve is noninterference against adversarial contexts (or, to use terminology from the next paragraph, to preserve robust noninterference). It is significantly harder to hide the difference between two programs that are observationally equivalent but otherwise arbitrary, compared to hiding some clearly identified secret data of a single program (e.g., the miTLS* private key), so a secure compilation chain for robust noninterference can likely be much more efficient than one for observational equivalence. Moreover, achieving full abstraction is hopeless in the presence of side-channels, while preserving noninterference is still possible, at least in specific scenarios [Barthe et al. 2018]. In general, stronger secure compilation criteria are also harder (or even impossible) to achieve efficiently and designers of secure compilation chains are faced with a difficult decision, having to strike a pragmatic balance between security and efficiency that matches their application domain. Finally, even when efficiency is not a concern (e.g., when security is enforced by *static* restrictions on target-level contexts [Abadi 1999, Ahmed 2015, Ahmed and Blume 2008, 2011, New et al. 2016]), stronger secure compilation criteria are still harder to prove. In our example, proving preservation of noninterference is likely much easier than proving full abstraction, a notoriously challenging task even for very simple languages, with apparently simple conjectures surviving for decades before being finally settled, sometimes negatively [Devriese et al. 2018].

Convinced that there is no “one-size-fits-all” solution to secure compilation, we set out to explore a large space of security properties that can be preserved against adversarial target-level contexts. We explore preserving classes of *trace properties* such as safety and liveness [Lamport and Schneider 1984], of *hyperproperties* such as noninterference [Clarkson and Schneider 2010], and of *relational hyperproperties* such as trace equivalence, against adversarial target-level contexts. All these property notions are phrased in terms of (finite and infinite) execution

traces that are built over events such as inputs from and outputs to an external environment [Kumar et al. 2014, Leroy 2009a]. For instance, *trace properties* are defined simply as sets of allowed traces [Lamport and Schneider 1984]. One says that a whole program W *satisfies* a trace property π when the set of traces produced by W is included in the set π or, formally, $\{t \mid W \rightsquigarrow t\} \subseteq \pi$, where $W \rightsquigarrow t$ indicates that program W can emit trace t . More interestingly, we say that a partial program P *robustly satisfies* [Gordon and Jeffrey 2004, Kupferman and Vardi 1999, Swasey et al. 2017] a trace property π when P linked with any (adversarial) context satisfies π . More formally, P robustly satisfies π if for all contexts C we have that $C[P]$ satisfies π , where $C[P]$ is the operation of linking the partial program P with the context C to produce a whole program that can be executed. Armed with this, we define our first secure compilation criterion as the preservation of robust satisfaction of trace properties, which we call *Robust Trace Property Preservation (RTP)*. So if a partial source program P robustly satisfies a trace property $\pi \in 2^{\text{Trace}}$ (wrt. all source contexts) then its compilation $P\downarrow$ must also robustly satisfy π (wrt. all target-level contexts). If we unfold all intermediate definitions, a compilation chain satisfies *RTP* if

$$\text{RTP} : \quad \forall \pi \in 2^{\text{Trace}}. \forall P. (\forall C_S t. C_S[P] \rightsquigarrow t \Rightarrow t \in \pi) \Rightarrow (\forall C_T t. C_T[P\downarrow] \rightsquigarrow t \Rightarrow t \in \pi)$$

In such criteria we use a **blue, sans-serif** font for **source** elements, an **orange, bold** font for **target** elements and a *black, italic* font for elements common to both languages. Throughout this thesis we assume that traces are exactly the same in both the source and target language, as is also the case in CompCert [Leroy 2009a] (we discuss lifting this limitation in §2.8).

In this chapter we study various such secure compilation criteria, all based on the preservation of robust satisfaction, as outlined by the nodes of the diagram in Figure 2.1. We first look at robustly preserving classes of *trace properties* (§2.2) such as *safety* and *dense properties*—i.e., the criteria in the yellow area in Figure 2.1. *Safety properties* intuitively require that a violation never happens in a finite prefix of a trace, since this prefix is observable for instance to a reference monitor. Less standardly, in our trace model with both finite and infinite traces, the role of *liveness* is taken by what we call *dense properties*, which are simply trace properties that can only be falsified by non-terminating executions. We then generalize robust preservation from properties of individual program traces to *hyperproperties* (§2.3), which are properties over multiple traces of a program [Clarkson and Schneider 2010] (the criteria in the red area in Figure 2.1). The canonical example of a hyperproperty is noninterference, which generally requires considering two traces of a program that differ on secret inputs [Askarov et al. 2008, Goguen and Meseguer 1982, McLean 1992, Sabelfeld and Myers 2003, Sabelfeld and Sands 2001, Zdancewic and Myers 2003]. We then generalize this further to what we call *relational hyperproperties* (§2.4), which relate multiple runs of *different* programs (the criteria in the blue area in Figure 2.1). An example of relational hyperproperty is trace equivalence, which requires that two programs produce the same set of traces.

For each of the studied criteria we propose an equivalent “property-free” characterization that is generally better tailored for proofs. For instance, by simple logical reasoning we prove that *RTP* can equivalently be stated as follows:

$$\text{RTC} : \quad \forall P. \forall C_T. \forall t. C_T[P\downarrow] \rightsquigarrow t \Rightarrow \exists C_S. C_S[P] \rightsquigarrow t$$

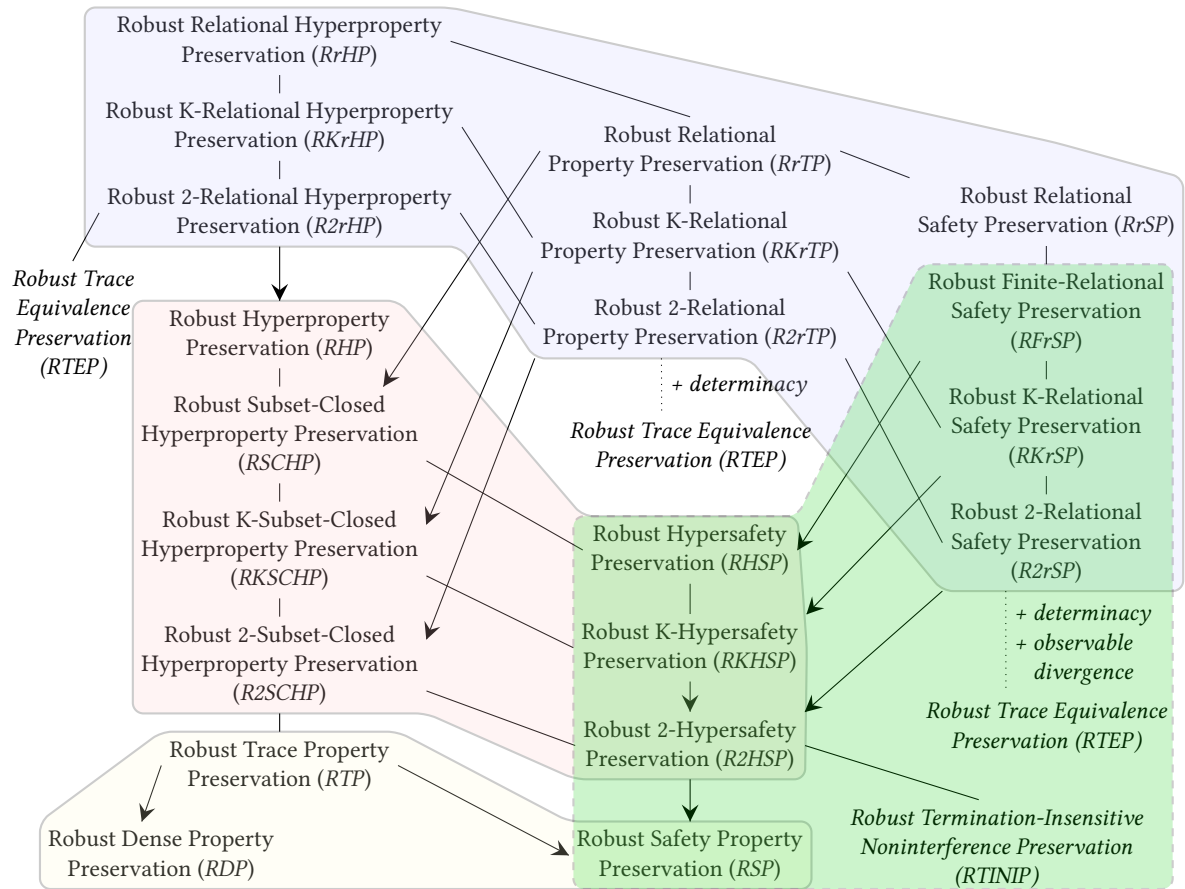



Figure 2.1: Partial order with the secure compilation criteria studied in this chapter. Criteria higher in the diagram imply the lower ones to which they are connected by edges. Criteria based on trace properties are grouped in a yellow area, those based on hyperproperties are in a red area, and those based on relational hyperproperties are in a blue area. Criteria in the green area can be proved by back-translating a finite set of finite execution prefixes into a source context. Criteria with an *italics name* preserve a single property that belongs to the class they are connected to; dashed edges require additional assumptions (stated on the edge) to hold. Finally, edges with a thick arrow denote a *strict* implication.

This requires that, given a compiled program $P\downarrow$ and a target context C_T which together produce some bad trace t , we can generate a source context C_S that produces trace t when linked with P . When proving that a compilation chain satisfies *RTC* we can pick a different context C_S for each t and, in fact, construct C_S from the trace t itself or from the execution $C_T[P\downarrow] \rightsquigarrow t$. In contrast, for stronger criteria a single context C_S will have to work for several traces. In general, the shape of the property-free characterization gives us a good clue for what kind of *back-translation* techniques are possible for producing C_S , when proving that a concrete compilation chain is secure.

We order the secure compilation criteria we study by their relative strength as illustrated by the partial order in Figure 2.1. In this Hasse diagram edges represent logical implication from higher criteria to lower ones, so the higher is a property, the harder it is to achieve and prove. While most of the implications in the diagram are unsurprising and follow directly from the inclusion between the property classes [Clarkson and Schneider 2010], we discover that preserving hyperliveness is in fact equivalent to preserving all hyperproperties (§2.3.4). To show the absence of more such collapses, we also prove various separation results, for instance that *Robust Safety Property Preservation* (*RSP*) and *Robust Dense Property Preservation* (*RDP*) when taken separately are *strictly* weaker than *RTP*. While these results are natural, the separation result for dense properties crucially relies on a trace model that explicitly distinguishes finite and infinite traces (§2.2.2), since with just infinite traces [Alpern and Schneider 1985, Clarkson and Schneider 2010] things do collapse even for liveness.

We moreover show (§2.4.1) that in general *Robust Trace Equivalence Preservation* (*RTEP*) follows only from *Robust 2-relational Hyperproperty Preservation*, which is one of our strongest criteria. However, in the absence of internal nondeterminism (i.e., if the source and target languages are determinate [Engelfriet 1985, Leroy 2009a]) and under some mild extra assumptions (such as input totality [Focardi and Gorrieri 1995, Zakinthinos and Lee 1997]) *RTEP* follows from the weaker *Robust 2-relational Trace Property Preservation* (*R2rTP*), and under much stronger assumptions (divergence being finitely observable) from *Robust 2-relational Safety Preservation* (*R2rSP*). In determinate settings, where observational equivalence is equivalent to trace equivalence [Cheval et al. 2013, Engelfriet 1985], these results provide a connection to *Observational Equivalence Preservation*, i.e., the direction of fully abstract compilation that is interesting for secure compilation (§2.5).

Finally, we show that even the strongest of our secure compilation criteria, *Robust Relational Hyperproperty Preservation* (*RrHP*), is achievable for a simple translation from a statically typed to a dynamically typed language with first-order functions and input-output (§2.6). We prove this using a “universal embedding,” which is a context back-translation technique previously developed for fully abstract compilation [New et al. 2016]. For the same simple translation we also illustrate that for proving *Robust Finite-relational Safety Preservation* (*RFrSP*) a less powerful but more generic technique can back-translate a finite set of finite execution prefixes into a source context. This technique is applicable to all the criteria contained in area below *RFrSP* (indicated in green in Figure 2.1), which includes robust preservation of safety, of noninterference, and sometimes even trace equivalence.

We close with discussions of related (§2.7) and future (§2.8) work. Appendices included as supplementary material present omitted technical details. Many of the formal results of §2.2, §2.3, and §2.4 were mechanized in Coq and are marked with . This Coq development is available as another supplementary material. All these materials are available at <https://github.com/secure-compilation/exploring-robust-property-preservation>

2.2 Robustly Preserving Classes of Trace Properties

We start by looking at robustly preserving classes of *trace properties*. The introduction already defined *RTP*, the robust preservation of *all* trace properties, so we first explore this criterion in more detail (§2.2.1). We then step back and define a model for program execution traces that can be non-terminating or terminating, finite or infinite (§2.2.2). Using this model we define *safety properties* in the standard way as the trace properties that can be falsified by a finite trace prefix (e.g., a program never performs a certain dangerous system call). Perhaps more surprisingly, in our model the role usually played by liveness is taken by what we call *dense properties*, which we define simply as the trace properties that can only be falsified by non-terminating traces (e.g., a reactive program that runs forever eventually answers every network request it receives). To validate that dense properties indeed play the same role liveness plays in previous models [Alpern and Schneider 1985, Lamport 2002, Lamport and Schneider 1984, Manna and Pnueli 2012, Schneider 1997], we prove various properties, including that every trace property is the intersection of a safety property and a dense property (this is our variant of a standard decomposition result [Alpern and Schneider 1985]). We then use these definitions to study the robust preservation of safety properties (*RSP*; §2.2.3) and dense properties (*RDP*; §2.2.4). These secure compilation criteria are highlighted in yellow in Figure 2.1.

2.2.1 Robust Trace Property Preservation (*RTP*)

Like all secure compilation criteria we study in this chapter, the *RTP* criterion presented in the introduction and further explained below is a generic property of a *compilation chain*, which includes a source and a target language, each with a notion of partial programs (P) and contexts (C) that can be linked together to produce whole programs ($C[P]$), and each with a trace-producing semantics for whole programs ($C[P] \rightsquigarrow t$). The sets of partial programs and of contexts of the source and target languages are arbitrary parameters of our secure compilation criteria; our generic criteria make no assumptions about their structure, or whether static typing is involved or not, or whether the program or the context gets control initially once linked and executed (e.g., the context could be an application that embeds a library program or the context could be a library that is embedded into an application program). Similarly, the traces of the source and target semantics are arbitrary for *RTP*, while starting with §2.2.2 we will consider finite or infinite lists of events drawn from an arbitrary set. The intuition is that the traces capture the interaction between a whole program and its external environment, including for instance user input, output to a terminal, network communication, system calls, etc. [Kumar et al. 2014, Leroy 2009a]. As opposed to a context, which is just a piece of a program,

the environment is not (and often *cannot* be) precisely modeled by the programming language, beyond the (often nondeterministic) events that we store in the trace (and which often record the data that the program inputs and outputs). Finally, a compilation chain includes a compiler. The compilation of a partial source program P is a partial target program written $P\downarrow$.

The responsibility of enforcing secure compilation does not have to rest just with the compiler, but may be freely shared by various parts of the compilation chain. In particular, to help enforce security, the target-level linker could disallow linking with a suspicious context (e.g., one that is not well-typed [Abadi 1999, Ahmed 2015, Ahmed and Blume 2008, 2011, New et al. 2016]) or could always allow linking but introduce protection barriers between the program and the context (e.g., by instrumenting the program [Devriese et al. 2016a, New et al. 2016] or the context [Abate et al. 2018a, Tan 2017, Wahbe et al. 1993] to introduce dynamic checks). Similarly, the semantics of the target language can include various protection mechanisms (e.g., processes with different virtual address spaces [Bittau et al. 2008, Gudka et al. 2015, Kilpatrick 2003, Provos et al. 2003, Reis and Gribble 2009], protected modules [Patrignani et al. 2015], capabilities [El-Korashy et al. 2018], tags [Abate et al. 2018a, Azevedo de Amorim et al. 2015]). Finally, the compiler might have to refrain from too aggressive optimizations that would break security [Barthe et al. 2018, D’Silva et al. 2015, Simon et al. 2018]. In this chapter we propose general secure compilation criteria that are agnostic to the concrete enforcement mechanism used by the compilation chain to protect the compiled program from the adversarial target context.

In §2.1, we defined RTP as the preservation of robust satisfaction of trace properties:

$$RTP : \quad \forall \pi \in 2^{Trace}. \forall P. (\forall C_S t. C_S[P] \rightsquigarrow t \Rightarrow t \in \pi) \Rightarrow (\forall C_T t. C_T[P\downarrow] \rightsquigarrow t \Rightarrow t \in \pi)$$

Trace properties are simply sets of allowed traces [Lamport and Schneider 1984] and a whole program *satisfies* a trace property if all the traces it can produce are in the set of allowed traces representing the trace property. A partial program *robustly satisfies* a property if the traces it can produce when linked with any context are all included in the set representing the property.

The definition of RTP above directly captures which security properties of the source are preserved by the compilation chain. However, in order to prove that a compilation chain satisfies RTP we gave an equivalent “property-free” characterization in the introduction:

$$RTC : \quad \forall P. \forall C_T. \forall t. C_T[P\downarrow] \rightsquigarrow t \Rightarrow \exists C_S. C_S[P] \rightsquigarrow t$$

The equivalence proof between RTP and RTC is simple, but still illustrative:

Theorem 1. $RTP \iff RTC$ 

Proof. (\Rightarrow) Let P be arbitrary. We need to show that $\forall C_T. \forall t. C_T[P\downarrow] \rightsquigarrow t \Rightarrow \exists C_S. C_S[P] \rightsquigarrow t$. We can directly conclude this by applying RTP to P and the property $\pi = \{t \mid \exists C_S. C_S[P] \rightsquigarrow t\}$; for this application to be possible we need to show that $\forall C_S t. C_S[P] \rightsquigarrow t \Rightarrow \exists C'_S. C'_S[P] \rightsquigarrow t$, which is trivial if taking $C'_S = C_S$.

(\Leftarrow) Given a compilation chain that satisfies *RTC* and some P and π so that $\forall C_S t. C_S [P] \rightsquigarrow t \Rightarrow t \in \pi$ (H) we have to show that $\forall C_T t. C_T [P \downarrow] \rightsquigarrow t \Rightarrow t \in \pi$. Let C_T and t so that $C_T [P \downarrow] \rightsquigarrow t$, we still have to show that $t \in \pi$. We can apply *RTC* to obtain $\exists C_S. C_S [P] \rightsquigarrow t$, which we can use to instantiate H to conclude that $t \in \pi$.

□

The *RTC* characterization is similar to “backward simulation”, which is the standard criterion for compiler correctness [Leroy 2009a]:

$$TP : \quad \forall W. \forall t. W \downarrow \rightsquigarrow t \Rightarrow W \rightsquigarrow t$$

Maybe less known is that this property-free characterization of correct compilation also has an equivalent property-full characterization as the preservation of all trace properties:

$$TP : \quad \forall \pi \in 2^{Trace}. \forall W. (\forall t. W \rightsquigarrow t \Rightarrow t \in \pi) \Rightarrow (\forall t. W \downarrow \rightsquigarrow t \Rightarrow t \in \pi)$$

The major difference compared to *RTP* is that *TP* only preserves the trace properties of whole programs and does not allow any form of linking. In contrast, *RTP* allows linking a compiled partial program with arbitrary target contexts and protects the program so that all *robust* trace properties are preserved. In general, *RTP* and *TP* are incomparable. However, *RTP* strictly implies *TP* when whole programs (W) are a subset of partial programs (P) and, additionally, the semantics of *whole* programs is independent of any linked context (i.e., $\forall W t C. W \rightsquigarrow t \iff C[W] \rightsquigarrow t$, which happens, intuitively, when the whole program starts execution and, being whole, never calls into the context).

More compositional criteria for compiler correctness have been proposed in the literature [Kang et al. 2016, Neis et al. 2015, Perconti and Ahmed 2014, Stewart et al. 2015]. At a minimum such criteria allow linking with contexts that are the compilation of source contexts [Kang et al. 2016], which in our setting can be formalized as follows:

$$SCC : \quad \forall P. \forall C_S. \forall t. C_S \downarrow [P \downarrow] \rightsquigarrow t \Rightarrow C_S [P] \rightsquigarrow t$$

More permissive criteria allow linking with any target context that behaves like some source context [Neis et al. 2015], which in our setting can be written as:

$$CCC : \quad \forall P. \forall C_T. \forall C_S. \forall t. C_T \approx C_S \wedge C_T [P \downarrow] \rightsquigarrow t \Rightarrow C_S [P] \rightsquigarrow t$$

RTP is incomparable to *SCC* and *CCC*. On one hand, *RTP* allows linking with *arbitrary* target-level contexts, which is not allowed by *SCC* and *CCC* and requires inserting strong protection barriers. On the other hand, in *RTP* all source-level reasoning has to be done with respect to an *arbitrary* source context, while with *SCC* and *CCC* one can reason about a known source context. Technically, *RTC* does not imply *SCC*, since even if we instantiate *RTC* with $C_S \downarrow$ for C_T , what we obtain in the source is $\exists C'_S. C'_S [P] \rightsquigarrow t$, for some C'_S that is unrelated to the original C_S . Similarly, *RTC* does not imply *CCC*, which is strictly stronger than *SCC* under the natural assumption that $C_S \downarrow \approx C_S$.

2.2.2 Trace Model with Finite and Infinite Traces and Its Impact on Safety and Liveness

For studying safety and liveness, traces need a bit of structure. We do this by introducing a precise model of traces and of finite trace prefixes, which also forms the base of our Coq formalization. Our trace model is a non-trivial extension of the standard trace models used for studying safety and liveness of reactive systems [Alpern and Schneider 1985, Clarkson and Schneider 2010, Lamport 2002, Lamport and Schneider 1984, Manna and Pnueli 2012, Schneider 1997], since (1) we need to balance the strength of the properties we preserve and the optimizations the compiler can still perform; and (2) we are interested in securely compiling both terminating and non-terminating programs. The extension of the trace model directly impacts the meaning of safety, which we try to keep as natural as possible, and also created the need for a new definition of *dense properties* to take the place of liveness.

The first departure from some of the previous work on trace properties [Lamport 2002, Lamport and Schneider 1984, Manna and Pnueli 2012, Schneider 1997] and hyperproperties [Clarkson and Schneider 2010] for reactive systems is that our traces are built from events, not from states. This is standard for formalizing correct compilation [Kumar et al. 2014, Leroy 2009a], where one wants to give the compiler enough freedom to perform optimizations by requiring it only to preserve relatively coarse-grained events such as input from and outputs to an external environment. For instance, the CompCert verified compiler [Leroy 2009a] follows the C standard and defines the result of a program to be a trace of all I/O and volatile operations it performs, plus an indication of whether and how it terminates.¹

The *events* in our traces are drawn from an arbitrary nonempty set (and a few of our results require at least 2 events). Intuitively, traces t are finite or infinite lists of events, where a finite trace means that the program terminates or enters an unproductive infinite loop after producing all the events in the list. This is natural for usual programming languages where most programs do indeed terminate and is standard for verified compilers like CompCert. This constitutes a second non-trivial extension of the trace model usually considered for abstract modeling of reactive systems (e.g., in a transition system or a process calculus), which looks only at infinite traces [Clarkson and Schneider 2010, Lamport 2002, Manna and Pnueli 2012, Schneider 1997].

Safety Properties For defining safety properties the main ingredient is a definition of finite trace prefixes, which capture the finite observations that can be made about an execution, for instance by a reference monitor. A reference monitor can generally observe that the program has terminated, so in our extended trace model finite trace prefixes are lists of events in which it is observable whether a prefix is terminated and can no longer be extended or if it is not yet terminated and can still be extended with more events. Moreover, we take the stance that while termination and silent divergence are two different terminal trace events, no observer can distinguish the two in finite time, since one cannot tell whether a program that seems

¹Our trace model is close to that of CompCert, but as opposed to CompCert, in this thesis we use the word “trace” for the result of a single program execution and later “behavior” for the set of all traces of a program (§2.3).

to be looping will *eventually* terminate. Technically, in our model finite trace prefixes m are lists with two different nil constructors: \bullet for terminated prefixes and \circ for not yet terminated prefixes. In contrast, traces can end either with \bullet if the program terminates or with \circ if the program silently diverges, or they can go on infinitely. The prefix relation $m \leq t$ is defined between a finite prefix m and a trace t according to the intuition above: $\bullet \leq \bullet$, $\circ \leq t$ for any t , and $e \cdot m' \leq e \cdot t'$ whenever $m' \leq t'$ (we write \cdot for concatenation).

The definition of safety properties is unsurprising for this trace model:

$$\text{Safety} \triangleq \{\pi \in 2^{\text{Trace}} \mid \forall t \notin \pi. \exists m \leq t. \forall t' \geq m. t' \notin \pi\}$$

A trace property π is *Safety* if, within any trace t that violates π , there exists a finite “bad prefix” m that can only be extended to traces t' that also violate π . For instance, the trace property $\pi_{\square \neg e} = \{t \mid e \notin t\}$, stating that the bad event e never occurs in the trace, is *Safety*, since for every trace t violating $\pi_{\square \neg e}$ there exists a finite prefix $m = m' \cdot e \cdot \circ$ (some prefix m' followed by e and then by the unfinished prefix symbol \circ) that is a prefix of t and so that every trace extending m still contains e , so it still violates $\pi_{\square \neg e}$. Similarly, $\pi_{\square \neg \bullet} = \{t \mid \bullet \notin t\}$ is a safety property that rejects all terminating traces and only accepts the non-terminating ones. This being safety crucially relies on allowing \bullet in the finite trace prefixes: For any finite trace t rejected by $\pi_{\square \neg \bullet}$ there exists a bad prefix $m = m' \cdot \bullet$ so that all extensions of m are also rejected by $\pi_{\square \neg \bullet}$; where this last part is trivial since the prefix m is terminating (ends with \bullet) and can thus only be extended to t itself. Finally, the trace property $\pi_{\diamond e}^{\text{term}} = \{t \mid t \text{ terminating} \Rightarrow e \in t\}$, stating that in every *terminating* trace the event e must eventually happen, is also a safety property in our model, since for each terminating trace $t = m' \cdot \bullet$ violating $\pi_{\diamond e}^{\text{term}}$ there exists a bad prefix $m = t$ that can only be extended to traces $t' = t$ that also violate $\pi_{\diamond e}^{\text{term}}$. In general, all trace properties like $\pi_{\square \neg \bullet}$ and $\pi_{\diamond e}^{\text{term}}$ that only reject terminating traces and thus allow all non-terminating traces are safety properties in our model; i.e., if $\forall t \text{ non-terminating}. t \in \pi$ then π is safety. So the trace property $\pi_S = \pi \cup \{t \mid t \text{ non-terminating}\}$ is safety for any π .

Dense Properties In our trace model the liveness definition of Alpern and Schneider [1985] does not have its intended intuitive meaning, so instead we focus on the main properties that the Alpern and Schneider liveness definition satisfies in the infinite trace model and, in particular, that each trace property can be decomposed as the intersection of a safety property and a liveness property. We discovered that in our model the following surprisingly simple notion of *dense properties* satisfies all the characterizing properties of liveness and is, in fact, uniquely determined by these properties and the definition of safety above:

$$\text{Dense} \triangleq \{\pi \in 2^{\text{Trace}} \mid \forall t \text{ terminating}. t \in \pi\}$$

We say that a trace property π is *Dense* if it allows all terminating traces; or, conversely, it can only be violated by non-terminating traces. For instance, the trace property $\pi_{\diamond e}^{\neg \text{term}} = \{t \mid t \text{ non-terminating} \Rightarrow e \in t\}$, stating that the good event e will eventually happen along every non-terminating trace is a dense property, since it accepts all terminating traces. The property $\pi_{\square \neg \circ} = \{t \mid \circ \notin t\} = \{t \mid t \text{ non-terminating} \Rightarrow t \text{ infinite}\}$ stating that the program does not silently diverge is also dense. Similarly, $\pi_{\square \neg \diamond e}^{\neg \text{term}} = \{t \mid t \text{ non-terminating} \Rightarrow t \text{ infinite} \wedge$

$\forall m. \exists m'. m \cdot m' \cdot e \leq t$ is dense and states that the event e happens infinitely often in any non-terminating trace. The trace property $\pi_{\diamond\bullet} = \{t \mid t \text{ terminating}\}$, which only contains all the terminating traces and thus rejects all the non-terminating traces, is the minimal dense property in our model. Finally, any property becomes dense in our model if we change it to allow all terminating traces: i.e., $\pi_L = \pi \cup \{t \mid t \text{ terminating}\}$ is dense for any π . For instance, while $\pi_{\square \neg e}$ is safety, the following dense property $\pi_{\square \neg e}^{\text{term}}$ states that event e never occurs along the non-terminating traces: $\pi_{\square \neg e}^{\text{term}} = \{t \mid t \text{ non-terminating} \Rightarrow e \notin t\}$.

We have proved that our definition of dense properties satisfies the good properties of Alpern and Schneider's liveness [Alpern and Schneider 1985], including their topological characterization, and in particular that any trace property can be decomposed as the intersection of a safety property and of a dense property (QED). For instance, the trace property $\pi_{\diamond e} = \{t \mid e \in t\}$ that in our model is neither safety nor dense, decomposes as the intersection of $\pi_{\diamond e}^{\text{term}}$ (which is safety) and $\pi_{\diamond e}^{\neg \text{term}}$ (which is dense). The proof of this decomposition theorem is in fact very simple in our model: Given any trace property π , define $\pi_S = \pi \cup \{t \mid t \text{ non-terminating}\}$ and $\pi_D = \pi \cup \{t \mid t \text{ terminating}\}$. As discussed above, $\pi_S \in \text{Safety}$ and $\pi_D \in \text{Dense}$. Finally, $\pi_S \cap \pi_D = (\pi \cup \{t \mid t \text{ non-terminating}\}) \cap (\pi \cup \{t \mid t \text{ terminating}\}) = \pi$.

Moreover, we have proved that our definition of dense properties is uniquely determined given our trace model, our definition of safety, and the following 3 properties:

1. Every trace property can be written as the intersection of a safety property and a dense property: $\forall \pi \in 2^{\text{Trace}}. \exists \pi_S \in \text{Safety}. \exists \pi_D \in \text{Dense}. \pi = \pi_S \cap \pi_D$
2. *Safety* and *Dense* are nearly disjoint: $\text{Safety} \cap \text{Dense} = \{\pi \mid \forall t. t \in \pi\}$
3. Dense properties cannot be empty: $\forall \pi \in \text{Dense}. \exists t. t \in \pi$

What does not hold in our model though, is that any trace property can also be decomposed as the intersection of two liveness properties [Alpern and Schneider 1985], since this rather counterintuitive decomposition seems to crucially rely on all traces of the system being infinite.

Finally, in case one wonders about the relation between dense properties and the liveness definition of Alpern and Schneider [1985], the two are in fact equivalent *in our model*, but this seems to be a coincidence and only happens because the Alpern and Schneider definition completely loses its original intent in our model, as the following theorem and simple proof suggests:

Theorem 2. $\forall \pi \in 2^{\text{Trace}}. \pi \in \text{Dense} \iff \forall m. \exists t. m \leq t \wedge t \in \pi$



For showing the \Rightarrow direction take some $\pi \in \text{Dense}$ and some finite prefix m . We can construct $t_{m\bullet}$ from m by simply replacing any final \circ with \bullet . By definition $m \leq t_{m\bullet}$ and moreover, since $t_{m\bullet}$ is terminating and $\pi \in \text{Dense}$, we can conclude that $t \in \pi$. For showing the \Leftarrow direction take some $\pi \in 2^{\text{Trace}}$ and some terminating trace t ; since t is terminating we can choose $m = t$ and since this finite prefix extends only to t we immediately obtain $t \in \pi$.

2.2.3 Robust Safety Property Preservation (*RSP*)

Robust safety preservation is an interesting criterion for secure compilation because it is easier to achieve and prove than most criteria of Figure 2.1, while still being quite expressive [Gordon and Jeffrey 2004, Swasey et al. 2017].

The definition of *RSP* simply restricts the preservation of robust satisfaction from all trace properties in *RTP* to only the safety properties; otherwise the definition is exactly the same:

$$RSP : \quad \forall \pi \in \text{Safety}. \forall P. (\forall C_S t. C_S [P] \rightsquigarrow t \Rightarrow t \in \pi) \Rightarrow (\forall C_T t. C_T [P \downarrow] \rightsquigarrow t \Rightarrow t \in \pi)$$

One might wonder how one can get safety properties to be robustly satisfied in the source, given that the execution traces can potentially be influenced not only by the partial program but also by the adversarial context, who could cause “bad events” to happen. A first alternative is for the semantics of the source language to simply prevent the context from producing any events, maybe other than termination, as we do in the compilation chain from §2.6, or be more fine-grained and prevent the context from producing only certain privileged events. With this alternative the source program will robustly satisfy safety properties over the privileged events the context cannot produce, but then the compilation chain needs to sandbox [Tan 2017, Wahbe et al. 1993] the context to make sure that it can only produce non-privileged events. A second alternative is for the source semantics to record enough information in the trace so that one can determine the originator of each event (as done for instance by the informative traces of §2.6.4); then safety properties can explicitly talk only about the events of the program, not the ones of the context. With this second alternative the compilation chain does not need to restrict the context from producing certain events, but the obtained global guarantees are weaker, e.g., one cannot enforce that the whole program does not cause a dangerous system call, only that the trusted partial program cannot be tricked into causing it.

The equivalent property-free characterization for *RSP* (\mathfrak{A}) simply requires one to back-translate a program (P), a target context (C_T), and a *finite* bad execution prefix ($C_T [P \downarrow] \rightsquigarrow m$) into a source context (C_S) producing the same finite trace prefix (m) in the source ($C_S [P] \rightsquigarrow m$):

$$RSC : \quad \forall P. \forall C_T. \forall m. C_T [P \downarrow] \rightsquigarrow m \Rightarrow \exists C_S. C_S [P] \rightsquigarrow m$$

Syntactically, the only change with respect to *RTC* is the switch from whole traces t to finite trace prefixes m . Similarly to *RTC*, we can pick a different context C_S for each execution $C_T [P \downarrow] \rightsquigarrow m$, which in our formalization we define generically as $\exists t \geq m. W \rightsquigarrow t$. The fact that for *RSC* these are *finite* execution prefixes can significantly simplify the back-translation task, since we can produce a source context only from this finite execution prefix. In fact, in §2.6.4 we produce a single source context in a fairly generic way even from a *finite set* of (related) finite execution prefixes.

Finally, we have proved that *RTP* *strictly implies* *RSP* (\mathfrak{A}). The implication follows immediately from safety properties being trace properties, but showing the lack of an implication from *RSP* to *RTP* is more interesting and involves constructing a counterexample compilation chain. We take any target language that can produce infinite traces. We take the source language to be

a variant of the target with the same partial programs, but where we extend whole programs and contexts with a bound on the number of events they can produce before being terminated. Compilation simply erases this bound. (While this construction might seem artificial, languages with a fuel bound are gaining popularity [Wood 2014].) This compilation chain satisfies *RSP* but not *RTP*. To show that it satisfies *RSP*, we simply back-translate a target context \mathbf{C}_T and a finite trace prefix m to a source context $(\mathbf{C}_T, \text{length}(m))$ that uses the length of m as the correct bound, so this context can still produce m in the source without being prematurely terminated. However, this compilation chain does not satisfy *RTP*, since in the source all executions are finite, so all dense properties are vacuously satisfied, which is clearly not the case in the target, where we also have infinite executions.

2.2.4 Robust Dense Property Preservation (*RDP*)

RDP simply restricts *RTP* to only the dense properties:

$$RDP : \quad \forall \pi \in \text{Dense}. \forall \mathbf{P}. (\forall \mathbf{C}_S t. \mathbf{C}_S [\mathbf{P}] \rightsquigarrow t \Rightarrow t \in \pi) \Rightarrow (\forall \mathbf{C}_T t. \mathbf{C}_T [\mathbf{P} \downarrow] \rightsquigarrow t \Rightarrow t \in \pi)$$

Again, one might wonder how one can get dense properties to be robustly satisfied in the source and then preserved by compilation. Enforcing that the context is responsive or eventually gives back control along the infinite traces seems often difficult, but one can still imagine devising enforcement mechanisms for this, for instance running the context in a separate process that gets terminated or preempted if a certain amount of time has passed, and then exposing such asynchronous programming in the source language. Alternatively, one can keep the source language unchanged but make the traces informative enough to identify the actions of the program and of the context, so that we can guarantee that the program satisfies dense properties like being responsive, even if the context does not.

The property-free variant of *RDP* restricts *RTC* to only back-translating *non-terminating* traces:

$$RDC : \quad \forall \mathbf{P}. \forall \mathbf{C}_T. \forall t \text{ non-terminating}. \mathbf{C}_T [\mathbf{P} \downarrow] \rightsquigarrow t \Rightarrow \exists \mathbf{C}_S. \mathbf{C}_S [\mathbf{P}] \rightsquigarrow t$$

In contrast to *RSC*, we are not aware of good ways to make use of the *infinite* execution $\mathbf{C}_T [\mathbf{P} \downarrow] \rightsquigarrow t$ to produce a *finite* context \mathbf{C}_T , so the back-translation for *RDC* will likely have to use \mathbf{C}_T and \mathbf{P} .

Finally, we have proved that *RTP* strictly implies *RDP* (\Rightarrow). The counterexample compilation chain we use for showing the separation is roughly the inverse of the one we used for *RSP*. We take the source to be arbitrary, with the sole assumption that there exists a program \mathbf{P}_Ω that can produce a single infinite trace w irrespective of the context. We compile programs by simply pairing them with a constant bound on the number of steps, i.e., $\mathbf{P} \downarrow = (\mathbf{P}, k)$. On the one hand, *RDC* holds vacuously, as target programs cannot produce infinite traces. On the other hand, this compilation chain does not have *RTP*, since the property $\pi = \{w\}$ is robustly satisfied by \mathbf{P}_Ω in the source but not by its compilation (\mathbf{P}_Ω, k) in the target.

This separation result does not hold in models with only infinite traces, wherein any trace property can be decomposed as the intersection of two liveness properties [Alpern and Schneider 1985]. In such a model, *Robust Liveness Property Preservation* and *RTP* collapse.

From the decomposition into safety and liveness from §2.2.2 and the fact that *RDP* does not imply *RTP*, it follows that *RDP* also does not imply *RSP*. Similarly, *RSP* does not imply *RDP*.

2.3 Robustly Preserving Classes of Hyperproperties

So far, we have studied the robust preservation of trace properties, which are properties of *individual* traces of a program. In this section we generalize this to *hyperproperties*, which are properties of *multiple* traces of a program [Clarkson and Schneider 2010]. The most well-known hyperproperty is noninterference, which has many variants [Askarov et al. 2008, Goguen and Meseguer 1982, McLean 1992, Sabelfeld and Myers 2003, Zdancewic and Myers 2003], but usually requires considering two traces of a program that differ on secret inputs. Another hyperproperty is bounded mean response time over all executions. We study the robust preservation of various subclasses of hyperproperties: all hyperproperties (§2.3.1), subset-closed hyperproperties (§2.3.2), hypersafety and *K*-hypersafety (§2.3.3), and hyperliveness (§2.3.4). The corresponding secure compilation criteria are outlined in red in Figure 2.1.

2.3.1 Robust Hyperproperty Preservation (*RHP*)

While trace properties are sets of traces, hyperproperties are sets of sets of traces [Clarkson and Schneider 2010]. If we call the set of traces of a whole program W the *behavior* of W ($\text{Behav}(W) = \{t \mid W \rightsquigarrow t\}$) then a hyperproperty is a set of allowed behaviors. We say that W satisfies hyperproperty H if the behavior of W is a member of the set H (i.e., $\text{Behav}(W) \in H$, or, if we unfold, $\{t \mid W \rightsquigarrow t\} \in H$). Contrast this to W satisfying trace property π , which holds if the behavior of W is a subset of the set π (i.e., $\text{Behav}(W) \subseteq \pi$, or, if we unfold, $\forall t. W \rightsquigarrow t \Rightarrow t \in \pi$). So while a trace property determines whether each individual trace of a program should be allowed or not, a hyperproperty determines whether the set of traces of a program, its behavior, should be allowed or not. For instance, the trace property $\pi_{123} = \{t_1, t_2, t_3\}$ is satisfied by programs with behaviors such as $\{t_1\}$, $\{t_2\}$, $\{t_2, t_3\}$, and $\{t_1, t_2, t_3\}$, but a program with behavior $\{t_1, t_4\}$ does not satisfy π_{123} . A hyperproperty like $H_{1+23} = \{\{t_1\}, \{t_2, t_3\}\}$ is satisfied only by programs with behavior $\{t_1\}$ or with behavior $\{t_2, t_3\}$. A program with behavior $\{t_2\}$ does not satisfy H_{1+23} , so hyperproperties can express that if some traces (e.g., t_2) are possible then some other traces (e.g., t_3) should also be possible. A program with behavior $\{t_1, t_2, t_3\}$ also does not satisfy H_{1+23} , so hyperproperties can express that if some traces (e.g., t_2 and t_3) are possible then some other traces (e.g., t_1) should not be possible. Finally, trace properties can be easily lifted to hyperproperties: A trace property π becomes the hyperproperty $[\pi] = 2^\pi$, which is just the powerset of π .

We say that a partial program P *robustly satisfies* a hyperproperty H if it satisfies H for any context C . Given this we define RHP as the preservation of robust satisfaction of arbitrary hyperproperties:

$$RHP : \quad \forall H \in 2^{2^{Trace}}. \forall P. (\forall C_S. \text{Behav}(C_S[P]) \in H) \Rightarrow (\forall C_T. \text{Behav}(C_T[P\downarrow]) \in H)$$

The equivalent property-free characterization of RHP (⌚) is not very surprising:

$$\begin{aligned} RHC : \quad & \forall P. \forall C_T. \exists C_S. \text{Behav}(C_T[P\downarrow]) = \text{Behav}(C_S[P]) \\ RHC : \quad & \forall P. \forall C_T. \exists C_S. \forall t. C_T[P\downarrow] \rightsquigarrow t \iff C_S[P] \rightsquigarrow t \end{aligned}$$

This requires that, each partial program P and target context C_T can be back-translated to a source context C_S in a way that perfectly preserves the set of traces produced when linking with P and $P\downarrow$ respectively. There are two differences from RTP : (1) the $\exists C_S$ and $\forall t$ quantifiers are swapped, so now one needs to produce a C_S that works for all traces t , and (2) the implication in RTP (\Rightarrow) became a two-way implication in RHP (\iff), so compilation has to perfectly preserve the set of traces. Because of point (2), if the source language is nondeterministic a compilation chain satisfying RHP cannot refine this nondeterminism, e.g., it cannot implement nondeterministic scheduling via an actual deterministic scheduler, etc.

In the following subsections we study restrictions of RHP to various sub-classes of hyperproperties. To prevent duplication we define $RHP(X)$ to be the robust satisfaction of a class X of hyperproperties (so RHP above is simply $RHP(2^{2^{Trace}})$):

$$RHP(X) : \quad \forall H \in X. \forall P. (\forall C_S. \text{Behav}(C_S[P]) \in H) \Rightarrow (\forall C_T. \text{Behav}(C_T[P\downarrow]) \in H)$$

2.3.2 Robust Subset-Closed Hyperproperty Preservation ($RSCHP$)

If one restricts robust preservation to only subset-closed hyperproperties then refinement of nondeterminism is again allowed. A hyperproperty H is subset-closed, written $H \in SC$, if for any two behaviors b_1 and b_2 so that $b_1 \subseteq b_2$, if $b_2 \in H$ then $b_1 \in H$. For instance, the lifting $[\pi]$ of any trace property π is subset-closed, but the hyperproperty H_{1+23} above is not. It can be made subset-closed by allowing all smaller behaviors: $H_{1+23}^{SC} = \{\emptyset, \{t_1\}, \{t_2\}, \{t_3\}, \{t_2, t_3\}\}$ is subset-closed, although it is not the lifting of a trace property (i.e., not a powerset).

Robust Subset-Closed Hyperproperty Preservation ($RSCHP$) is simply defined as $RHP(SC)$. The equivalent property-free characterization of $RSCHC$ (⌚) simply gives up the \Leftarrow direction of RHP :

$$RSCHC : \quad \forall P. \forall C_T. \exists C_S. \forall t. C_T[P\downarrow] \rightsquigarrow t \Rightarrow C_S[P] \rightsquigarrow t$$

The most interesting sub-class of subset-closed hyperproperties is hypersafety, which we discuss in the next sub-section. The appendix also introduces and studies a series of sub-classes we call K -subset-closed hyperproperties that can be seen as generalizing K -hypersafety below.

2.3.3 Robust Hypersafety Preservation (RHSP)

Hypersafety is a generalization of safety that is very important in practice, since several important notions of noninterference are hypersafety, such as termination-insensitive noninterference [Askarov et al. 2008, Fenton 1974, Sabelfeld and Sands 2001], observational determinism [McLean 1992, Roscoe 1995, Zdancewic and Myers 2003], and nonmalleable information flow [Cecchetti et al. 2017].

According to Alpern and Schneider [Alpern and Schneider 1985], the “bad thing” that a safety property disallows must be *finitely observable* and *irremediable*. For safety the “bad thing” is a finite trace prefix that cannot be extended to any trace satisfying the safety property. For hypersafety, Clarkson and Schneider [2010] generalize the “bad thing” to a finite set of finite trace prefixes that they call an *observation*, drawn from the set $Obs = 2_{Fin}^{FinPref}$, which denotes the set of all *finite* subsets of finite prefixes. They then lift the prefix relation to sets: an observation $o \in Obs$ is a prefix of a behavior $b \in 2^{Trace}$, written $o \leq b$, if $\forall m \in o. \exists t \in b. m \leq t$. Finally, they define hypersafety analogously to safety, but the domains involved include an extra level of sets:

$$Hypersafety \triangleq \{H \mid \forall b \notin H. (\exists o \in Obs. o \leq b \wedge (\forall b' \geq o. b' \notin H))\}$$

Here the “bad thing” is an observation o that cannot be extended to a behavior b' satisfying the hypersafety property H . We use this to define *Robust Hypersafety Preservation (RHSP)* as $RHP(Hypersafety)$ and propose the following equivalent characterization for it (♣):

$$RHSC : \quad \forall P. \forall C_T. \forall o \in Obs. o \leq \text{Behav}(C_T[P\downarrow]) \Rightarrow \exists C_S. o \leq \text{Behav}(C_S[P])$$

This says that to prove *RHSP* one needs to be able to back-translate a partial program P , a context C_T , and a prefix o of the behavior of $C_T[P\downarrow]$, to a source context C_S so that the behavior of $C_S[P]$ extends o . It is possible to use the finite set of finite executions corresponding to observation o to drive this back-translation, as we illustrate in §2.6.4 for a stronger criterion.

For hypersafety the involved observations are finite sets but their cardinality is otherwise unrestricted. In practice though, most hypersafety properties can be falsified by very small observations: counterexamples to termination-insensitive noninterference [Askarov et al. 2008, Fenton 1974, Sabelfeld and Sands 2001] and observational determinism [McLean 1992, Roscoe 1995, Zdancewic and Myers 2003] are observations containing 2 finite prefixes, while counterexamples to nonmalleable information flow [Cecchetti et al. 2017] are observations containing 4 finite prefixes. To account for this, Clarkson and Schneider [2010] introduce K -hypersafety as a restriction of hypersafety to observations of a fixed cardinality K . Given $Obs_K = 2_{Fin(K)}^{FinPref}$, the set of observations with cardinality K , all definitions and results above can be ported to K -hypersafety by simply replacing Obs with Obs_K .

The set of lifted safety properties, $\{[\pi] \mid \pi \in \text{Safety}\}$, is precisely the same as 1-hypersafety, since the counterexample for them is a single finite prefix. For a more interesting example, termination-insensitive noninterference (*TINI*) [Askarov et al. 2008, Fenton 1974, Sabelfeld and Sands 2001] can be defined as follows in our setting:

$$TINI \triangleq \{b \mid \forall t_1, t_2 \in b. (t_1 \text{ terminating} \wedge t_2 \text{ terminating})\}$$

$$\begin{aligned} & \wedge \text{pub-inputs}(t_1) = \text{pub-inputs}(t_2) \\ & \Rightarrow \text{pub-events}(t_1) = \text{pub-events}(t_2) \end{aligned}$$

This requires that trace events are either inputs or outputs, each of them associated with a security level: public or secret. *TINI* ensures that for any two terminating traces of the program behavior for which the two sequences of public inputs are the same, the two sequences of public events—inputs and outputs—are also the same. *TINI* is in 2-hypersafety, since $b \notin \text{TINI}$ implies that there exist finite traces t_1 and t_2 that agree on the public inputs but not on all public events, so we can simply take $o = \{t_1, t_2\}$. Since the traces in o end with \bullet any extension b' of o can only add extra traces, i.e., $\{t_1, t_2\} \subseteq b'$, so $b' \notin \text{TINI}$ as needed to conclude that *TINI* is in 2-hypersafety. In Figure 2.1, we write *Robust Termination-Insensitive Noninterference Preservation* (*RTINIP*) for $\text{RHP}(\{\text{TINI}\})$.

Enforcing *RHSP* is strictly more demanding than enforcing *RSP*. Because even *R2HSP* implies the preservation of noninterference properties like *TINI*, a compilation chain satisfying *R2HSP* has to make sure that a target-level context cannot infer more information from the internal state of $\mathbf{P}\downarrow$ than a source context could infer from the state of \mathbf{P} . By contrast, a *RSP* compilation chain can allow arbitrary reads of $\mathbf{P}\downarrow$'s internal state, even if \mathbf{P} 's state is private at the source level. Intuitively, for proving *RSC*, the source context produced by back-translation can guess any secret $\mathbf{P}\downarrow$ receives in the *single* considered execution, but for *R2HSP* the single source context needs to work for *two* different executions, potentially with two different secrets, so guessing is no longer an option. We use this to prove a separation result between *RHSP* and *RSP*, by exhibiting a toy compilation chain in which private variables are readable in the target language, but not in source. This compilation chain satisfies *RSP* but not *R2HSP*. Using a more complex counterexample involving a system of K linear equations, we have also shown that, for any K , robust preservation of K -hypersafety (*RKHSP*), does not imply robust preservation of $(K+1)$ -hypersafety ($\text{R}(K+1)\text{HSP}$).

2.3.4 Where is Robust Hyperliveness Preservation?

Robust Hyperliveness Preservation (*RHLP*) does not appear in Figure 2.1, because it is equivalent to *RHP*. We define *RHLP* as $\text{RHP}(\text{Hyperliveness})$ for the following standard definition of *Hyperliveness* [Clarkson and Schneider 2010]:

$$\text{Hyperliveness} \triangleq \{H \mid \forall o \in \text{Obs}. \exists b \geq o. b \in H\}$$

The proof that *RHLP* implies *RHC* (\clubsuit) involves showing that $\{b \mid b \neq \text{Behav}(\mathbf{C}_T[\mathbf{P}\downarrow])\}$, the hyperproperty allowing all behaviors other than $\text{Behav}(\mathbf{C}_T[\mathbf{P}\downarrow])$, is hyperliveness. Another way to obtain this result is from the fact that, as in previous models [Alpern and Schneider 1985], each hyperproperty can be decomposed as the intersection of two hyperliveness properties. This collapse of preserving hyperliveness and preserving all hyperproperties happens irrespective of the adversarial contexts.

2.4 Robustly Preserving Classes of Relational Hyperproperties

So far, we have described the robust preservation of trace properties and hyperproperties, which are *predicates* on the behavior of a single program. However, we may be interested in showing that compilation robustly preserves *relations* between the behaviors of two or more programs. For example, suppose we hand-optimize a partial source program P_1 to a partial source program P_2 and we reason in the source semantics that P_2 runs faster than P_1 in any source context. We may want compilation to preserve this “runs faster than” *relation* between the two program behaviors even against arbitrary target contexts. Similarly, we may reason that in any source context the behavior (i.e., set of traces) of P_1 is the same as that of P_2 and then want secure compilation to preserve such trace equivalence [Baelde et al. 2017, Cheval et al. 2018, Delaune and Hirschi 2017] against arbitrary target contexts. This last criterion, which we call *Robust Trace Equivalence Preservation (RTEP)* in Figure 2.1, is interesting because in various determinate settings [Cheval et al. 2013, Engelfriet 1985] it coincides with preserving observational equivalence, i.e., the direction of full abstraction interesting for security, as discussed in §2.5.

In this section, we study the robust preservation of such *relational hyperproperties* and several interesting subclasses, all of which are predicates on the behaviors of multiple programs. Unlike hyperproperties and trace properties, relational hyperproperties have not been defined as a general concept in the literature, so even their definitions are new. We describe relational hyperproperties and their robust preservation in §2.4.1, then look at a subclass called *relational properties* in §2.4.2, and a smaller subclass, *relational safety properties*, in §2.4.3. The corresponding secure compilation criteria are highlighted in blue in Figure 2.1. In §2.4.4 we show that none of these relational criteria are implied by any non-relational criterion (from §2.2 and §2.3).

2.4.1 Relational Hyperproperty Preservation (RrHP)

Recall that the behavior $\text{Behav}(P)$ of a program P is the set of all traces of P and let $\text{Behavs} = 2^{\text{Trace}}$ be the set of all possible behaviors. We define a *relational hyperproperty* as a predicate (relation) on a list of behaviors. A list of programs is then said to have the relational hyperproperty if their respective behaviors satisfy the predicate. Depending on the arity of the predicate, we get different subclasses of relational hyperproperties. For arity 1, the resulting subclass describes relations on the behavior of individual programs, which coincides with hyperproperties defined in §2.3. For arity 2, the resulting subclass consists of relations on the behaviors of pairs of programs. Both examples described at the beginning of this section lie in this subclass. This can then be generalized to any finite arity K (predicates on behaviors of K programs), and to the infinite arity (predicates on all programs of the language).

Next, we define the robust preservation of these subclasses. For arity 2, *robust 2-relational hyperproperty preservation*, $R2rHP$, is defined as follows:

$$R2rHP : \quad \forall R \in 2^{(\text{Behavs}^2)}. \forall P_1 P_2. (\forall C_S. (\text{Behav}(C_S[P_1]), \text{Behav}(C_S[P_2])) \in R) \Rightarrow$$

$$(\forall \mathbf{C}_T. (\text{Behav}(\mathbf{C}_T [\mathbf{P}_1 \downarrow]), \text{Behav}(\mathbf{C}_S [\mathbf{P}_2 \downarrow])) \in R)$$

R2rHP says that for any binary relation R on behaviors of programs, if the behaviors of \mathbf{P}_1 and \mathbf{P}_2 satisfy R in every source context, then so do the behaviors of $\mathbf{P}_1 \downarrow$ and $\mathbf{P}_2 \downarrow$ in every target context. In other words, a compiler satisfies *R2rHP* iff it preserves any relation between pairs of program behaviors that hold in all contexts. In particular, such a compilation chain preserves trace equivalence in all contexts (i.e., *RTEP*), which we obtain by instantiating R with equality in the above definition (\mathfrak{E}). Similarly, such a compiler preserves the may- and must-testing equivalences [De Nicola and Hennessy 1984]. If execution time is recorded on program traces, then such a compiler also preserves relations like “the average execution time of \mathbf{P}_1 across all inputs is no more than the average execution time of \mathbf{P}_2 across all inputs” and “ \mathbf{P}_1 runs faster than \mathbf{P}_2 on all inputs” (i.e., \mathbf{P}_1 is an improvement of \mathbf{P}_2). The last property can also be described as a relational predicate on traces (rather than behaviors); we return to this point in §2.4.2.

Like all our earlier definitions, *R2rHP* has an equivalent (\mathfrak{E}) property-free characterization that does not mention the relations R :

$$\begin{aligned} R2rHC : \forall \mathbf{P}_1 \mathbf{P}_2 \mathbf{C}_T. \exists \mathbf{C}_S. & \text{Behav}(\mathbf{C}_T [\mathbf{P}_1 \downarrow]) = \text{Behav}(\mathbf{C}_S [\mathbf{P}_1]) \wedge \\ & \text{Behav}(\mathbf{C}_T [\mathbf{P}_2 \downarrow]) = \text{Behav}(\mathbf{C}_S [\mathbf{P}_2]) \end{aligned}$$

R2rHC is a direct generalization of *RHC* from §2.3.1. Different from *RHC* is the requirement that the same source context \mathbf{C}_S simulate the behaviors of two target programs, $\mathbf{C}_T [\mathbf{P}_1 \downarrow]$ and $\mathbf{C}_T [\mathbf{P}_2 \downarrow]$.

R2rHP generalizes to any finite arity K in the obvious way, yielding *RKrHP*. Finally, *R2rHP* generalizes to the infinite arity. We call this *Robust Relational Hyperproperty Preservation* (*RrHP*); it robustly preserves relations over the behaviors of *all* programs of the source language or, equivalently, relations over *functions* from source programs (drawn from the set \mathbf{Progs}) to behaviors.

$$\begin{aligned} RrHP : \forall R \in 2^{(\mathbf{Progs} \rightarrow \mathbf{Behavs})}. & (\forall \mathbf{C}_S. (\lambda \mathbf{P}. \text{Behav}(\mathbf{C}_S [\mathbf{P}])) \in R) \Rightarrow \\ & (\forall \mathbf{C}_T. (\lambda \mathbf{P}. \text{Behav}(\mathbf{C}_T [\mathbf{P} \downarrow])) \in R) \end{aligned}$$

RrHP is the strongest criterion we study and, hence, it is the highest point in the partial order of Figure 2.1. It also has an equivalent (\mathfrak{E}) property-free characterization, *RrHC*, requiring for every target context \mathbf{C}_T , a source context \mathbf{C}_S that can simulate the behavior of \mathbf{C}_T for *any* program:

$$RrHC : \forall \mathbf{C}_T. \exists \mathbf{C}_S. \forall \mathbf{P}. \text{Behav}(\mathbf{C}_T [\mathbf{P} \downarrow]) = \text{Behav}(\mathbf{C}_S [\mathbf{P}])$$

It is instructive to compare the property-free characterizations of the robust preservation of trace properties (*RTC*), hyperproperties (*RHC*) and relational hyperproperties (*RrHC*). In *RTC*, the source context \mathbf{C}_S may depend on the target context \mathbf{C}_T , the source program \mathbf{P} and a given trace t . In *RHC*, \mathbf{C}_S may depend only on \mathbf{C}_T and \mathbf{P} . In *RrHC*, \mathbf{C}_S may depend only on \mathbf{C}_T . This directly reflects the increasing expressive power of trace properties, hyperproperties, and relational hyperproperties, as predicates on traces, behaviors (set of traces), and program-indexed sets of behaviors (sets of sets of traces), respectively.

2.4.2 Relational Trace Property Preservation (RrTP)

Relational (trace) properties are the subclass of relational hyperproperties that are fully characterized by relations on traces of multiple programs. Specifically, a K -ary relational hyperproperty is a relational trace property if there is a K -ary relation R on *traces* such that P_1, \dots, P_K are related by the relational hyperproperty iff $(t_1, \dots, t_k) \in R$ for any $t_1 \in \text{Behav}(P_1), \dots, t_k \in \text{Behav}(P_K)$. For example, the relation “ P_1 runs faster than P_2 on every input” is a 2-ary relational property characterized by pairs of traces which either differ in the input or on which P_1 ’s execution time is less than that of P_2 . Relational properties of some arity are a subclass of relational hyperproperties of the same arity. Next, we define the robust preservation of relational properties of different arities. For arity 1, this coincides with *RTP* from §2.2.1. For arity 2, we define *Robust 2-relational Property Preservation* or *R2rTP* as follows.

$$\begin{aligned} R2rTP : \forall R \in 2^{(\text{Trace}^2)}. \forall P_1 P_2. (\forall C_S t_1 t_2. (C_S[P_1] \rightsquigarrow t_1 \wedge C_S[P_2] \rightsquigarrow t_2) \Rightarrow (t_1, t_2) \in R) \Rightarrow \\ (\forall C_T t_1 t_2. (C_T[P_1 \downarrow] \rightsquigarrow t_1 \wedge C_T[P_2 \downarrow] \rightsquigarrow t_2) \Rightarrow (t_1, t_2) \in R) \end{aligned}$$

R2rTP implies the robust preservation of relations like “ P_1 runs faster than P_2 on every input”. However, *R2rTP* is weaker than its relational hyperproperty counterpart, *R2rHP* (§2.4.1): Unlike *R2rHP*, *R2rTP* does not imply the robust preservation of relations like “the average execution time of P_1 across all inputs is no more than the average execution time of P_2 across all inputs” (a relation between average execution times of P_1 and P_2 cannot be characterized by any relation between individual traces of P_1 and P_2). We have also proved that *R2rTP* implies robust trace equivalence preservation (*RTEP*) for languages without internal nondeterminism, under standard conditions.

R2rTP also has an equivalent (\mathfrak{A}) property-free characterization, *R2rTC*.

$$\begin{aligned} R2rTC : \forall P_1 P_2. \forall C_T. \forall t_1 t_2. (C_T[P_1 \downarrow] \rightsquigarrow t_1 \wedge C_T[P_2 \downarrow] \rightsquigarrow t_2) \Rightarrow \\ \exists C_S. (C_S[P_1] \rightsquigarrow t_1 \wedge C_S[P_2] \rightsquigarrow t_2) \end{aligned}$$

Establishing *R2rTC* requires constructing a source context C_S that can simultaneously simulate a given trace of $C_T[P_1 \downarrow]$ and a given trace of $C_T[P_2 \downarrow]$.

R2rTP generalizes from arity 2 to any finite arity K in the obvious way. It also generalizes to the infinite arity, i.e., to the robust preservation of all relations on *all* programs of the language that can be characterized by individual traces or, equivalently, relations on functions from programs to traces. We call this *Robust Relational Trace Property Preservation* or *RrTP*. This and its equivalent (\mathfrak{A}) property-free characterization, *RrTC*, are defined as follows.

$$\begin{aligned} RrTP : \forall R \in 2^{(\text{Progs} \rightarrow \text{Trace})}. (\forall C_S. \forall f. (\forall P. C_S[P] \rightsquigarrow f(P)) \Rightarrow R(f)) \Rightarrow \\ (\forall C_T. \forall f. (\forall P. C_T[P \downarrow] \rightsquigarrow f(P)) \Rightarrow R(f)) \\ RrTC : \forall f : \text{Progs} \rightarrow \text{Trace}. \forall C_T. (\forall P. C_T[P \downarrow] \rightsquigarrow f(P)) \Rightarrow \exists C_S. (\forall P. C_S[P] \rightsquigarrow f(P)) \end{aligned}$$

In *RrTC*, the same context C_S must simulate one selected trace of every source program.

2.4.3 Robust Relational Safety Preservation (*RrSP*)

Relational safety properties are a subclass of relational trace properties, much as safety properties are a subclass of trace properties. Specifically, a K -ary relational hyperproperty is a K -ary relational safety property if there is a set M of size- K sets of trace prefixes with the following condition: P_1, \dots, P_K are *not* related by the hyperproperty iff P_1, \dots, P_K respectively have traces t_1, \dots, t_K such that $m \leq \{t_1, \dots, t_K\}$. Here, \leq is the lifted prefix relation from §2.3.3. This is quite similar to hypersafety, except that the “bad” traces $\{t_1, \dots, t_K\}$ come from different programs.

Relational safety properties are a natural generalization of safety and hypersafety properties to multiple programs, and an important subclass of relational trace properties. Several interesting relational trace properties are actually relational safety properties. For instance, if we restrict the earlier relational trace property “ P_1 runs faster than P_2 on all inputs” to terminating programs it becomes a relational safety property, characterized by pairs of bad prefixes in which both prefixes have the termination symbol, both prefixes have the same input, and the left prefix shows termination no earlier than the right prefix. In a setting without internal non-determinism (i.e., determinate [Engelfriet 1985, Leroy 2009a]) where, additionally, divergence is observable, trace equivalence in all contexts is also a 2-relational safety property, so robustly preserving all 2-relational safety properties (*R2rSP*) implies *RTEP* (♣).

Next, we define the robust preservation of relational safety properties for different arities. At arity 2, we define *robust 2-relational safety preservation* or *R2rSP* as follows.

$$\begin{aligned} R2rSP : \forall R \in 2^{(FinPref^2)}. \forall P_1 P_2. \\ (\forall C_S m_1 m_2. (C_S[P_1] \rightsquigarrow_{m_1} \wedge C_S[P_2] \rightsquigarrow_{m_2}) \Rightarrow (m_1, m_2) \in R) \Rightarrow \\ (\forall C_T m_1 m_2. (C_T[P_1\downarrow] \rightsquigarrow_{m_1} \wedge C_T[P_2\downarrow] \rightsquigarrow_{m_2}) \Rightarrow (m_1, m_2) \in R) \end{aligned}$$

In words: If all pairs of finite trace prefixes of source programs P_1, P_2 robustly satisfy a relation R , then so do all pairs of trace prefixes of the compiled programs $P_1\downarrow, P_2\downarrow$. R here represents the *complement* of the bad prefixes M in the definition of relational safety properties. So, this definition can also be read as saying that if all prefixes of P_1, P_2 in every context are good (for any definition of good), then so are all prefixes of $P_1\downarrow, P_2\downarrow$ in every context. The only difference from the stronger *R2rTP* (§2.4.2) is between considering full traces and only finite prefixes, and the same holds for the equivalent property-free characterization, *R2rSC* (♣).

Comparison of proof obligations We briefly compare the robust preservation of (variants of) relational hyperproperties (*RrHP*, §2.4.1), relational trace properties (*RrTP*, §2.4.2) and relational safety properties (*RrSP*, this subsection) in terms of the difficulty of back-translation proofs. For this, it is instructive to look at the property-free characterizations. In a proof of *RrSP* or any of its variants, we must construct a source context C_S that can induce a given set of *finite prefixes of traces*, one from each of the programs being related. In *RrTP* and its variants, this obligation becomes harder—now the constructed C_S must be able to induce a given set of *full traces*. In *RrHP* and its variants, the obligation is even harder— C_S must be able to induce

entire behaviors (sets of traces) from each of the programs being related. Thus, the increasing strength of $RrSP$, $RrTP$ and $RrHP$ is directly reflected in corresponding proof obligations.

Looking further at just the different variants of relational safety described in this subsection, we note that the number of trace prefixes the constructed context C_S must simultaneously induce in the source programs is exactly the arity of the relational property. Constructing C_S from a finite number of prefixes is much easier than constructing C_S from an infinite number of prefixes. Consequently, it is meaningful to define a special point in the partial order of Figure 2.1 that is the join of $RKrSP$ for all finite K s, which is the strongest preservation criterion that can be established by back-translating source contexts C_S starting from a *finite* number of trace prefixes. We call this *robust finite-relational safety preservation*, or $RFrSP$. Its property-free characterization, $RFrSC$, is shown below.

$$RFrSC : \forall K. \forall P_1 \dots P_K. \forall C_T. \forall m_1 \dots m_K. (C_T[P_1 \downarrow] \rightsquigarrow m_1 \wedge \dots \wedge C_T[P_K \downarrow] \rightsquigarrow m_K) \Rightarrow \\ \exists C_S. (C_S[P_1] \rightsquigarrow m_1 \wedge \dots \wedge C_S[P_K] \rightsquigarrow m_K)$$

We sketch an illustrative proof of $RFrSC$ in §2.6.4. In Figure 2.1, all criteria weaker than $RFrSP$ are highlighted in green.

2.4.4 Robust Non-Relational Preservation Doesn't Imply Robust Relational Preservation

Relational (hyper)properties (§2.4.1, §2.4.2) and hyperproperties (§2.3) are different but both have a “relational” nature: Relational (hyper)properties are relations on the behaviors or traces of multiple programs, while hyperproperties are relations on multiple traces of the same program. So one may wonder whether there is any case in which the *robust preservation* of a class of relational hyper(properties) is equivalent to that of a class of hyperproperties. Might it not be the case that a compiler that robustly preserves all hyperproperties (RHP , §2.3.1) also robustly preserves at least some class of 2-relational (hyper)properties?

This is, in fact, not the case— RHP does not imply the robust preservation of any subclass of relational properties that we have described in this section (except, of course, relational properties of arity 1, that are just hyperproperties). Since RHP is the strongest non-relational robust preservation criterion that we study, this also means that no non-relational robust preservation criterion implies any relational robust preservation criterion in Figure 2.1. In other words, all edges from relational to non-relational points in Figure 2.1 are strict implications.

To prove this, we construct a compiler that satisfies RHP , but does not have $R2rSP$, the weakest relational criterion in Figure 2.1.

Theorem 3. There is a compiler that satisfies RHP but not $R2rSP$.

Proof sketch. Consider a source language that lacks code introspection, and a target language which is exactly the same, but additionally has a primitive with which the context can read the code of the compiled partial program as data (and then analyze it). Consider the trivial

compiler that is syntactically the identity. It should be clear that this compiler satisfies *RHP* since the added operation of code introspection offers no advantage to the context when we consider properties of a single program (as is the case in *RHP*). More precisely, in establishing *RHC*, given a target context \mathbf{C}_T and a program P , we can construct a simulating source context \mathbf{C}_S by modifying \mathbf{C}_T to hard-code P wherever \mathbf{C}_T performs code introspection. (Note that \mathbf{C}_S can depend on P in *RHC*.)

Now consider two programs that differ only in some dead code, that both read a value from the context and write it back verbatim to the output. These two programs satisfy the relational safety property “the outputs of the two programs are equal” in any *source* context. However, there is a trivial *target* context that causes the compiled programs to break this relational property. This context reads the code of the program it is linked to, and provides 1 as input if it happens to be the first of our two programs and 2 otherwise. Consequently, in this target context, the two programs produce outputs 1 and 2 and do not have this relational safety property in all contexts. Hence, this compiler does not satisfy *R2rSP*. (Technically, the trick of hard-coding the program in \mathbf{C}_S no longer works since there are two different programs here.) \square

This proof provides a fundamental insight: To robustly preserve any subclass of relational (hyper)properties, compilation must ensure that target contexts cannot learn anything about the *syntactic program* they interact with beyond what source contexts can also learn (this requirement is in addition to everything needed to attain the robust preservation of the corresponding subclass of *non-relational* hyperproperties). When the target language is low-level, hiding code attributes can be difficult: it may require padding the code segment of the compiled program to a fixed size, and cleaning or hiding any code-layout-dependent data like code pointers from memory and registers when passing control to the context. These complex protections are not necessary for any non-relational preservation criteria (even *RHP*), but are already known to be necessary for fully abstract compilation to low-level code [Juglaret et al. 2016, Kennedy 2006, Patrignani et al. 2015, 2016]. They can also be trivially circumvented if the context has access to side-channels (e.g., it can measure time via a different thread).

2.5 Where is full abstraction?

Full abstraction—the preservation and reflection of observational equivalence—is a well-studied criterion for secure compilation (§2.7). The actually security-relevant direction of full abstraction is *Observational Equivalence Preservation* (*OEP*):

$$OEP : \forall P_1 P_2. P_1 \approx P_2 \Rightarrow P_1 \downarrow \approx P_2 \downarrow$$

One natural question, then, is how *OEP* relates to our criteria of robust preservation.

The answer to this question seems to be nuanced and we haven’t fully resolved it, but we provide a partial answer here in the specific case where programs don’t have internal non-determinism. In various such determinate settings observational equivalence coincides with trace equivalence in all contexts [Cheval et al. 2013, Engelfriet 1985] and, hence, *OEP* coincides with

robust trace-equivalence preservation (*RTEP*, §2.4). Further, we argued in §2.4.2 that, in this setting, *RTEP* is implied by robust 2-relational property preservation (*R2rTP*), so *OEP* is also implied by *R2rTP*. If we additionally assume that divergence is finitely observable, or that the language is terminating, then *RTEP* and *OEP* are both implied by the weaker criterion, robust 2-relational safety preservation (*R2rSP*, §2.4.3).

In the other direction, we have proved that for determinate programs, *RTEP* (and, hence, *OEP*) does *not* imply *any* of the criteria that are above *RSP* or *RDP* in Figure 2.1. We explain here the key idea behind the construction. Fundamentally, *RTEP* (or *OEP*) only requires preserving *equivalence* of behavior. Consequently, a compiler from a language of booleans to itself that *bijectively* renames true to false, false to true, AND to OR, and OR to AND has *RTEP*. On the other hand, this compiler does not have *RSP* or *RDP* since it does not preserve safety or dense properties. For example, the constant function that outputs an infinite stream of trues is mapped to the constant function that outputs an infinite stream of falses. The source function satisfies the safety property “never output false”, while the compiled function does not. Similarly, the source function satisfies the dense property “on any infinite trace, output at least one true”, while the compiled function does not.

Our actual result is a bit stronger: We show that there exists a compiler that has both *RTEP* and compiler correctness—in the sense of *TP*, *SCC*, and *CCC* defined in §2.2.1—but has neither *RSP* nor *RDP*. The proof is similar, but the construction is different, basically exploiting that even with *SCC* and *CCC* a correctly compiled program $P \downarrow$ only needs to be able to properly deal with interactions with target contexts that behave like source contexts, and thus $P \downarrow$ can perform unsafe actions when interacting with target contexts that have no source equivalent.

Theorem 4. There is a compiler between two deterministic languages that satisfies *RTEP*, *TP*, *SCC*, and *CCC*, but none of the criteria above *RSP* and *RDP* in Figure 2.1.

2.6 Proving Secure Compilation

This section demonstrates that the studied criteria can be proved by adapting existing back-translation techniques. We introduce a statically typed source language with first-order functions and input-output and a similar dynamically typed target language (§2.6.1), and we present a simple compiler between the two (§2.6.2). We then describe two very different secure compilation proofs for this compilation chain, both based on techniques originally developed for showing fully abstract compilation. The first proof shows *RrHP* (§2.6.3), the strongest criterion from Figure 2.1, using a context-based back-translation, which provides an “universal embedding” of a target context into a source context [New et al. 2016]. The second proof shows a slightly weaker criterion, Robust Finite-Relational Safety Preservation (§2.6.4), which, however, still includes robust preservation of safety, of noninterference, and in some settings also of trace equivalence, as illustrated by the green area of Figure 2.1. This second proof relies on a trace-based back-translation [Jeffrey and Rathke 2005a, Patrignani et al. 2015, 2016], extended to back-translating a *finite set* of finite execution prefixes. This second technique is more generic, as it only depends on the model for context-program interaction (e.g., calls and

returns), not on all other details of the languages. Since back-translation is often the hardest part of proving a compilation chain secure [Devriese et al. 2016a], we believe that a such a generic back-translation technique that requires less creativity can be very useful, especially when the source and target languages have large abstraction gaps that would make context-based back-translation complicated, if at all possible.

2.6.1 Source and Target Languages

For the sake of simplicity, the languages we consider are simple first-order languages with named procedures where values are boolean and naturals. The source language \mathbf{L}^τ is typed while the target language \mathbf{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), branch and use let-in bindings. Expressions can read naturals from and write naturals to the environment, which generates trace events. Additionally, \mathbf{L}^u has a primitive $\mathbf{e} \text{ has } \tau$ to dynamic check whether expression e has type τ . Contexts \mathbb{C} can call in the program and can manipulate its returned values, but cannot contain read nor write e actions, as those are security-sensitive.

Program $\mathbf{P} ::= \bar{\mathbf{I}}; \bar{\mathbf{F}}$ *Functions* $\mathbf{F} ::= \mathbf{f}(x : \tau) : \tau \mapsto \text{return } e$ *Interfaces* $\mathbf{I} ::= \mathbf{f} : \tau \rightarrow \tau$
Contexts $\mathbb{C} ::= e$ *Types* $\tau ::= \text{Bool} \mid \text{Nat}$
Expressions $e ::= x \mid \text{true} \mid \text{false} \mid n \in \mathbb{N} \mid e \oplus e \mid \text{let } x : \tau = e \text{ in } e \mid \text{if } e \text{ then } e \text{ else } e \mid e \geq e$
 $\mid \text{call } f \ e \mid \text{read} \mid \text{write } e \mid \text{fail}$
Program $\mathbf{P} ::= \bar{\mathbf{I}}; \bar{\mathbf{F}}$ *Functions* $\mathbf{F} ::= \mathbf{f}(x) \mapsto \text{return } e$ *Interfaces* $\mathbf{I} ::= \mathbf{f}$
Contexts $\mathbb{C} ::= e$ *Types* $\tau ::= \text{Bool} \mid \mathbb{N}$
Expressions $e ::= x \mid \text{true} \mid \text{false} \mid n \in \mathbb{N} \mid e \oplus e \mid \text{let } x = e \text{ in } e \mid \text{if } e \text{ then } e \text{ else } e \mid e \geq e$
 $\mid \text{call } f \ e \mid \text{read} \mid \text{write } e \mid \text{fail} \mid e \text{ has } \tau$
Labels $\lambda ::= \epsilon \mid \alpha$ *Actions* $\alpha ::= \text{read } n \mid \text{write } n \mid \downarrow \mid \uparrow \mid \perp$

Program states either describe the evaluation of an expression, given a lookup table for procedure bodies or they describe the reaching of a stuck state: $\cdot ::= P \triangleright e \mid \text{fail}$. Each language has a standard small-step operational semantics ($\Omega \xrightarrow{\lambda} \Omega'$) that describes how a program state evolves, as well as a big step trace semantics ($\Omega \rightsquigarrow \bar{\alpha}$, to use the same notation of §2.1) that concatenates all actions α in a trace $\bar{\alpha}$. The initial state of a program P plugged in context \mathbb{C} , denoted as $\Omega_0(\mathbb{C}[P])$, is the state $P \triangleright e$, where $\mathbb{C} = e$. We now have all the necessary material to define the behavior of a program as the set of traces that it can perform following the semantics rules starting from its initial state.

$$\text{Behav}(\mathbb{C}[P]) = \{\bar{\alpha} \mid \Omega_0(\mathbb{C}[P]) \rightsquigarrow \bar{\alpha}\}$$

2.6.2 The Compiler

The compiler $\cdot\downarrow$ takes L^τ programs and generates L^u ones; technically speaking the compiler operates on programs and then on expressions; we overload the compiler notation for simplicity to refer to all of them. The main feature of the compiler is that it replaces static type annotations with dynamic type checks of function arguments upon invocation of a function (case $\cdot\downarrow\text{-Fun}$).

$$\begin{aligned}
l_1, \dots, l_m; F_1, \dots, F_n \downarrow &= l_1 \downarrow, \dots, l_m \downarrow; F_1 \downarrow, \dots, F_n \downarrow & (\cdot\downarrow\text{-Prog}) \\
f : \tau \rightarrow \tau' \downarrow &= \mathbf{f} & (\cdot\downarrow\text{-Intf}) \\
f(x : \tau) : \tau' \mapsto \text{return } e \downarrow &= \mathbf{f(x) \mapsto \text{return if } x \text{ has } \tau \downarrow \text{ then } e \downarrow \text{ else fail}} & (\cdot\downarrow\text{-Fun}) \\
\text{Nat} \downarrow &= \mathbb{N} & \text{Bool} \downarrow = \mathbf{Bool} \\
n \downarrow &= \mathbf{n} & \text{true} \downarrow = \mathbf{true} & \text{false} \downarrow = \mathbf{false} \\
x \downarrow &= \mathbf{x} & e \oplus e' \downarrow &= e \downarrow \oplus e' \downarrow & e \geq e' \downarrow &= e \downarrow \geq e' \downarrow \\
\text{call } f \ e \downarrow &= \mathbf{\text{call } f \ e} & \text{read} \downarrow &= \mathbf{read} & \text{write } e \downarrow &= \mathbf{write \ e} \\
\text{let } x : \tau = e \text{ in } e' \downarrow &= \mathbf{\text{let } x = e \text{ in } e' \downarrow} & \text{if } e \text{ then } e' \text{ else } e'' \downarrow &= \mathbf{\text{if } e \downarrow \text{ then } e' \downarrow \text{ else } e'' \downarrow}
\end{aligned}$$

2.6.3 Proving Robust Relational Hyperproperty Preservation

To prove that $\cdot\downarrow$ attains *RrHP*, we need a way to back-translate target contexts into source ones, and we use an universal embedding as previously proposed for showing fully abstract compilation [New et al. 2016]. The back-translation needs to generate a source context that respects source-level constraints, in this case it must be well typed. To ensure this, we use Nat as a *Universal Type* in back-translated source contexts. The intuition of the back-translation is that it will encode \mathbf{true} as 0 , \mathbf{false} as 1 and any \mathbf{n} as $n + 2$. Next we need ways to exchange values to and from a regular source type and our universal type. Specifically, we define the following shorthands: $\text{inject}_\tau(e)$ takes an expression e of type τ and returns an expression whose type is the universal type while $\text{extract}_\tau(e)$ takes an expression e of universal type and returns an expression whose type is τ .

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

$\text{inject}_\tau(e)$ will never incur in runtime errors while $\text{extract}_\tau(e)$ can. This mimics the target context ability to write ill-typed code such as $\mathbf{3} + \mathbf{true}$, which we must be able to back-translate and preserve the semantics of (see Example 1)

The back-translation is defined inductively on the structure of target contexts. For clarity we omit the list of function interfaces \bar{l} (i.e., what the back-translated context links against) that is needed for the **call** case.

$$\begin{aligned}
& \mathbf{true} \uparrow = 1 & \mathbf{false} \uparrow = 0 & \mathbf{n} \uparrow = n + 2 & \mathbf{x} \uparrow = x \\
& \mathbf{e} \geq \mathbf{e}' \uparrow = \text{let } x1 : \text{Nat} = \text{extract}_{\text{Nat}}(\mathbf{e} \uparrow) \\
& \quad \text{in let } x2 : \text{Nat} = \text{extract}_{\text{Nat}}(\mathbf{e}' \uparrow) \\
& \quad \text{in inject}_{\text{Bool}}(x1 \geq x2) \\
& \mathbf{e} \oplus \mathbf{e}' \uparrow = \text{let } x1 : \text{Nat} = \text{extract}_{\text{Nat}}(\mathbf{e} \uparrow) \\
& \quad \text{in let } x2 : \text{Nat} = \text{extract}_{\text{Nat}}(\mathbf{e}' \uparrow) \\
& \quad \text{in inject}_{\text{Nat}}(x1 \oplus x2) \\
& \left(\begin{array}{l} \mathbf{let } x = \mathbf{e} \\ \mathbf{in } \mathbf{e}' \end{array} \right) \uparrow = \begin{array}{l} \text{let } x : \text{Nat} = \mathbf{e} \uparrow \\ \text{in } \mathbf{e}' \uparrow \end{array} & \left(\begin{array}{l} \mathbf{if } \mathbf{e} \text{ then } \\ \mathbf{e}' \text{ else } \mathbf{e}'' \end{array} \right) \uparrow = \begin{array}{l} \text{if } \text{extract}_{\text{Bool}}(\mathbf{e} \uparrow) \text{ then} \\ \mathbf{e}' \uparrow \text{ else } \mathbf{e}'' \uparrow \end{array} \\
& \mathbf{e} \text{ has } \tau \uparrow = \begin{cases} \text{let } x : \text{Nat} = \mathbf{e} \uparrow \text{ in if } x \geq 2 \text{ then } 0 \text{ else } 1 & \text{if } \tau \equiv \text{Bool} \\ \text{let } x : \text{Nat} = \mathbf{e} \uparrow \text{ in if } x \geq 2 \text{ then } 1 \text{ else } 0 & \text{if } \tau \equiv \mathbb{N} \end{cases} \\
& \mathbf{call } f \mathbf{ e} \uparrow = \text{inject}_{\tau'}(\text{call } f \text{ extract}_{\tau}(\mathbf{e} \uparrow)) & \text{if } f : \tau \rightarrow \tau' \in \bar{l} & \mathbf{fail} \uparrow = \mathbf{fail}
\end{aligned}$$

Example 1 (Back-Translation). We show the back-translation of two simple target contexts and intuitively explain why the back-translation is correct and why it needs **inject** and **extract**.

Consider context $\mathbb{C}_1 = \mathbf{3} * \mathbf{5}$ that reduces to **15** irrespective of the program it links against. The back-translation must intuitively ensure that $\mathbb{C}_1 \uparrow$ reduces to **17**, which is the back-translation of **15**. If we unfold the definition of $\mathbb{C}_1 \uparrow$ we have the following (given that $\mathbf{3} \uparrow = 5$ and $\mathbf{5} \uparrow = 7$).

$$\text{let } x1 : \text{Nat} = \text{extract}_{\text{Nat}}(5) \text{ in let } x2 : \text{Nat} = \text{extract}_{\text{Nat}}(7) \text{ in inject}_{\text{Nat}}(x1 * x2)$$

By examining the code of **extract_{Nat}** we see that in both cases it will just perform a subtraction by 2 turning the 5 and 7 respectively into 3 and 5. So after few reduction steps we have the following term: **inject_{Nat}(3 * 5)**. This again seems correct: the multiplication returns **15** and the inject returns **17**, which would be the result of $\langle\langle \mathbf{15} \rangle\rangle$.

Let us now consider a different context $\mathbb{C}_2 = \mathbf{false} + \mathbf{3}$. We know that no matter what program links against it, it will reduce to **fail**. Its statically typed back-translation is:

$$\text{let } x1 : \text{Nat} = \text{extract}_{\text{Nat}}(0) \text{ in let } x2 : \text{Nat} = \text{extract}_{\text{Nat}}(7) \text{ in inject}_{\text{Nat}}(x1 * x2)$$

By looking at its code we can see that the execution of **extract_{Nat}(0)** will indeed result in a **fail**, which is what we want and expect, as that is the back-translation of **fail**. \square

We proved *RrHP* for this simple compilation chain, using a simple logical relation that includes cases both for terms of source type (intuitively used for compiler correctness) as well as for terms of back-translation type [Devriese et al. 2016a, New et al. 2016]. We prove the usual compatibility lemmas at source type for the compiler case while each back-translation case is proved correct at back-translation type, as illustrated by Example 1. The appendix in the supplementary materials provides full details.

2.6.4 Proving Robust Finite-Relational Safety Preservation

Proving that this simple compilation chain attains *RFrSC* does not require back-translating a target context, as we only need to build a source context that can reproduce a finite set of finite trace prefixes, but that is not necessarily equivalent to the original target context. We describe this back-translation on an example. The interested reader can find all details in the appendix.

Example 2 (Back-Translation of traces). Consider the following programs (the interfaces are omitted for concision):

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

The compiled programs are analogous, except they include dynamic type checks of the arguments:

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

Now, consider the target context

$$C = \text{let } x1 = \text{call } f \ 5 \text{ in if } x1 \geq 5 \text{ then call } g \ (x1) \text{ else call } g \ (\text{false})$$

The programs plugged into the context can generate (at least) the following traces, where \Downarrow means termination and \perp means failure:

$$C[P_1 \downarrow] \rightsquigarrow \Downarrow \qquad C[P_2 \downarrow] \rightsquigarrow \text{read } 5; \Downarrow \quad C[P_2 \downarrow] \rightsquigarrow \text{read } 0; \perp$$

In the first execution of $C[P_2 \downarrow]$, the programs reads 5, and the first branch of the if-then-else of the context is entered. In the second execution of $C[P_2 \downarrow]$, the programs reads 0, the second branch of the context is entered 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 have any information on the control flow of the program, and in particular on which function produces which input or output. Therefore, we use the execution prefixes to we enrich the traces with information about the calls and returns between program and context. To do so, we modify the semantics to record the call stack. Now, there are two rules for handling calls and returns: one modelling control flow going from context to program (and this is decorated with a $?$) and one modelling the opposite: control flow going from program to context (and this is decorated with a $!$). Each rule generates the appropriate event, $\text{call } f \ v?$ or $\text{ret } v!$ respectively. If the

call or the return occurs within the program no event is generated, as such calls and returns are not recorded, as they are not relevant for back-translating a context.

Since the semantics are otherwise identical, obtaining the new informative traces is straightforward: we can just replay the execution by substituting the rules for calls and returns.

$$\text{Labels } \lambda ::= \dots \mid \beta \qquad \text{Interactions } \beta ::= \text{call } f \ v? \mid \text{ret } v!$$

Now, the traces generated by the compiled programs plugged into the context become:

$$\begin{aligned} \mathbb{C}[P_1 \downarrow] &\leadsto \text{call } f \ 5?; \quad \text{ret } 5!; \text{call } g \ 5?; \text{ret } \text{true}!; \downarrow \\ \mathbb{C}[P_2 \downarrow] &\leadsto \text{call } f \ 5?; \text{read } 5; \text{ret } 5!; \text{call } g \ 5?; \text{ret } \text{true}!; \downarrow \\ \mathbb{C}[P_2 \downarrow] &\leadsto \text{call } f \ 5?; \text{read } 0; \text{ret } 0!; \text{call } g \ \text{false}?; \perp \end{aligned}$$

In our example language, read and writes are only performed by the programs. The context specifies the flow of the program. Therefore, the role of the back-translated source context will be to perform appropriate calls; the I/O events will be obtained by correctness of the compiler.

Since the context is the same in all executions, the only source of non-determinism in the execution is the program. Therefore, two traces generated by the same context (but not necessarily the same program), where I/O events have been removed, must be equal up to the point where there are two different return events: these traces are organized as a tree (Figure 2.2, on the left). This tree can be back-translated to a source context using nested if-then-else as depicted below (Figure 2.2, on the right, dotted lines indicate what the back-translation generates for each action in the tree). When additional branches are missing (e.g., there is no third behavior that analyzes the first return or no second behavior that analyses the second return on the left execution), the back-translation inserts `fail` in the code – they are dead code branches (marked with a `**`).

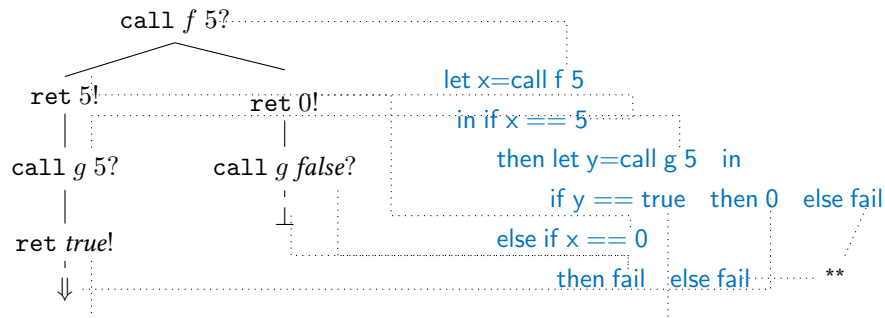


Figure 2.2: Example of a back-translation of traces.

Correctness of the back-translation shows that this source context will produce exactly the same non-informative traces as before, therefore yielding *RFrSP*. However, would not be true of informative traces (that track calls and returns). In fact the call to `g` with a boolean would

be ill-typed, and the back-translation has to solve this issue by shifting the failure from the program to the context, so the picture links the call `g false?` action to a `fail`. The call will never be executed at the source level. \square

Using the technique illustrated on the example above we have proved *RFrSP* for the compilation chain of this section. Complete details are in the appendix.

2.7 Related Work

Full Abstraction, originally applied to secure compilation in the seminal work of Abadi [1999], has since received a lot of attention [Patrignani et al. 2019]. Abadi [1999] and, later, Kennedy [2006] identified failures of full abstraction in the Java to JVM and C# to CIL compilers, some of which were fixed, but also others for which fixing was deemed too costly compared to the perceived practical security gain. Abadi et al. [2002] proved full abstraction of secure channel implementations using cryptography, but to prevent network traffic attacks they had to introduce noise in their translation, which in practice would consume network bandwidth. Ahmed et al. [Ahmed 2015, Ahmed and Blume 2008, 2011, New et al. 2016] proved the full abstraction of type-preserving compiler passes for simple functional languages. Abadi and Plotkin [2012] and Jagadeesan et al. [2011] expressed the protection provided by address space layout randomization as a probabilistic variant of full abstraction. Fournet et al. [2013] devised a fully abstract compiler from a subset of ML to JavaScript. Patrignani et al. [Larmuseau et al. 2015, Patrignani et al. 2015, 2016] studied fully abstract compilation to machine code, starting from single modules written in simple, idealized object-oriented and functional languages and targeting a hardware isolation mechanism similar to Intel’s SGX [Intel].

Until recently, most formal secure compilation work was focused only on fully abstract compilation. The goal of our work is to explore a diverse set of secure compilation criteria, a few of them formally stronger than (the interesting direction of) full abstraction at least in various determinate settings, but most of them not directly comparable to full abstraction, some of them easier to achieve and prove than full abstraction, while others potentially harder to achieve and prove. This exploration clarifies the trade-off between security guarantees and efficient enforcement for secure compilation: On one extreme, *RTP* robustly preserves only trace properties, but does not require enforcing confidentiality; on the other extreme, robustly preserving relational properties gives very strong guarantees, but requires enforcing that both the private data and the code of a program remain hidden from the context, which is often harder to achieve. The best criterion to apply depends on the application domain, but we provide a framework in which interesting design questions such as the following two can be addressed: (1) *What secure compilation criterion, when violated, would the developers of the Java to JVM and C# to CIL compilers, at least in principle, be willing to fix?* The work of Kennedy [2006] indicates that fully abstract compilation is not such a good answer to this question, and we wonder whether *RTP* or *RHP* could be better answers. (2) *What weaker secure compilation criterion would the translations of Abadi et al. [2002] satisfy if they did not introduce (inefficient) noise to prevent network traffic analysis?* Abadi et al. [2002] explicitly leave this problem open

in their paper, and we believe one answer could be *RTP*, since it does not require preserving any confidentiality.

Our exploration also forced us to challenge the assumptions and design decisions of prior work. This is most visible in our attempt to use as generic and realistic a trace model as possible. To start, this meant moving away from the standard assumption in the hyperproperties literature [Clarkson and Schneider 2010] that all traces are infinite, and switching instead to a trace model inspired by CompCert’s [Leroy 2009a] with both terminating and non-terminating traces, and where non-terminating traces can be finite but not finitely observable (to account for silent divergence). This more realistic model required us to find a class of trace properties to replace liveness. At places, this model is also at odds with the external observation notions typically used for fully abstract compilation, specifically that divergence is an observable event. To accommodate such cases, extra assumptions are needed, such as the one we used to show that *RTEP* follows from *R2rSP* in some settings (§2.4.3).

Proof techniques The context-based back-translation we use to prove *RrHP* in §2.6.3 is adapted from the universal embedding technique of New et al. [2016], who propose it for proving full abstraction of translations from typed to untyped languages. Devriese et al. [2016a, 2017] show that even when a precise universal type does not exist in the source, one can use an approximate embedding that only works for a certain number of execution steps. They illustrate such an approximate back-translation by proving full abstraction for a compiler from the simply typed to the untyped λ -calculus. The trace-based back-translation technique we use in §2.6.4 was first proposed by Jeffrey and Rathke [2005a,b] for proving the full abstraction of so called “trace semantics” (which are often used to prove observational equivalences). This back-translation technique was then adapted to show full abstraction of compilation chains to low-level target languages [Agten et al. 2015, Patrignani and Clarke 2015, Patrignani et al. 2016]. While many other proof techniques have been previously investigated [Abadi and Plotkin 2012, Abadi et al. 2002, Ahmed and Blume 2008, 2011, Fournet et al. 2013, Jagadeesan et al. 2011], proofs of full abstraction remain notoriously difficult, even for very simple languages, with apparently simple conjectures surviving for decades before being finally settled [Devriese et al. 2018].

It will be interesting to investigate how many of the existing full abstraction proofs can be repurposed to show stronger criteria from Figure 2.1, like we did in §2.6.3 for the universal embedding technique of New et al. [2016]. For instance, it will be interesting to determine the strongest criterion from Figure 2.1 for which the approximate back-translation of Devriese et al. [2016a, 2017] can be used.

Development of *RSP* Two pieces of concurrent work have examined more carefully how to attain and prove one of the weakest of our preservation criteria, *RSP* (§2.2.3). Patrignani and Garg [2018] show *RSP* for compilers from simple sequential and concurrent languages to capability machines [Watson et al. 2015b]. They observe that that if the source language has a verification system for robust safety and compilation is limited to verified programs, then

RSP can be established without directly resorting to back-translation. Independently, in chapter 3 we aim at devising realistic secure compilation chains for protecting mutually distrustful components written in an unsafe language like C. We show that by moving away from the full abstraction variant used in earlier work [Juglaret et al. 2016] to a variant of the *RSP* criterion from §2.2.3, we can support a more realistic model of dynamic compromise of components, while at the same time obtaining a criterion that is easier to achieve and prove.

Hypersafety Preservation The high-level idea of specifying secure compilation as the preservation of properties and hyperproperties in adversarial contexts goes back to the work of Patrignani and Garg [2017]. However, that work’s technical development is limited to one criterion—the preservation of finite prefixes of program traces by compilation. Superficially, this is similar to one of our criteria, *RHSP*, but there are several differences even from *RHSP*. First, Patrignani and Garg [2017] do not consider adversarial contexts explicitly. This suffices for their setting of closed reactive programs, where traces are inherently fully abstract (so considering the adversarial context is irrelevant), but not in general. Second, they are interested in designing a criterion that accommodates specific fail-safe like mechanisms for low-level enforcement, so the preservation of hypersafety properties is not perfect, and one has to show, for every relevant property, that the criterion is meaningful. However, Patrignani and Garg [2017] consider translations of trace symbols induced by compilation, something that our criteria could also be extended with.

Source-level reasoning about robust satisfaction While this chapter studies secure compilation criteria based on *preserving* the robust satisfaction for various classes of properties, formally verifying that a partial source program robustly satisfies a specification is a challenging problem. So far, most of the research has focused on techniques for proving observational equivalence [Abadi et al. 2018, Cheval et al. 2018, Delaune and Hirschi 2017, Jeffrey and Rathke 2005a,b] or trace equivalence [Baelde et al. 2017, Cheval et al. 2013]. For robust satisfaction of trace properties, Kupferman and Vardi [1999] study robust model checking of systems modeled by nondeterministic Moore machines and properties specified by branching temporal logic. Robust safety, the robust satisfaction of safety properties, was studied for the analysis of security protocols [Backes et al. 2008, 2011, Gordon and Jeffrey 2004], and more recently for compositional verification [Swasey et al. 2017]. Verifying the robust satisfaction of hyperproperties and relational hyperproperties beyond observational equivalence and trace equivalence seems to be an open research problem.

2.8 Conclusion and Future Work

This chapter proposes a possible foundation for secure compilation by exploring many different criteria based on robust property preservation (Figure 2.1), but the road to building practical compilation chains achieving one of these criteria is still long and challenging. Even for *RSP*,

scaling up to realistic programming languages and efficiently enforcing protection of the compiled program without restrictions on the linked context is challenging [Abate et al. 2018a, Patrignani and Garg 2018]. For *R2HSP* the problem becomes harder because one needs to protect the secrecy of the program’s data, which is especially challenging in a realistic attacker model with side-channels, in which a *RTINIP*-like property seems the best one can hope for in practice. Finally, as soon as one is outside the green area of Figure 2.1 the generic back-translation technique of §2.6.4 stops applying and one needs to get creative about the proofs.

We made the simplifying assumption that the source and the target languages have the same trace model, and while this assumption is currently true for CompCert [Leroy 2009a], it is a big restriction in general. Fortunately, the criteria of this chapter can be easily extended to take a relation between source and target traces as an extra component of the compilation chain. It is also easy to automatically lift this relation on traces to a relation on sets of traces, sets of sets of traces, etc. What is less obvious is whether this automatic lifting is what one always wants, and more importantly whether the users of a secure compilation chain will understand this relation between the properties they reason about at the source languages and the ones they get at the target level.

Finally, the relation between the criteria of Figure 2.1 and fully abstract compilation requires further investigation. We identify the sufficient conditions under which trace-equivalence preservation follows from certain of these criteria, which gives us a one-way relation to observational equivalence preservation in the cases in which observational equivalence coincides with trace equivalence [Cheval et al. 2013, Engelfriet 1985]. Even under this assumption, what is less clear is whether there are any sufficient conditions for fully abstract compilation to imply any of the criteria of Figure 2.1. The separation result of §2.5 shows that compiler correctness, even when reasonably compositional (i.e., satisfying *SCC* and *CCC*), is not enough. Yet the fact that fully abstract compilers often provide both the necessary enforcement mechanisms and the proof techniques to achieve even the highest criterion in Figure 2.1 (as illustrated in §2.6.3) suggests that there is more to full abstraction than currently meets the eye. One lead is to look at source program encodings targeted at illustrating confidentiality and internally observable safety properties in terms of observational equivalence [Abadi 1999, Patrignani et al. 2019].

3 When Good Components Go Bad: Secure Compilation for Unsafe Languages

3.1 Overview

Compartmentalization offers a strong, practical defense against a range of devastating low-level attacks, such as control-flow hijacks exploiting buffer overflows and other vulnerabilities in C, C++, and other unsafe languages [Bittau et al. 2008, Gudka et al. 2015, Watson et al. 2015b]. Widely deployed compartmentalization technologies include process-level privilege separation [Bittau et al. 2008, Gudka et al. 2015, Kilpatrick 2003] (used in OpenSSH [Provos et al. 2003] and for sandboxing plugins and tabs in web browsers [Reis and Gribble 2009]), software fault isolation [Tan 2017, Wahbe et al. 1993] (e.g., Google Native Client [Yee et al. 2010]), WebAssembly modules [Haas et al. 2017] in modern web browsers, and hardware enclaves (e.g., SGX [Intel]); many more are on the drawing boards [Azevedo de Amorim et al. 2015, Chisnall et al. 2015, Skorstengaard et al. 2018a, Watson et al. 2015b]. These mechanisms offer an attractive base for building more secure compilation chains that mitigate low-level attacks [Gollamudi and Fournet 2018, Gudka et al. 2015, Juglaret et al. 2015, Patrignani et al. 2016, Tsampas et al. 2017, van Ginkel et al. 2016, Van Strydonck et al. 2018]. In particular, compartmentalization can be applied in unsafe low-level languages to structure large, performance-critical applications into mutually distrustful components that have clearly specified privileges and interact via well-defined interfaces.

Intuitively, protecting each component from all the others should bring strong security benefits, since a vulnerability in one component need not compromise the security of the whole application. Each component will be protected from all other components for as long as it remains “good.” If, at some point, it encounters an internal vulnerability such as a buffer overflow, then, from this point on, it is assumed to be compromised and under the control of the attacker, potentially causing it to attack the remaining uncompromised components. The main goal of this chapter is to formalize this dynamic-compromise intuition and precisely characterize what it means for a compilation chain to be secure in this setting.

We want a characterization that supports *source-level security reasoning*, allowing programmers to reason about the security properties of their code without knowing anything about the complex internals of the compilation chain (compiler, linker, loader, runtime system, system software, etc). What makes this particularly challenging for C and C++ programs is that they may encounter *undefined behaviors*—situations that have no source-level meaning whatsoever. Compilers are allowed to assume that undefined behaviors never occur in programs, and they

aggressively exploit this assumption to produce the fastest possible code for well-defined programs, in particular by avoiding the insertion of run-time checks. For example, memory safety violations [Azevedo de Amorim et al. 2018, Szekeres et al. 2013] (e.g., accessing an array out of bounds, or using a pointer after its memory region has been freed) and type safety violations [Duck and Yap 2018, Haller et al. 2016] (e.g., invalid unchecked casts)—cause real C compilers to produce code that behaves arbitrarily, often leading to exploitable vulnerabilities [Heartbleed, Szekeres et al. 2013].

Of course, not every undefined behavior is necessarily exploitable. However, for the sake of strong security guarantees, we make a worst-case assumption that *any* undefined behavior encountered within a component can lead to its compromise. Indeed, in the remainder we equate the notions of “encountering undefined behavior” and “becoming compromised.”

While the dangers of memory safety and casting violations are widely understood, the C and C++ standards [ISO/IEC 2011] call out large numbers of undefined behaviors [Hathhorn et al. 2015, Krebbers 2015] that are less familiar, even to experienced C/C++ developers [Lattner 2011, Wang et al. 2013]. To minimize programmer confusion and lower the risk of introducing security vulnerabilities, real compilers generally give sane and predictable semantics to some of these behaviors. For example, signed integer overflow is officially an undefined behavior in standard C, but many compilers (at least with certain flags set) guarantee that the result will be calculated using wraparound arithmetic. Thus, for purposes of defining secure compilation, the set of undefined behaviors is effectively defined by the compiler at hand rather than by the standard.

The purpose of a compartmentalizing compilation chain is to ensure that the arbitrary, potentially malicious, effects of undefined behavior are limited to the component in which it occurs. For a start, it should restrict the *spatial* scope of a compromise to the component that encounters undefined behavior. Such compromised components can only influence other components via controlled interactions respecting their interfaces and the other abstractions of the source language (e.g., the stack discipline on calls and returns). Moreover, to model dynamic compromise and give each component full guarantees as long as it has not yet encountered undefined behavior, the *temporal* scope of compromise must also be restricted. In particular, compiler optimizations should never cause the effects of undefined behavior to show up before earlier “observable events” such as system calls. Unlike the spatial restriction, which requires some form of run-time enforcement in software or hardware, the temporal restriction can be enforced just by foregoing certain aggressive optimizations. For example, the temporal restriction (but not the spatial one) is already enforced by the CompCert C compiler [Leroy 2009a, Regehr 2010], providing a significantly cleaner model of undefined behavior than other C compilers [Regehr 2010].

We want a characterization that is *formal*—that brings mathematical precision to the security guarantees and attacker model of compartmentalizing compilation. This can serve both as a clear specification for verified secure compilation chains and as useful guidance for unverified ones. Moreover, we want the characterization to provide source-level reasoning principles that can be used to assess the security of compartmentalized applications. To make this feasible in practice, the *amount* of source code to be verified or audited has to be relatively small. So, while

we can require developers to carefully analyze the privileges of each component and the correctness of some very small pieces of security-critical code, we cannot expect them to establish the full correctness—or even absence of undefined behavior—for most of their components.

Our secure compilation criterion improves on the state of the art in three important respects. First, our criterion applies to *compartmentalized* programs, while most existing formal criteria for secure compilation are phrased in terms of protecting a single trusted program from an untrusted context [Abadi 1999, Abadi and Planul 2013, Abadi and Plotkin 2012, Abate et al. 2018b, Agten et al. 2012, 2015, Fournet et al. 2013, Larmuseau et al. 2015, Patrignani et al. 2015]. Second, unlike some recent criteria that do consider modular protection [Devriese et al. 2017, Patrignani et al. 2016], our criterion applies to *unsafe* source languages with undefined behaviors. And third, it considers a *dynamic* compromise model—a critical advance over the recent proposal of Juglaret et al. [2016], which does consider components written in unsafe languages, but which is limited to a static compromise model. This is a serious limitation: components whose code contains any vulnerability that might potentially manifest itself as undefined behavior are given no guarantees whatsoever, irrespective of whether an attacker actually exploits these vulnerabilities. Moreover, vulnerable components lose all guarantees from the start of the execution—possibly long before any actual compromise. Experience shows that large enough C or C++ codebases essentially always contain vulnerabilities [Szekeres et al. 2013]. Thus, although static compromise models may be appropriate for safe languages, they are not useful for unsafe low-level languages.

As we will see in §3.5, the limitation to static compromise scenarios seems inescapable for previous techniques, which are all based on the formal criterion of *full abstraction* [Abadi 1999]. To support dynamic compromise scenarios, we take an unconventional approach, dropping full abstraction and instead phrasing our criterion in terms of preserving safety properties [Lamport and Schneider 1984] in adversarial contexts (chapter 2), where, formally, safety properties are predicates over execution traces that are informative enough to detect the compromise of components and to allow the execution to be “rewound” along the same trace. Moving away from full abstraction also makes our criterion easier to achieve efficiently in practice and to prove at scale. Finally, we expect our criterion to scale naturally from properties to hyperproperties such as confidentiality (see §2.3.3, §3.5, and §3.6).

Contributions Our first contribution is *Robustly Safe Compartmentalizing Compilation (RSCC)*, a new secure compilation criterion articulating strong end-to-end security guarantees for components written in unsafe languages with undefined behavior. This criterion is the first to support *dynamic compromise* in a system of *mutually distrustful components* with clearly specified privileges. We start by illustrating the intuition, informal attacker model, and source-level reasoning behind RSCC using a simple example application (§3.2).

Our second contribution is a formal presentation of RSCC. We start from *Robustly Safe Compilation (RSC)*, the simple security criterion from §2.2.3, and extend this first to dynamic compromise (RSC^{DC} , §3.3.1), then mutually distrustful components (RSC_{MD}^{DC} , §3.3.2), and finally to the full definition of RSCC (§3.3.3). We also give an effective and generic proof technique for RSCC

(§3.3.4). We start with a target-level execution and explain any finite sequence of calls and returns in terms of the source language by constructing a whole source program that produces this prefix. We then use standard simulation proofs to relate our semantics for whole programs to semantics that capture the behavior of a partial program in an arbitrary context. This proof architecture yields simpler and more scalable proofs than previous work in this space [Juglaret et al. 2016]. One particularly important advantage is that it allows us to reuse a whole-program compiler correctness result à la CompCert [Leroy 2009a] as a black box, avoiding the need to prove any other simulations between the source and target languages.

Our third contribution is a proof-of-concept secure compilation chain (§3.4) for a simple unsafe sequential language featuring buffers, procedures, components, and a CompCert-like block-based memory model [Leroy and Blazy 2008] (§3.4.1). Our entire compilation chain is implemented in the Coq proof assistant. The first step compiles our source language to a simple low-level abstract machine with built-in compartmentalization (§3.4.2). We use the proof technique from §3.3.4 to construct careful proofs—many of them machine-checked in Coq—showing that this compiler satisfies *RSCC* (§3.4.3). Finally, we describe two back ends for our compiler, showing that the protection guarantees of the compartmentalized abstract machine can be achieved at the lowest level using either software fault isolation (SFI, §3.4.4) or a tag-based reference monitor (§3.4.6). The tag-based back end, in particular, is novel, using linear return capabilities to enforce a cross-component call/return discipline. Neither back end has yet been formally verified, but we have used property-based testing to gain confidence that the SFI back end satisfies RSC_{MD}^{DC} .

These contributions lay a solid foundation for future secure compilation chains that could bring sound and practical compartmentalization to C, C++, and other unsafe low-level languages. We address three fundamental questions: (1) *What is the desired secure compilation criterion and to what attacker model and source-level security reasoning principles does it correspond?* Answer: We propose the *RSCC* criterion from §3.2–§3.3. (2) *How can we effectively enforce secure compilation?* Answer: Various mechanisms are possible; the simple compilation chain from §3.4 illustrates how either software fault isolation or tagged-based reference monitoring can enforce *RSCC*. (3) *How can we achieve high assurance that the resulting compilation chain is indeed secure?* Answer: We show that formal verification (§3.4.3) and property-based testing (§3.4.4) can be successfully used together for this in a proof assistant like Coq.

We close with related (§3.5) and future (§3.6) work. Our Coq development is available at <https://github.com/secure-compilation/when-good-components-go-bad/>

3.2 *RSCC* By Example

We begin by an overview of compartmentalizing compilation chains, our attacker model, and how viewing this model as a dynamic compromise game leads to intuitive principles for security analysis.

We need not be very precise, here, about the details of the source language; we just assume that it is equipped with some compartmentalization facility [Gudka et al. 2015, Vasilakis et al. 2018] that allows programmers to break up security-critical applications into mutually distrustful *components* that have clearly specified *privileges* and can only interact via well-defined *interfaces*. In fact we assume that the interface of each component gives a precise description of its privilege. The notions of component and interface that we use for defining the secure compilation criteria in §3.3 are quite generic: interfaces can include any requirements that can be enforced on components, including type signatures, lists of allowed system calls, or more detailed access-control specifications describing legal parameters to cross-component calls (e.g., ACLs for operations on files). We assume that the division of an application into components and the interfaces of those components are statically determined and fixed. For the illustrative language of §3.4, we will use a simple setup in which components don’t directly share state, interfaces just list the procedures that each component provides and those that it expects to be present in its context, and the only thing one component can do to another one is to call procedures allowed by their interfaces.

The goal of a compartmentalizing compilation chain is to ensure that components interact according to their interfaces even in the presence of undefined behavior. Our secure compilation criterion does not fix a specific mechanism for achieving this: responsibility can be divided among the different parts of the compilation chain, such as the compiler, linker, loader, runtime system, system software, and hardware. In §3.4 we study a compilation chain with two alternative back ends—one using software fault isolation and one using tag-based reference monitoring for compartmentalization. What a compromised component *can* still do in this model is to use its access to other components, as allowed by its interface, to either trick them into misusing their own privileges (i.e., confused deputy attacks) or even compromise them as well (e.g., by sending them malformed inputs that trigger control-hijacking attacks exploiting undefined behaviors in their code).

We model input and output as interaction with a designated *environment* component *E* that is given an interface but no implementation. When invoked, environment functions are assumed to immediately return a non-deterministically chosen value [Leroy 2009a]. In terms of security, the environment is thus the initial source of arbitrary, possibly malformed, inputs that can exploit buffer overflows and other vulnerabilities to compromise other components.

As we argued in the introduction, it is often unrealistic to assume that we know in advance which components will be compromised and which ones will not. This motivates our model of *dynamic compromise*, in which each component receives secure compilation guarantees until it becomes compromised by encountering an undefined behavior, causing it to start attacking the remaining uncompromised components. In contrast to earlier static-compromise models [Juglaret et al. 2016], a component only loses guarantees in our model after an attacker discovers and manages to exploit a vulnerability, by sending it inputs that lead to an undefined behavior. The mere existence of vulnerabilities—undefined behaviors that can be reached after some sequence of inputs—is not enough for the component to be considered compromised.

This model allows developers to reason informally about various compromise scenarios and their impact on the security of the whole application [Gudka et al. 2015]. If the consequences

```

component C0 {
  export valid;
  valid(data) { ... }
}
component C1 {
  import E.read, C2.init, C2.process;
  main() {
    C2.init();
    x := E.read();
    y := C1.parse(x);    // (V1) can yield Undef for some x
    C2.process(x,y);
  }
  parse(x) { ... }
}
component C2 {
  import E.write, C0.valid;
  export init, process;
  init() { ... }
  process(x,y) {
    C2.prepare();        // (V2) can yield Undef if not initialized
    data := C2.handle(y); // (V3) can yield Undef for some y
    if C0.valid(data) then E.write(<data,x>)
  }
  prepare() { ... }
  handle(y) { ... }
}

```

Figure 3.1: Pseudocode of compartmentalized application

of some plausible compromise seem too serious, developers can further reduce or separate privilege by narrowing interfaces or splitting components, or they can make components more defensive by validating their inputs.

As a first running example, consider the idealized application in Figure 3.1. It defines three components (C_0 , C_1 , and C_2) that interact with the environment E via input ($E.read$) and output ($E.write$) operations. Component C_1 defines a $main()$ procedure, which first invokes $C_2.init()$ and then reads a request x from the environment (e.g., coming from some remote client), parses it by calling an internal procedure to obtain y , and then invokes $C_2.process(x,y)$. This, in turn, calls $C_2.prepare()$ and $C_2.handle(y)$, obtaining some data that it validates using $C_0.valid$ and, if this succeeds, writes data together with the original request x to the environment.

Suppose we would like to establish two properties:

- (S_1) any call $E.write(<data,x>)$ happens as a response to a previous $E.read()$ call by C_1 obtaining the request x ; and
- (S_2) the application only writes valid data (i.e., data for which $C_0.valid$ returns true).

These can be shown to hold of executions that do not encounter undefined behavior simply by analyzing the control flow. But what if undefined behavior does occur? Suppose that we can rule out this possibility—by auditing, testing, or formal verification—for some parts of the code, but we are unsure about three subroutines:

- (V₁) C₁.parse(x) performs complex array computations, and we do not know if it is immune to buffer overflows for all x;
- (V₂) C₂.prepare() is intended to be called only if C₂.init() has been called beforehand to set up a shared data structure; otherwise, it might dereference an undefined pointer;
- (V₃) C₂.handle(y) might cause integer overflow on some inputs.

If the attacker finds an input that causes the undefined behavior in V₁ to occur, then C₁ can get compromised and call C₂.process(x,y) with values of x that it hasn't received from the environment, thus invalidating S₁. Nevertheless, if no other undefined behavior is encountered during the execution, this attack cannot have any effect on the code run by C₂, so S₂ remains true.

Now consider the possible undefined behavior from V₂. If C₁ is not compromised, this undefined behavior cannot occur, since C₂.init() will be called before C₂.prepare(). Moreover, this undefined behavior cannot occur even if C₁ is compromised by the undefined behavior in V₁, because that can only occur *after* C₂.init() has been called. Hence V₁ and V₂ together are no worse than V₁ alone, and property S₂ remains true. Inferring this crucially depends on our model of dynamic compromise, in which C₁ can be treated as honest and gets guarantees until it encounters undefined behavior. If instead we were only allowed to reason about C₁'s ability to do damage based on its *interface*, as would happen in a model of static compromise [Juglaret et al. 2016], we wouldn't be able to conclude that C₂ cannot be compromised: an arbitrary component with the same interface as C₁ could indeed compromise C₂ by calling C₂.process before C₂.init. Finally, if execution encounters undefined behavior in V₃, then C₂ can get compromised irrespective of whether C₁ is compromised beforehand, invalidating both S₁ and S₂.

Though we have not yet made it formal, this security analysis already identifies C₂ as a single point of failure for both desired properties of our system. This suggests several ways the program could be improved: The code in C₂.handle could be hardened to reduce its chances of encountering undefined behavior, e.g. by doing better input validation. Or C₁ could validate the values it sends to C₂.process, so that an attacker would have to compromise both C₁ and C₂ to break the validity of writes. To ensure the correspondence of reads and writes despite the compromise of C₁, we could make C₂ read the request values directly from E, instead of via C₁.

To achieve the best security though, we can refactor so that the read and write privileges are isolated in C₀, which performs no complex data processing and thus is a lot less likely to be compromised by undefined behavior (Figure 3.2). In this variant, C₀ reads a request, calls C₁.parse on this request, passes the result to C₂.process, validates the data C₂ returns and then writes it out. This way both our desired properties hold even if both C₁ and C₂ are compromised, since now the core application logic and privileges have been completely separated from the dangerous data processing operations that could cause vulnerabilities.

Let's begin making all this a bit more formal. The first step is to make the security goals of our example application more precise. We do this in terms of execution *traces* that are built

```

component C0 {
  import E.read, E.write, C2.init, C1.parse, C2.process;
  main() {
    C2.init();
    x := E.read();
    y := C1.parse(x);
    data := C2.process(y);
    if C0.valid(data) then E.write(<data,x>)
  }
  valid(data) { ... }
}
component C1 {
  export parse;
  parse(x) { ... }           //(V1) can yield Undef for some x
}
component C2 {
  export init, process;
  init() { ... }
  process(y) {
    C2.prepare();           //(V2) can yield Undef if not initialized
    return C2.handle(y);   //(V3) can yield Undef for some y
  }
  prepare() { ... }
  handle(y) { ... }
}

```

Figure 3.2: More secure refactoring of the application

from *events* such as cross-component calls and returns. The two intuitive properties from our example can be phrased in terms of traces as follows: If `E.write(<data,x>)` appears in an execution trace, then

- (S_1) `E.read` was called previously and returned `x`, and
- (S_2) `C0.valid(data)` was called previously and returned `true`.

The refactored application in Figure 3.2 achieves both properties despite the compromise of both C_1 via V_1 and C_2 via V_3 , but, for the first variant in Figure 3.1 the properties need to be weakened as follows: If `E.write(<data,x>)` appears in an execution trace then

- (W_1) `E.read` previously returned `x` or `E.read` previously returned an `x'` that can cause undefined behavior in `C1.parse(x')` or `C2.process(x,y)` was called previously with a `y` that can cause undefined behavior in `C2.handle(y)`, and
- (W_2) `C0.valid(data)` was called previously and returned `true` or `C2.process(x,y)` was called previously with a `y` that can cause undefined behavior in `C2.handle(y)`.

While these properties are significantly weaker (and harder to understand), they are still not trivial; in particular, they still tell us something useful under the assumption that the attacker has not actually discovered how to compromise C_1 or C_2 .

Properties S_1 , S_2 , W_1 , W_2 are all *safety properties* [Lamport and Schneider 1984]—inspired, in this case, by the sorts of “correspondence assertions” used to specify authenticity in security protocols [Gordon and Jeffrey 2003, 2004, Woo and Lam 1993]. A trace property is a safety

property if, within any (possibly infinite) trace that violates the property, there exists a finite “bad prefix” that violates it. For instance here is a bad prefix for S_2 that includes a call to $E.write(<data,x>)$ with no preceding call to $C_0.valid(data)$:

```
[C0.main(); C2.init(); Ret; E.read; Ret(x); C1.parse(x);
Ret(y); C2.process(y); Ret(data); E.write(<data,x>)]
```

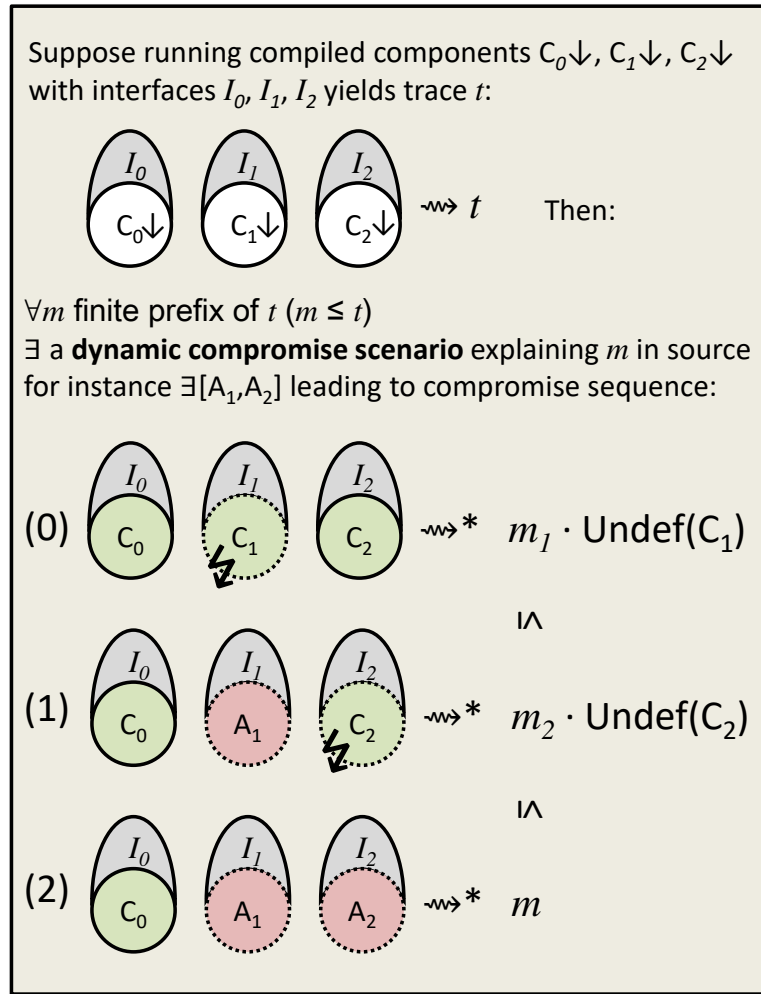
The program from Figure 3.2 cannot produce traces with this bad prefix, but it could do so if we removed the validity check in $C_0.main()$; this variant would invalidate safety property S_2 .

Compiler correctness is often phrased in terms of preserving trace properties in general [Leroy 2009a] (and thus safety properties as a special case). However, this is often predicated on the assumption that the source program has no undefined behavior; if it does, all security guarantees are lost, globally. By contrast, we want our secure compilation criterion to still apply even when some components are dynamically compromised by encountering undefined behavior. In particular, we want to ensure that dynamically compromised components are not able to break the safety properties of the system at the target level any more than equally privileged components without undefined behavior already could in the source.

We call our criterion *Robustly Safe Compartmentalizing Compilation (RSCC)*. It is phrased in terms of a “security game,” illustrated in Figure 3.3 for our running example. With an RSCC compilation chain, given any execution of the compiled and linked components $C_0\downarrow$, $C_1\downarrow$ and, $C_2\downarrow$ producing trace t in the target language, we can explain any (intuitively bad) finite prefix m of t (written $m \leq t$) in terms of the source language. As soon as any component of the program has an undefined behavior though, the semantics of the source language can no longer *directly* help us. Similar to CompCert [Leroy 2009a], we model undefined behavior in our source language as a special event $Undef(C_i)$ that terminates the trace. For instance, in step 0 of Figure 3.3, component C_1 is the first to encounter undefined behavior after producing a prefix m_1 of m .

Since undefined behavior can manifest as arbitrary target-level behavior, the further actions of component C_1 can no longer be explained in terms of its source code. So how can we explain the rest of m in the source language? Our solution in RSCC is to require that one can replace C_1 , the component that encountered undefined behavior, with some other source component A_1 that has the same interface and can produce its part of the whole m in the source language without itself encountering undefined behavior. In order to replace component C_1 with A_1 we have to go back in time and re-execute the program from the beginning obtaining a longer trace, in this case $m_2 \cdot Undef(C_2)$ (where we write “ \cdot ” for appending the event $Undef(C_2)$ to m_2). We iterate this process until all components that encountered undefined behavior have been replaced with new source components that do not encounter undefined behavior and produce the whole m . In the example dynamic compromise scenario from Figure 3.3, this means replacing C_1 with A_1 and C_2 with A_2 , after which the program can produce the whole prefix m in the source.

Let’s now use this RSCC security game to deduce that in our example from Figure 3.2, even compromising both C_1 and C_2 does not break property S_2 at the target level. Assume, for the sake of a contradiction, that a trace of our compiled program breaks property S_2 . Then there exists a finite prefix “ $m \cdot E.write(<data,x>)$ ” such that $C_0.valid(data)$ does not appear in m .



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

```
m = [C0.main(); C2.init(); Ret; E.read; Ret(x); C1.parse(x);
      Ret(y); C2.process(y); Ret(d);
      C0.valid(d); Ret(true); E.write(<d, x>)]
```

```
m1 = [C0.main(); C2.init(); Ret; E.read; Ret(x); C1.parse(x)]
```

```
m2 = [C0.main(); C2.init(); Ret; E.read; Ret(x); C1.parse(x);
      Ret(y); C2.process(y)]
```

Figure 3.3: The *RSCC* dynamic compromise game for our example. We start with all components being uncompromised (in green) and incrementally replace any component that encounters undefined behavior with an arbitrary component (in red) that has the same interface and will do its part of the trace prefix m without causing undefined behavior.

Using *RSCC* we obtain that there exists some dynamic compromise scenario explaining m in the source. The simplest case is when no components are compromised. The most interesting case is when this scenario involves the compromise of both C_1 and C_2 as in Figure 3.3. In this case, replacing C_1 and C_2 with arbitrary A_1 and A_2 with the same interfaces allows us to reproduce the whole bad prefix m in the source (step 2 from Figure 3.3). We can now reason in the source, either informally or using a program logic for robust safety [Swasey et al. 2017], that this cannot happen, since the source code of C_0 does call $C_0.\text{valid}(\text{data})$ and only if it gets true back does it call $E.\text{write}(\langle \text{data}, x \rangle)$.

While in this special case we have only used the last step in the dynamic compromise sequence, where all compromised components have already been replaced (step 2 from Figure 3.3), the previous steps are also useful in general for reasoning about the code our original components execute *before* they get compromised. For instance, this kind of reasoning is crucial for showing property W_2 for the original example from Figure 3.1. Property W_2 gives up on the validity of the written data only if C_2 receives a y that exploits $C_2.\text{handle}(y)$ (vulnerability V_3). However, as discussed above, a compromised C_1 could, in theory, try to compromise C_2 by calling $C_2.\text{process}$ without proper initialization (exploiting vulnerability V_2). Showing that this cannot actually happen requires using step 0 of the game from Figure 3.3, which gives us that the original compiled program obtained by linking $C_0\downarrow$, $C_1\downarrow$ and, $C_2\downarrow$ can produce the trace $m_1 \cdot \text{Undef}(C_1)$, for some prefix m_1 of the bad trace prefix in which $C_2.\text{process}$ is called without calling $C_2.\text{init}$ first. But it is easy to check that the straight-line code of the $C_1.\text{main}()$ procedure can only cause undefined behavior *after* it has called $C_2.\text{init}$, contradicting the existence of a bad trace exploiting V_2 .

3.3 Formally Defining *RSCC*

For pedagogical purposes, we define *RSCC* in stages, incrementally adapting the *Robust Safety Properties Preservation* (*RSC*) criterion introduced in §2.2.3. We first bring *RSC* to *unsafe languages with undefined behavior* (§3.3.1), and then further extend its protection to any set of *mutually distrustful components* (§3.3.2). These ideas lead to the more elaborate *RSCC* property (§3.3.3), which directly captures the informal dynamic compromise game from §3.2. These definitions are generic, and will be illustrated with a concrete instance in §3.4. In the reminder of this section, we describe an effective and general proof technique for *RSCC* (§3.3.4). Finally, we investigate the class of trace properties preserved by our simplest definition (§3.3.5) and contrast our dynamic compromise model with previous work in the static compromise model [Juglaret et al. 2016] (§3.3.6).

3.3.1 *RSC*^{DC}: Dynamic Compromise

The *RSC* criterion from §2.2.3 is about protecting a partial program written in a *safe* source language against adversarial target-level contexts. We now adapt the idea behind *RSC* to an *unsafe* source language with undefined behavior, in which the protected partial program itself

can become compromised. As explained in §3.2, we model undefined behavior as a special `Undef` event terminating the trace: whatever happens afterwards at the target level can no longer be explained in terms of the code of the source program. We further assume that each undefined behavior in the source language can be attributed to the part of the program that causes it by labeling the `Undef` event with “blame the program” (P) or “blame the context” (C) (while in §3.3.2 we will blame the precise component encountering undefined behavior).

Definition 3.3.1. A compilation chain provides *Robustly Safe Compilation with Dynamic Compromise* (RSC^{DC}) iff

$$\forall P \ C_T \ t. \ C_T[P\downarrow] \rightsquigarrow t \Rightarrow \forall m \leq t. \exists C_S \ t'. \ C_S[P] \rightsquigarrow t' \wedge (m \leq t' \vee t' \prec_P m).$$

Roughly, this definition relaxes RSC by forgoing protection for the partial program P after it encounters undefined behavior. More precisely, instead of always requiring that the trace t' produced by $C_S[P]$ contain the entire prefix m (i.e., $m \leq t'$), we also allow t' to be itself a prefix of m followed by an undefined behavior in P , which we write as $t' \prec_P m$ (i.e., $t' \prec_P m \triangleq \exists m' \leq m. t' = (m' \cdot \text{Undef}(P))$). In particular, the context C_S is guaranteed to be free of undefined behavior before the whole prefix m is produced or P encounters undefined behavior. However, nothing prevents C_S from passing values to P that try to trick P into causing undefined behavior.

To illustrate, consider the partial program P defined below.

```

program P {
  import E.write; export foo;
  foo(x) {
    y := P.process(x);
    E.write(y);
  }
  // can encounter Undef for some x
  process(x) { ... }
}

context C_S {
  import E.read, P.foo;
  main() {
    x := E.read();
    P.foo(x);
  }
}

```

Suppose we compile P with a compilation chain that satisfies RSC^{DC} , link the result with a target context C_T obtaining $C_T[P\downarrow]$, execute this and observe the following finite trace prefix:

$m = [E.read(); \text{Ret}("feedbeef"); P.foo("feedbeef"); E.write("bad")]$

According to RSC^{DC} there exists a source-level context C_S (for instance the one above) that explains the prefix m in terms of the source language in one of two ways: either $C_S[P]$ can do the entire m in the source, or $C_S[P]$ encounters an undefined behavior in P after a prefix of m , for instance the following one:

$t' = [E.read(); \text{Ret}("feedbeef"); P.foo("feedbeef"); \text{Undef}(P)]$

As in `CompCert` [Leroy 2009a, Regehr 2010], we treat undefined behaviors as *observable* events at the end of the execution trace, allowing compiler optimizations that move an undefined behavior to an earlier point in the execution, but not past any other observable event. While some other C compilers would need to be adapted to respect this discipline [Regehr 2010], limiting the temporal scope of undefined behavior is a necessary prerequisite for achieving

security against dynamic compromise. Moreover, if trace events are coarse enough (e.g., system calls and cross-component calls) we expect this restriction to have a negligible performance impact in practice.

One of the top-level CompCert theorems does, in fact, already capture dynamic compromise in a similar way to RSC^{DC} . Using our notations this CompCert theorem looks as follows:

$$\forall P t. (P \Downarrow) \rightsquigarrow t \Rightarrow \exists t'. P \rightsquigarrow t' \wedge (t' = t \vee t' \prec t)$$

This says that if a compiled whole program $P \Downarrow$ can produce a trace t with respect to the target semantics, then in the source P can produce either the same trace or a prefix of t followed by undefined behavior. In particular this theorem does provide guarantees to undefined programs up to the point at which they encounter undefined behavior. The key difference compared to our secure compilation chains is that CompCert does not restrict undefined behavior *spatially*: in CompCert undefined behavior breaks all security guarantees of the *whole* program, while in our work we restrict undefined behavior to the component that causes it. This should become clearer in the next section, where we explicitly introduce components, but even in RSC^{DC} we can already imagine $P \Downarrow$ as a set of uncompromised components for trace prefix m , and C_T as a set of already compromised ones.

A smaller difference with respect to the CompCert theorem is that (like RSC) RSC^{DC} only looks at finite prefixes in order to simplify the difficult proof step of context back-translation, which is not a concern that appears in CompCert and the usual verified compilers. Abate et al. [2018b] precisely characterize the subclass of safety properties that is preserved by RSC^{DC} even in adversarial contexts.

3.3.2 RSC_{MD}^{DC} : Mutually Distrustful Components

RSC^{DC} gives a model of dynamic compromise for secure compilation, but is still phrased in terms of protecting a trusted partial program from an untrusted context. We now adapt this model to protect any set of *mutually distrustful components* with clearly specified privileges from an untrusted context. Following Juglaret et al.’s [2016] work in the full abstraction setting, we start by taking both partial programs and contexts to be sets of components; linking a program with a context is then just set union. We compile sets of components by separately compiling each component. Each component is assigned a well-defined interface that precisely captures its *privilege*; components can only interact as specified by their interfaces. Most importantly, context back-translation respects these interfaces: each component of the target context is mapped back to a source component with exactly the same interface. As Juglaret et al. argue, least-privilege design crucially relies on the fact that, when a component is compromised, it does not gain any more privileges.

Definition 3.3.2. A compilation chain provides *Robustly Safe Compilation with Dynamic Compromise and Mutual Distrust* (RSC_{MD}^{DC}) if there exists a back-translation function \uparrow taking a finite trace prefix m and a component interface I_i to a source component with the same interface,

such that, for any compatible interfaces I_P and I_C ,

$$\begin{aligned} & \forall P:I_P. \forall C_T:I_C. \forall t. (C_T \cup P \downarrow) \rightsquigarrow t \Rightarrow \forall m \leq t. \\ & \exists t'. (\{(m, I_i) \uparrow \mid I_i \in I_C\} \cup P) \rightsquigarrow t' \wedge (m \leq t' \vee t' \prec_{I_P} m). \end{aligned}$$

This definition closely follows RSC^{DC} , but it restricts programs and contexts to compatible interfaces I_P and I_C . We write $P : I$ to mean “partial program P satisfies interface I .” The source-level context is obtained by applying the back-translation function \uparrow pointwise to all the interfaces in I_C . As before, if the prefix m is cropped prematurely because of an undefined behavior, then this undefined behavior must be in one of the program components, not in the back-translated context components ($t' \prec_{I_P} m$).

3.3.3 Formalizing *RSCC*

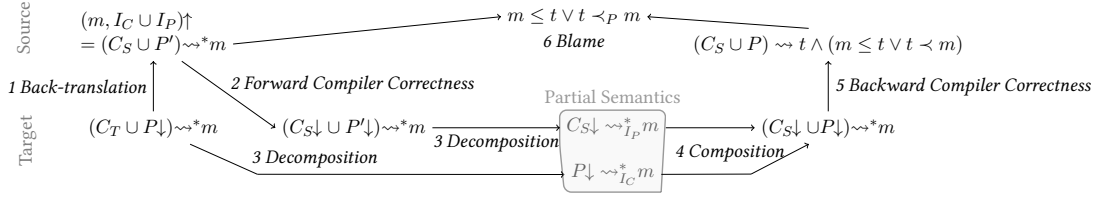
Using these ideas, we now define *RSCC* by following the dynamic compromise game illustrated in Figure 3.3. We use the notation $P \rightsquigarrow^* m$ when there exists a trace t that extends m (i.e., $m \leq t$) such that $P \rightsquigarrow t$. We start with all components being uncompromised and incrementally replace each component that encounters undefined behavior in the source with an arbitrary component with the same interface that may now attack the remaining components.

Definition 3.3.3. A compilation chain provides *Robustly Safe Compartmentalizing Compilation (RSCC)* iff \forall compatible interfaces I_1, \dots, I_n ,

$$\begin{aligned} & \forall C_1:I_1, \dots, C_n:I_n. \forall m. \{C_1 \downarrow, \dots, C_n \downarrow\} \rightsquigarrow^* m \Rightarrow \\ & \exists A_{i_1}:I_{i_1}, \dots, A_{i_k}:I_{i_k}. \\ & \quad (1) \forall j \in 1 \dots k. \exists m_j. (m_j \prec_{I_{i_j}} m) \wedge (m_{j-1} \prec_{I_{i_{j-1}}} m_j) \wedge \\ & \quad \quad (\{C_1, \dots, C_n\} \setminus \{C_{i_1}, \dots, C_{i_{j-1}}\} \cup \{A_{i_1}, \dots, A_{i_{j-1}}\}) \rightsquigarrow^* m_j \\ & \quad \wedge (2) (\{C_1, \dots, C_n\} \setminus \{C_{i_1}, \dots, C_{i_k}\} \cup \{A_{i_1}, \dots, A_{i_k}\}) \rightsquigarrow^* m. \end{aligned}$$

This says that C_{i_1}, \dots, C_{i_k} constitutes a compromise sequence corresponding to finite prefix m produced by a compiled set of components $\{C_1 \downarrow, \dots, C_n \downarrow\}$. In this compromise sequence each component C_{i_j} is taken over by the already compromised components at that point in time $\{A_{i_1}, \dots, A_{i_{j-1}}\}$ (part 1). Moreover, after replacing all the compromised components $\{C_{i_1}, \dots, C_{i_k}\}$ with their corresponding source components $\{A_{i_1}, \dots, A_{i_k}\}$ the entire m can be reproduced in the source language (part 2).

This formal definition allows us to play an iterative game in which components that encounter undefined behavior successively become compromised and attack the other components. This is the first security definition in this space to support both dynamic compromise and mutual distrust, whose interaction is subtle and has eluded previous attempts at characterizing the security guarantees of compartmentalizing compilation as extensions of fully abstract compilation [Juglaret et al. 2016] (further discussed in §3.5).

Figure 3.4: Outline of our generic proof technique for RSC_{MD}^{DC}

3.3.4 A Generic Proof Technique for $RSCC$

We now describe an effective and general proof technique for $RSCC$. First, we observe that the slightly simpler RSC_{MD}^{DC} implies $RSCC$. Then we provide a generic proof in Coq that any compilation chain obeys RSC_{MD}^{DC} if it satisfies certain well-specified assumptions on the source and target languages and the compilation chain.

Our proof technique yields simpler and more scalable proofs than previous work in this space [Juglaret et al. 2016]. In particular, it allows us to directly reuse a compiler correctness result *à la* CompCert, which supports separate compilation but only guarantees correctness for whole programs [Kang et al. 2016]; which avoids proving any other simulations between the source and target languages. Achieving this introduces some slight complications in the proof structure, but it nicely separates the correctness and security proofs and allows us to more easily tap into the CompCert infrastructure. Finally, since only the last step of our proof technique is specific to unsafe languages, our technique can be further simplified to provide scalable proofs of vanilla RSC for safe source languages [Abate et al. 2018b, Patrignani and Garg 2018].

RSC_{MD}^{DC} implies $RSCC$ The first step in our proof technique reduces $RSCC$ to RSC_{MD}^{DC} , using a theorem showing that $RSCC$ can be obtained by iteratively applying RSC_{MD}^{DC} . This result crucially relies on back-translation in RSC_{MD}^{DC} being performed pointwise and respecting interfaces, as explained in §3.3.2.

Theorem 3.3.4. RSC_{MD}^{DC} implies $RSCC$.

We proved this by defining a non-constructive function that produces the compromise sequence A_{i_1}, \dots, A_{i_1} by case analysis on the disjunction in the conclusion of RSC_{MD}^{DC} (using excluded middle in classical logic). If $m \leq t'$ we are done and we return the sequence we accumulated so far, while if $t' <_P m$ we obtain a new compromised component $c_i : I_i$ that we back-translate using $(m, I_i) \uparrow$ and add to the sequence before iterating this process.

Generic RSC_{MD}^{DC} proof outline Our high-level RSC_{MD}^{DC} proof is generic and works for any compilation chain that satisfies certain well-specified assumptions, which we introduce informally for now, leaving details to the end of this sub-section. The RSC_{MD}^{DC} proof for the compiler chain in §3.4 proves all these assumptions.

The proof outline is shown in Figure 3.4. We start (in the bottom left) with a complete target-level program $C_T \cup P \downarrow$ producing a trace with a finite prefix m that we assume contains no

undefined behavior (since we expect that the final target of our compilation will be a machine for which all behavior is defined). The prefix m is first back-translated to synthesize a complete source program $C_S \cup P'$ producing m (the existence and correctness of this back-translation are Assumption 1). For example, for the compiler in §3.4, each component C_i produced by back-translation uses a private counter to track how many events it has produced during execution. Whenever C_i receives control, following an external call or return, it checks this counter to decide what event to emit next, based on the order of its events on m (see §3.4.3 for details).

The generated source program $C_S \cup P'$ is then separately compiled to a target program $C_S \downarrow \cup P' \downarrow$ that, by compiler correctness, produces again the same prefix m (Assumption 2). Now from $(C_T \cup P \downarrow) \rightsquigarrow^* m$ and $(C_S \downarrow \cup P' \downarrow) \rightsquigarrow^* m$ we would like to obtain $(C_S \downarrow \cup P \downarrow) \rightsquigarrow^* m$ by first “decomposing” (Assumption 3) separate executions for $P \downarrow$ and $C_S \downarrow$, which we can then “compose” (Assumption 4) again into a complete execution for $(C_S \downarrow \cup P \downarrow)$. However, since $P \downarrow$ and C_S are not complete programs, how should they execute? To answer this we rely on a *partial semantics* that captures the traces of a partial program when linked with *any* context satisfying a given interface. When the partial program is running, execution is the same as in the normal operational semantics of the target language; when control is passed to the context, arbitrary actions compatible with its interface are non-deterministically executed. Using this partial semantics we can execute $C_S \downarrow$ with respect to the interface of $P \downarrow$, and $P \downarrow$ with respect to the interface of $C_S \downarrow$, as needed for the decomposition and composition steps of our proof.

Once we know that $(C_S \downarrow \cup P \downarrow) \rightsquigarrow^* m$, we use compiler correctness again—now in the backwards direction (Assumption 5)—to obtain an execution of the source program $C_S \cup P$ producing trace t . Because our source language is unsafe, however, t need not be an extension of m : it can end earlier with an undefined behavior (§3.3.1). So the final step in our proof shows that if the source execution ends earlier with an undefined behavior ($t' \prec m$), then this undefined behavior can only be caused by P (i.e., $t' \prec_P m$), not by C_S , which was correctly generated by our back-translation (Assumption 6).

Assumptions of the RSC_{MD}^{DC} proof The generic RSC_{MD}^{DC} proof outlined above relies on assumptions about the compartmentalizing compilation chain. In the reminder of this subsection we give details about these assumptions, while still trying to stay high level by omitting some of the low-level details in our Coq formalization.

The first assumption we used in the proof above is that every trace prefix that a target program can produce can also be produced by a source program with the same interface. A bit more formally, we assume the existence of a *back-translation function* \uparrow that given a finite prefix m that can be produced by a whole target program P_T , returns a whole source program with the same interface I_P as P_T and which can produce the same prefix m (i.e., $(m, I_P) \uparrow \rightsquigarrow^* m$).

Assumption 1 (Back-translation).

$$\exists \uparrow. \forall P: I_P. \forall m \text{ defined. } P \rightsquigarrow^* m \Rightarrow (m, I_P) \uparrow : I_P \wedge (m, I_P) \uparrow \rightsquigarrow^* m$$

Back-translating only finite prefixes simplifies our proof technique but at the same time limits it to only safety properties. While the other assumptions from this section can probably also

be proved for infinite traces, there is no general way to define a finite program that produces an arbitrary infinite trace. We leave devising scalable back-translation proof techniques that go beyond safety properties to future work.

It is not always possible to take an *arbitrary* finite sequence of events and obtain a source program that realizes it. For example, in a language with a call stack and events $\{\text{call}, \text{return}\}$, there is no program that produces the single event trace `return`, since every `return` must be preceded by a `call`. Thus we only assume we can back-translate prefixes that are produced by the target semantics.

As further discussed in §3.5, similar back-translation techniques that start from finite execution prefixes have been used to prove fully abstract compilation [Jeffrey and Rathke 2005a, Patrignani and Clarke 2015] and very recently RSC [Patrignani and Garg 2018] and stronger variants, such as the one from §2.6.4. Our back-translation, on the other hand, produces not just a source context, but a whole program. In the top-left corner of Figure 3.4, we assume that this resulting program, $(m, I_C \cup I_P)^\uparrow$, can be partitioned into a context C_S that satisfies the interface I_C , and a program P' that satisfies I_P .

Our second assumption is a form of forward compiler correctness for unsafe languages and a direct consequence of a forward simulation proof in the style of CompCert [Leroy 2009a]. We assume separate compilation, in the style of a recent extension proposed by Kang et al. [2016] and implemented in CompCert since version 2.7. Our assumption says that if a whole program composed of parts P and C (written $C \cup P$) produces the finite trace prefix m that does not end with undefined behavior (m *defined*) then P and C when separately compiled and linked together ($C \downarrow \cup P \downarrow$) can also produce m .

Assumption 2 (Forward Compiler Correctness with Separate Compilation and Undefined Behavior).

$$\forall C \ P. \forall m \text{ defined}. (C \cup P) \rightsquigarrow^* m \Rightarrow (C \downarrow \cup P \downarrow) \rightsquigarrow^* m$$

The next assumption we make is *decomposition*, stating that if a program obtained by linking two partial programs P_T and C_T produces a finite trace prefix m that does not end in an undefined behavior in the complete semantics, then each of the two partial programs (below we take P_T , but the C_T case is symmetric) can produce m in the partial semantics:

Assumption 3 (Decomposition).

$$\forall P_T : I_P. \forall C_T : I_C. \forall m \text{ defined}. (C_T \cup P_T) \rightsquigarrow^* m \Rightarrow P_T \rightsquigarrow_{I_C}^* m$$

The converse of decomposition, *composition*, states that if two partial programs with matching interfaces produce the same prefix m with respect to the partial semantics, then they can be linked to produce the same m in the complete semantics:

Assumption 4 (Composition). For any I_P, I_C compatible interfaces:

$$\forall P_T : I_P. \forall C_T : I_C. \forall m. P_T \rightsquigarrow_{I_C}^* m \wedge C_T \rightsquigarrow_{I_P}^* m \Rightarrow (C_T \cup P_T) \rightsquigarrow^* m$$

When taken together, composition and decomposition capture that the partial semantics of the target language is adequate with respect to its complete counterpart. This adequacy notion is tailored to the *RSC* property and thus different from the requirement that a so called “trace semantics” is fully abstract [Jeffrey and Rathke 2005a, Patrignani and Clarke 2015].

In order to get back to the source language our proof uses a backwards compiler correctness assumption, again with separate compilation. As also explained in §3.3.1, we need to take into account that a trace prefix m in the target can be explained in the source either by an execution producing m or by one ending in an undefined behavior (i.e., producing $t \prec m$).

Assumption 5 (Backward Compiler Correctness with Separate Compilation and Undefined Behavior).

$$\forall C \ P \ m. (C \downarrow \cup P \downarrow) \rightsquigarrow^* m \Rightarrow \exists t. (C \cup P) \rightsquigarrow t \wedge (m \leq t \vee t \prec m)$$

Finally, we assume that the context obtained by back-translation can’t be blamed for undefined behavior:

Assumption 6 (Blame). $\forall C_S : I_C. \forall P, P' : I_P. \forall m \text{ defined}. \forall t.$

If $(C_S \cup P') \rightsquigarrow^* m$ and $(C_S \cup P) \rightsquigarrow t$ and $t \prec m$ then $m \leq t \vee t \prec_P m$.

We used Coq to prove the following theorem that puts together the assumptions from this subsection to show RSC_{MD}^{DC} :

Theorem 3.3.5. The assumptions above imply RSC_{MD}^{DC} .

3.3.5 Class of safety properties preserved by RSC^{DC}

Since *RSC* corresponds exactly to preserving robust safety properties (§2.2.3), one might wonder what properties RSC^{DC} preserves. In fact, RSC^{DC} corresponds exactly to preserving the following class Z_P against an adversarial context:

Definition 3.3.6. $Z_P \triangleq \text{Safety} \cap \text{Closed}_{\prec_P}$, where

$$\begin{aligned} \text{Safety} &\triangleq \{\pi \mid \forall t \notin \pi. \exists m \leq t. \forall t' \geq m. t' \notin \pi\} \\ \text{Closed}_{\prec_P} &\triangleq \{\pi \mid \forall t \in \pi. \forall t'. t \prec_P t' \Rightarrow t' \in \pi\} \\ &= \{\pi \mid \forall t' \notin \pi. \forall t. t \prec_P t' \Rightarrow t \notin \pi\} \end{aligned}$$

The class of properties Z_P is defined as the intersection of *Safety* and the class Closed_{\prec_P} of properties closed under extension of traces with undefined behavior in P [Leroy 2009a]. If a property π is in Closed_{\prec_P} and it allows a trace t that ends with an undefined behavior in P —i.e., $\exists m. t = m \cdot \text{Undef}(P)$ —then π should also allow any extension of the trace m —i.e., any trace t' that has m as a prefix. The intuition is simple: the compilation chain is free to implement a trace with undefined behavior in P as an arbitrary trace extension, so if the property accepts traces with undefined behavior it should also accept their extensions. Conversely, if a property π in Closed_{\prec_P} rejects a trace t' , then for any prefix m of t' the property π should also reject the trace $m \cdot \text{Undef}(P)$.

For a negative example that is not in $Closed_{\prec_P}$, consider the following formalization of the property S_1 from §3.2, requiring all writes in the trace to be preceded by a corresponding read:

$$S_1 = \{t \mid \forall m \text{ d } x. m \cdot E.write(\langle d, x \rangle) \leq t \\ \Rightarrow \exists m'. m' \cdot E.read \cdot Ret(x) \leq m\}$$

While property S_1 is *Safety* it is not $Closed_{\prec_P}$. Consider the trace $t' = [C_0.main(); E.write(\langle d, x \rangle)] \notin S_1$ that does a write without a read and thus violates S_1 . For S_1 to be $Closed_{\prec_P}$ it would have to reject not only t' , but also $[C_0.main(); Undef(P)]$ and $Undef(P)$, which it does not. One can, however, define a stronger variant of S_1 that is in Z_P :

$$S_1^{Z_P^+} = \{t \mid \forall m \text{ d } x. (m \cdot E.write(\langle d, x \rangle) \leq t \vee m \cdot Undef(P) \leq t) \\ \Rightarrow \exists m'. m' \cdot E.read \cdot Ret(x) \leq m\}$$

The property $S_1^{Z_P^+}$ requires any write *or undefined behavior* in P to be preceded by a corresponding read. While this property is quite restrictive, it does hold (vacuously) for the strengthened system in Figure 3.2 when taking $P = \{C_0\}$ and $C = \{C_1, C_2\}$, since we assumed that C_0 has no undefined behavior.

Using Z_P , we proved an equivalent RSC^{DC} characterization:

Theorem 3.3.7.

$$RSC^{DC} \iff \left(\begin{array}{l} \forall P \pi \in Z_P. (\forall C_S t. C_S[P] \rightsquigarrow t \Rightarrow t \in \pi) \\ \Rightarrow (\forall C_T t. C_T[P \downarrow] \rightsquigarrow t \Rightarrow t \in \pi) \end{array} \right)$$

This theorem shows that RSC^{DC} is equivalent to the preservation of all properties in Z_P for all P . One might still wonder how one obtains such robust safety properties in the source language, given that the execution traces can be influenced not only by the partial program but also by the adversarial context. In cases in which the trace records enough information so that one can determine the originator of each event, robust safety properties can explicitly talk only about the events of the program, not the ones of the context. Moreover, once we add interfaces in RSC_{MD}^{DC} (§3.3.2) we are able to effectively restrict the context from directly performing certain events (e.g., certain system calls), and the robust safety property can then be about these privileged events that the sandboxed context cannot directly perform.

One might also wonder what stronger property does one have to prove in the source in order to obtain a certain safety property π in the target using an RSC^{DC} compiler in the case in which π is not itself in Z_P . Especially when all undefined behavior is already gone in the target language, it seems natural to look at safety properties such as $S_1 \notin Z_P$ above that do not talk at all about undefined behavior. For S_1 above, we manually defined the stronger property $S_1^{Z_P^+} \in Z_P$ that is preserved by an RSC^{DC} compiler. In fact, given any safety property π we can easily define $\pi^{Z_P^+}$ that is in Z_P , is stronger than π , and is otherwise as permissive as possible:

$$\pi^{Z_P^+} \triangleq \pi \cap \{t \mid \forall t'. t \prec_P t' \Rightarrow t' \in \pi\}$$

We can also easily answer the dual question asking what is left of an arbitrary safety property established in the source when looking at the target of an RSC^{DC} compiler:

$$\pi^{Z_P^-} \triangleq \pi \cup \{t' \mid \exists t \in \pi. t \prec_P t' \vee t' \leq t\}$$

3.3.6 Comparison to Static Compromise

It is instructive to contrast the dynamic compromise model of our RSC^{DC} criterion with previous work in the static compromise model by Juglaret et al. [2016]. While the secure compilation criteria of Juglaret et al. are variants of full abstraction, the core idea is easy to port to robust safety preservation:

Definition 3.3.8. A compilation chain provides *Robustly Safe Compilation with Static Compromise* (RSC^{SC}) iff

$$\forall P \ C_T \ t. \textcolor{red}{P \text{ fully defined}} \wedge C_T[P \downarrow] \rightsquigarrow t \Rightarrow \forall m \leq t. \exists C_S \ t'. C_S[P] \rightsquigarrow t' \wedge m \leq t'.$$

Instead of the second disjunct in the conclusion of RSC^{DC} (§3.3.1), which deals with the possibility of undefined behavior in P , RSC^{SC} imposes a strong precondition, requiring that P does not cause undefined behavior in *any* context. In our trace model, which keeps track of whether the program or context causes undefined behavior, P *fully defined* can be defined simply as $\neg(\exists C_S \ m. C_S[P] \rightsquigarrow m \cdot \text{Undef}(P))$. With these definitions in place one can easily show that RSC^{DC} is strictly stronger than RSC^{SC} . For proving that RSC^{DC} implies RSC^{SC} all we have to show is that a fully defined P cannot be blamed for undefined behavior, which is true by the definition of full definedness. For proving the strictness of this implication it suffices to exhibit a compiler that only restricts the spatial scope of undefined behavior but not the temporal scope, for instance because it performs optimizations that aggressively use the assumption that the program does not have undefined behavior (e.g., unrestricted code motion, unrestricted backwards propagation of static analysis “facts” derived from the absence of undefined behavior). GCC and LLVM are already known to perform such aggressive optimizations [Regehr 2010].

As we already explained in §3.1, the full definedness precondition in RSC^{SC} is a serious practical limitation: a partial program P whose code contains any vulnerability that might potentially manifest itself as undefined behavior is given no guarantees whatsoever, irrespective of whether an attacker actually exploits these vulnerabilities. Moreover, a vulnerable P loses all guarantees from the start of the execution—possibly long before any actual compromise. In §3.5 we will argue that this limitation to static compromise scenarios seems inescapable for definitions based on full abstraction, like the one of Juglaret et al. [2016]. In this section we showed that, by moving away from full abstraction and by restricting the temporal scope of undefined behavior, we can overcome this limitation and support a dynamic compromise model.

3.4 Secure Compilation Chain

We designed a simple proof-of-concept compilation chain to illustrate the $RSCC$ property. The compilation chain is implemented in Coq and outlined in Figure 3.5. The *source* language is a simple, unsafe imperative language with buffers, procedures, and components (§3.4.1). It is first compiled to an intermediate *compartmentalized machine* featuring a compartmentalized, block-structured memory, a protected call stack, and a RISC-like instruction set augmented with an `Alloc` instruction for dynamic storage allocation plus cross-component `Call` and `Return`

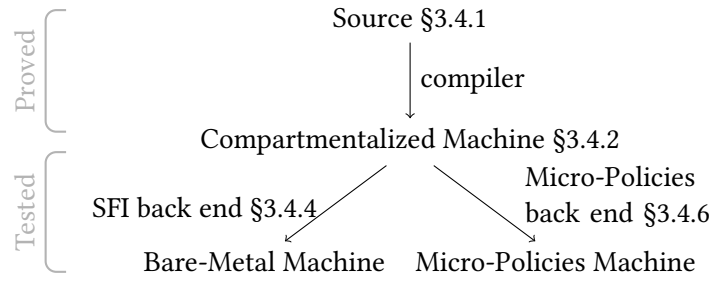


Figure 3.5: Our secure compilation chain

instructions (§3.4.2). We can then choose one of two back ends, which use different techniques to enforce the abstractions of the compartmentalized machine against realistic machine-code-level attackers, protecting the integrity of component memories and enforcing interfaces and cross-component call/return discipline.

When the compartmentalized machine encounters undefined behavior, both back ends instead produce an extended trace that respects high-level abstractions; however, they achieve this in very different ways. The *SFI back end* (§3.4.4) targets a *bare-metal machine* that has no protection mechanisms and implements an inline reference monitor purely in software, by instrumenting code to add address masking operations that force each component’s writes and (most) jumps to lie within its own memory. The *Micro-policies back end* (§3.4.6), on the other hand, relies on specialized hardware [Dhawan et al. 2015b] to support a novel tag-based reference monitor for compartmentalization. These approaches have complementary advantages: SFI requires no specialized hardware, while micro-policies can be engineered to incur little overhead [Dhawan et al. 2015b] and are a good target for formal verification [Azevedo de Amorim et al. 2015] due to their simplicity. Together, these two back ends provide evidence that our *RSCC* security criterion is compatible with any sufficiently strong compartmentalization mechanism. It seems likely that other mechanisms such as capability machines [Watson et al. 2015b] could also be used to implement the compartmentalized machine and achieve *RSCC*.

Both back ends target variants of a simple RISC machine. In contrast to the abstract, block-based memory model used at higher levels of the compilation chain, the machine-level memory is a single infinite array addressed by mathematical integers. (Using unbounded integers is a simplification that we hope to remove in the future, e.g. by applying the ideas of Mullen et al. [2016].) All compartments must share this flat address space, so—without proper protection—compromised components can access buffers out-of-bounds and read or overwrite the code and data of other components. Moreover, machine-level components can ignore the stack discipline and jump to arbitrary locations in memory.

We establish high confidence in the security of our compilation chain with a combination of proof and testing. For the compiler from the source language to the compartmentalized machine, we prove *RSCC* in Coq (§3.4.3) using the proof technique of §3.3.4. For the SFI back end, we use property-based testing with QuickChick [Paraskevopoulou et al. 2015] to systematically test RSC_{MD}^{DC} .

$e ::=$	v	values
	$ \text{ local}$	local static buffer
	$ e_1 \otimes e_2$	binary operations
	$ e_1; e_2$	sequence
	$ \text{ if } e_1 \text{ then } e_2 \text{ else } e_3$	conditional
	$ \text{ alloc } e$	memory allocation
	$!e$	dereferencing
	$ e_1 := e_2$	assignment
	$ C.P(e)$	procedure call
	$ \text{ exit}$	terminate

Figure 3.6: Syntax of source language expressions

3.4.1 Source Language

The source language from this section was designed with simplicity in mind. Its goal was to allow us to explore the foundational ideas of this work and illustrate them in the simplest possible concrete setting, keeping our formal proofs tractable. The language is expression based (see Figure 3.6). A program is composed of an interface, a set of procedures, and a set of static buffers. Interfaces contain the names of the procedures that the component *exports* to and *imports* from other components. Each procedure body is a single expression whose result value is returned to the caller. Internal and external calls share the same global, protected call stack. Additional buffers can be allocated dynamically. As in C, memory is manually managed; out-of-bounds accesses lead to undefined behavior.

Values include integers, pointers, and an undefined value \top , which is obtained when reading from an uninitialized piece of memory or as the result of an erroneous pointer operation. As in CompCert and LLVM [Lee et al. 2017], our semantics propagates these \top values and yields an undefined behavior if a \top value is ever inspected. (The C standard, by contrast, specifies that a program is undefined as soon as an uninitialized read or bad pointer operation takes place.)

Memory Model The memory model for both source and compartmentalized machine is a slightly simplified version of the one used in CompCert [Leroy and Blazy 2008]. Each component has an infinite memory composed of finite blocks, each an array of values. Accordingly, a pointer is a triple (C, b, o) , where C is the identifier of the component that owns the block, b is a unique block identifier, and o is an offset inside the block. Arithmetic operations on pointers are limited to testing equality, testing ordering (of pointers into the same block), and changing offsets. Pointers cannot be cast to or from integers. Dereferencing an integer yields undefined behavior. For now, components are not allowed to exchange pointers; as a result, well-defined components cannot access each others' memories at all. We hope to lift this restriction in the near future. This abstract memory model is shared by the compartmentalized machine and is mapped to a more realistic flat address space by the back ends.

Events Following CompCert, we use a labeled operational semantics whose events include all interactions of the program with the external world (e.g., system calls), plus events track-

$\text{instr} ::=$	Nop	$ $	Halt	$ $	$\text{Jal } l$
	$ \text{Const } i \rightarrow r$			$ $	$\text{Jump } r$
	$ \text{Mov } r_s \rightarrow r_d$			$ $	$\text{Call } C \ P$
	$ \text{BinOp } r_1 \otimes r_2 \rightarrow r_d$			$ $	Return
	$ \text{Load } *r_p \rightarrow r_d$			$ $	$\text{Bnz } r \ l$
	$ \text{Store } *r_p \leftarrow r_s$			$ $	$\text{Alloc } r_1 \ r_2$

Figure 3.7: Instructions of compartmentalized machine

ing control transfers from one component to another. Every call to an exported procedure produces a visible event $C \text{ Call } P(n) \ C'$, recording that component C called procedure P of component C' , passing argument n . Cross-component returns are handled similarly. All other computations, including calls and returns within the same component, result in silent steps in the operational semantics.

3.4.2 The Compartmentalized Machine

The compartmentalized intermediate machine aims to be as low-level as possible while still allowing us to target our two rather different back ends. It features a simple RISC-like instruction set (Figure 3.7) with two main abstractions: a block-based memory model and support for cross-component calls. The memory model leaves the back ends complete freedom in their layout of blocks. The machine has a small fixed number of registers, which are the only shared state between components. In the syntax, l represent *labels*, which are resolved to pointers in the next compilation phase.

The machine uses two kinds of call stacks: a single protected global stack for cross-component calls plus a separate unprotected one for the internal calls of each component. Besides the usual Jal and Jump instructions, which are used to compile internal calls and returns, two special instructions, Call and Return , are used for cross-component calls. These are the only instructions that can manipulate the global call stack.

The operational semantics rules for Call and Return are presented in Figure 3.8. A state is composed of the current executing component C , the protected stack σ , the memory mem , the registers reg and the program counter pc . If the instruction fetched from the program counter is a Call to procedure P of component C' , the semantics produces an event α recording the caller, the callee, the procedure and its argument, which is stored in register R_COM . The protected stack σ is updated with a new frame containing the next point in the code of the current component. Registers are mostly invalidated at Calls ; reg_\top has all registers set to \top and only two registers are passed on: R_COM contains the procedure's argument and R_RA contains the return address. So no data accidentally left by the caller in the other registers can be relied upon; instead the compiler saves and restores the registers. Finally, there is a redundancy between the protected stack and R_RA because during the Return the protected frame is used to verify that the register is used correctly; otherwise the program has an undefined behavior.

$$\begin{array}{c}
\text{fetch}(E, pc) = \text{Call } C' P \quad C \neq C' \\
P \in C.\text{import} \quad \text{entry}(E, C', P) = pc' \\
\text{reg}' = \text{reg}_T[\text{R_COM} \leftarrow \text{reg}[\text{R_COM}], \text{R_RA} \leftarrow pc + 1] \\
\alpha = C \text{ Call}(P, \text{reg}[\text{R_COM}]) C' \\
\hline
E \vdash (C, \sigma, \text{mem}, \text{reg}, pc) \xrightarrow{\alpha} (C', (pc + 1) :: \sigma, \text{mem}, \text{reg}', pc') \\
\\
\text{fetch}(E, pc) = \text{Return} \quad C \neq C' \\
\text{reg}[\text{R_RA}] = pc' \quad \text{component}(pc') = C' \\
\text{reg}' = \text{reg}_T[\text{R_COM} \leftarrow \text{reg}[\text{R_COM}]] \\
\alpha = C \text{ Return}(\text{reg}[\text{R_COM}]) C' \\
\hline
E \vdash (C, pc' :: \sigma, \text{mem}, \text{reg}, pc) \xrightarrow{\alpha} (C', \sigma, \text{mem}, \text{reg}', pc')
\end{array}$$

Figure 3.8: Compartmentalized machine semantics

3.4.3 RSCC Proof in Coq

We have proved that a compilation chain targeting the compartmentalized machine satisfies *RSCC*, applying the technique from §3.3.4. As explained in §3.2, the responsibility for enforcing secure compilation can be divided among the different parts of the compilation chain. In this case, it is the target machine of §3.4.2 that enforces compartmentalization, while the compiler itself is simple, standard, and not particularly interesting (so omitted here).

For showing RSC_{MD}^{DC} , all the assumptions from §3.3.4 are proved using simulations. Most of this proof is formalized in Coq: the only non-trivial missing pieces are compiler correctness (Assumptions 2 and 5) and composition (Assumption 4). The first is standard and essentially orthogonal to secure compilation; eventually, we hope to scale the source language up to a compartmentalized variant of C and reuse CompCert’s mechanized correctness proof. A mechanized proof of composition is underway. Despite these missing pieces, our formalization is more detailed than previous paper proofs in the area [Abadi and Plotkin 2012, Abadi et al. 2002, Ahmed 2015, Ahmed and Blume 2008, 2011, Fournet et al. 2013, Jagadeesan et al. 2011, Jeffrey and Rathke 2005a, Juglaret et al. 2016, New et al. 2016, Patrignani and Clarke 2015, Patrignani et al. 2015, 2016]. Indeed, we are aware of only one fully mechanized proof about secure compilation: Devriese et al.’s [2017] recent full abstraction result for a translation from the simply typed to the untyped λ -calculus in around 11KLOC of Coq.

Our Coq development comprises around 22KLOC, with proofs taking about 60%. Much of the code is devoted to generic models for components, traces, memory, and undefined behavior that we expect to be useful in proofs for more complex languages and compilers, such as CompCert. We discuss some of the most interesting aspects of the proof below.

Back-translation function We proved Assumption 1 by defining a \uparrow function that takes a finite trace prefix m and a program interface I and returns a whole source program that respects I and produces m . Each generated component uses the local variable `local[0]` to

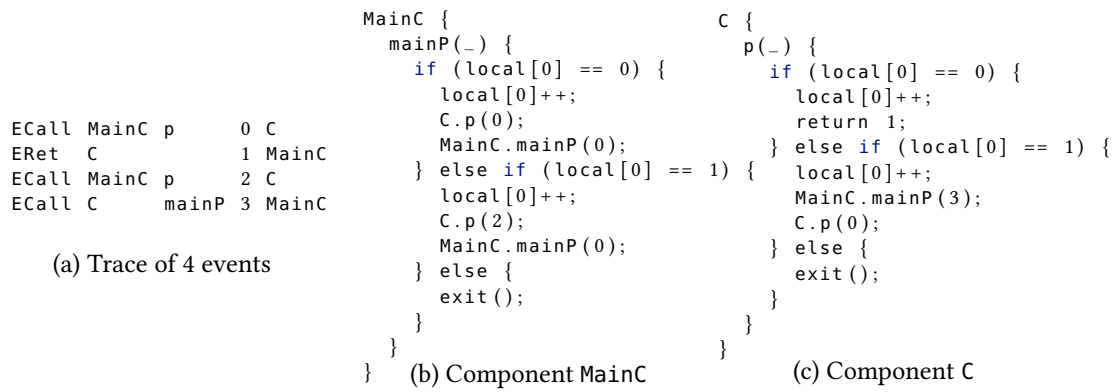


Figure 3.9: Example of program with two components back-translated from a trace of 5 events.

track how many events it has emitted. When a procedure is invoked, it increments `local[0]` and produces the event in m whose position is given by the counter's value. For this back-translation to work correctly, m is restricted to look like a trace emitted by a real compiled program with an I interface—in particular, every return in the trace must match a previous call.

This back-translation is illustrated in Figure 3.9 on a trace of four events. The generated program starts running `MainC.mainP`, with all counters set to 0, so after testing the value of `MainC.local[0]`, the program runs the first branch of `mainP`:

```
local[0]++; C.p(0); MainC.mainP(0);
```

After bumping `local[0]`, `mainP` emits its first event in the trace: the call `C.p(0)`. When that procedure starts running, `C`'s counter is still set to 0, so it executes the first branch of procedure `p`:

```
local[0]++; return 1;
```

The return is `C`'s first event in the trace, and the second of the program. When `mainP` regains control, it calls itself recursively to emit the other events in the trace (we can use tail recursion to iterate in the standard way, since internal calls are silent events). The program continues executing in this fashion until it has emitted all events in the trace, at which point it terminates execution.

Theorem 3.4.1 (Back-translation). The back-translation function \uparrow illustrated above satisfies Assumption 1.

Partial semantics Our partial semantics has a simple generic definition based on the small-step operational semantics of a *whole* target program, which we denote as $\xrightarrow{\alpha}$. In this semantics, each step is labeled with an action α that is either an event or a silent action τ . The definition of the partial semantics $\xrightarrow{\alpha}$ uses a *partialization* function `par` that, given a complete state cs

and the interface I_C of a program part C , returns a partial state ps where all information about C (such as its memory and stack frames) is erased.

$$\frac{\text{par}(cs, I_C) = ps \quad \text{par}(cs', I_C) = ps' \quad cs \xrightarrow{\alpha} cs'}{ps \xrightarrow{\alpha} ps'}$$

The partial semantics can step with action α from the partial state ps to ps' , if there exists a corresponding transition in the complete semantics whose states partialize to ps and ps' . We denote with $P \rightsquigarrow_{I_C}^* m$ that the partial program P produces the trace prefix m in the partial semantics after a finite execution prefix, with respect to the context interface I_C .

A consequence of abstracting away part of the program as non-deterministic actions allowed by its interface is that the abstracted part will always have actions it can do and it will never be stuck, whereas stuckness is the standard way of modeling undefined behavior [Leroy 2009a]. Given $P \rightsquigarrow_{I_C}^* m$, if m ends with an undefined behavior, then this was necessarily caused by P , which is still a concrete partial program running actual code, potentially unsafe.

Our partial semantics was partially inspired by so-called “trace semantics” [Jeffrey and Rathke 2005a, Juglaret et al. 2016, Patrignani and Clarke 2015], where a partial program of interest is decoupled from its context, of which only the observable behavior is relevant. One important difference is that our definition of partial semantics in terms of a partialization function is generic and can be easily instantiated for different languages. On the contrary previous works defined “trace semantics” as separate relations with many rules, making the proofs to correlate partial and complete semantics more involved. Moreover, by focusing on trace properties (instead of observational equivalence) composition and decomposition can be proved using standard simulations à la CompCert, which is easier than previous proof techniques for fully abstract “trace semantics.”

Theorem 3.4.2 (Partial Semantics). The source language and compartmentalized machine partial semantics defined as described above provide decomposition and composition (Assumptions 3 and 4).

Blame We prove Assumption 6 by noting that the behavior of the context C_S can only depend on its own state and on the events emitted by the program. A bit more formally, suppose that the states cs_1 and cs_2 have the same context state, which, borrowing the partialization notation from above, we write as $\text{par}(cs_1, I_P) = \text{par}(cs_2, I_P)$. Then:

- If $cs_1 \xrightarrow{\alpha_1} cs'_1$, $cs_2 \xrightarrow{\alpha_2} cs'_2$, and C_S has control in cs_1 and cs_2 , then $\alpha_1 = \alpha_2$ and $\text{par}(cs'_1, I_P) = \text{par}(cs'_2, I_P)$.
- If $cs_1 \xrightarrow{\tau} cs'_1$ and the program has control in cs_1 and cs_2 , then $\text{par}(cs'_1, I_P) = \text{par}(cs_2, I_P)$.
- If $cs_1 \xrightarrow{\alpha} cs'_1$, the program has control in cs_1 and cs_2 , and $\alpha \neq \tau$, then there exists cs'_2 such that $cs_2 \xrightarrow{\alpha} cs'_2$ and $\text{par}(cs'_1, I_P) = \text{par}(cs'_2, I_P)$.

By repeatedly applying these properties, we can analyze the behavior of two parallel executions $(C_S \cup P') \rightsquigarrow^* m$ and $(C_S \cup P) \rightsquigarrow t$, with $t \prec m$. By unfolding the definition of $t \prec m$ we get that $\exists m' \leq m. t = m' \cdot \text{Undef}(_)$. It suffices to show that $m \leq t \vee t = m' \cdot \text{Undef}(P)$. If $m = t = m' \cdot \text{Undef}(_)$, we have $m \leq t$, and we are done. Otherwise, the execution of $C_S \cup P$ ended earlier because of undefined behavior. After producing prefix m' , $C_S \cup P'$ and $C_S \cup P$ will end up in matching states cs_1 and cs_2 . Aiming for a contradiction, suppose that undefined behavior was caused by C_S . By the last property above, we could find a matching execution step for $C_S \cup P$ that produces the first event in m that is outside of m' ; therefore, $C_S \cup P$ cannot be stuck at cs_2 . Hence $t \prec_P m$.

Theorem 3.4.3 (Blame). Assumption 6 is satisfied.

Theorem 3.4.4 (RSCC). The compilation chain described so far in this section satisfies RSCC.

3.4.4 Software Fault Isolation Back End

The SFI back end uses a special memory layout and code instrumentation sequences to realize the desired isolation of components in the produced program. The target of the SFI back end is a bare-metal RISC processor with the same instructions as the compartmentalization machine minus `Call`, `Return`, and `Alloc`. The register file contains all the registers from the previous level, plus seven additional registers reserved for the SFI instrumentation.

The SFI back end maintains the following invariants: (1) a component may not write outside its own data memory; (2) a component may transfer control outside its own code memory only to entry points allowed by the interfaces or to the return address on top of the global stack; and (3) the global stack remains well formed.

Figure 3.10 shows the memory layout of an application with three components. The entire address space is divided in contiguous regions of equal size, which we will call slots. Each slot is assigned to a component or reserved for the use of the protection machinery. Data and code are kept in disjoint memory regions and memory writes are permitted only in data regions.

An example of a logical split of a physical address is shown in Figure 3.11. A logical address is a triple: offset in a slot, component identifier, and slot identifier unique per component. The slot size, as well as the maximum number of the components are constant for an application, and in Figures 3.10 and 3.11 we have 3 components and slots of size 2^{12} bits.

The SFI back end protects memory regions with instrumentation in the style of Wahbe et al. [1993], but adapted to our component model. Each memory update is preceded by two instructions that set the component identifier to the current one, to prevent accidental or malicious writes in a different component. The instrumentation of the Jump instruction is similar. The last four bits of the offset are always zeroed and all valid targets are sixteen-word-aligned by our back end [Morrissett et al. 2012]. This mechanism, along with careful layout of instructions, ensure that the execution of instrumentation sequences always starts from the first instruction and continues until the end.

Reserved (Code)	Component Code			Protected Stack	Component Data		
	1	2	3		1	2	3
Init Code	Slot 0	Slot 0	Slot 0	Slot 1	Slot 1	Slot 1	Slot 1
Unused	Slot 2	Slot 2	Slot 2	Slot 3	Slot 3	Slot 3	Slot 3
Unused	Slot 4	Slot 4	Slot 4	Slot 5	Slot 5	Slot 5	Slot 5
...

Figure 3.10: Memory layout of three user components

Slot (Unbounded)	Component Identifier (2 bits)	Offset (12 bits)
------------------	-------------------------------	------------------

Figure 3.11: Address Example

The global stack is implemented as a shadow stack [Szekeres et al. 2013] in memory accessible only from the SFI instrumentation sequences. Alignment of code [Morrisett et al. 2012] prevents corruption of the cross-component stack with prepared addresses and ROP attacks, since it is impossible to bypass the instructions in the instrumentation sequence that store the correct address in the appropriate register.

The `Call` instruction of the compartmentalized machine is translated to a `Jal` (jump and link) followed by a sequence of instructions that push the return address on the stack and then restore the values of the reserved registers for the callee component. To protect from malicious pushes that could try to use a forged address, this sequence starts with a `Halt` at an aligned address. Any indirect jump from the current component, will be aligned and will execute the `Halt`, instead of corrupting the cross-component stack. A call from a different component, will execute a direct jump, which is not subject to masking operations and can thus target an unaligned address (we check statically that it is a valid entry point). This `Halt` and the instructions that push on the stack are contained in the sixteen-unit block.

The `Return` instruction is translated to an aligned sequence: `pop` from the protected stack and jump to the retrieved address. This sequence also fits entirely in a sixteen-unit block. The protection of the addresses on the stack itself is realized by the instrumentation of all the `Store` and `Jump` instructions in the program.

We used the QuickChick property-based testing tool [Paraskevopoulou et al. 2015] for Coq to test the three compartmentalization invariants described at the beginning of the subsection. For each invariant, we implemented a test that executes the following steps: (i) randomly generates a valid compartmentalized machine program; (ii) compiles it; (iii) executes the resulting target code in a simulator and records a property-specific trace; and (iv) analyzes the trace to verify if the property has been violated. We also manually injected faults in the compiler by mutating the instrumentation sequences of the generated output and made sure that the tests can detect these injected errors.

More importantly, we also tested two variants of the RSC_{MD}^{DC} property, which consider different

parts of a whole program as the adversarial context. Due to the strict memory layout and the requirement that all components are instrumented, the SFI back end cannot to link with arbitrary target code, and has instead to compile a whole compartmentalized machine program. In a first test, we (1) generate a whole compartmentalized machine program P ; (2) compile P ; (3) run a target interpreter to obtain trace t_t ; (4) if the trace is empty, discard the test; (5) for each component C_T in the trace t_t (5-1) use back-translation to replace, in the program P , the component C_T with a component C_S without undefined behavior (5-2) run the new program on the compartmentalized machine and obtain a trace t_s (5-3) if the condition $t_t \leq t_s$ or $t_s \prec_{P \setminus \cup C_S} t_t$ is satisfied then the test passes, otherwise it fails. Instead of performing step (5), our second test replaces in one go all the components exhibiting undefined behavior, obtaining a compartmentalized machine program that should not have any undefined behavior.

3.4.5 Micro-policies Tagged Architecture

Our second back end is a novel application of a programmable tagged architecture that allows reference monitors, called *micro-policies*, to be defined in software but accelerated by hardware for performance [Azevedo de Amorim et al. 2015, Dhawan et al. 2015b]. On a micro-policy machine, each word in memory or registers carries a metadata tag large enough to hold a pointer to an arbitrary data structure in memory. As each instruction is dispatched by the processor, the opcode of the instruction as well as the tags on the instruction, its argument registers or memory cells, and the program counter are all passed to a software monitor that decides whether to allow the instruction and, if so, produces tags for the results. The positive decisions of this monitor are cached in hardware, so that, if another instruction is executed in the near future with similarly tagged arguments, the hardware can allow the request immediately, bypassing the software monitor.

This enforcement mechanism has been shown flexible enough to implement a broad range of tag-based reference monitors, and for many of them it has a relatively modest impact on runtime (typically under 10%) and power ceiling (less than 10%), in return for some increase in energy (typically under 60%) and chip area (110%) [Dhawan et al. 2015b]. Moreover, the mechanism is simple enough that we could formally verify in Coq that micro-policies for heap memory safety, compartment isolation, control-flow integrity, information-flow control, and dynamic sealing are correct and provide the expected security guarantees [Azevedo de Amorim 2017, Azevedo de Amorim et al. 2014, 2015, 2018]. The rest of this subsection introduces the micro-policies framework using heap memory safety as an illustrative example, while the next subsection (§3.4.6) presents our micro-policy for compartmentalization.

Our micro-policy for heap memory safety [Azevedo de Amorim et al. 2015, Dhawan et al. 2015a] enforces safe access to heap-allocated data, by preventing both spatial violations (e.g., accessing an array out of its bounds) and temporal violations (e.g., referencing through a pointer after the region has been freed). As explained above, such violations are a common source of serious security vulnerabilities. Moreover, with our micro-policy, pointers to the heap become unforgeable capabilities: one can only obtain a valid pointer to a heap region by allocating that region or by copying or offsetting an existing pointer to that region. To achieve this we

tag words representing pointers differently from non-pointers. We use tags to color each heap region differently and to record for each pointer the color of the memory region to which it should point. When a pointer is dereferenced we check that its color matches the color of the memory cell to which it points. We allow pointer arithmetic, which does not affect the color of pointers in any way. In particular, pointers can be taken temporarily out of bounds, as long as out-of-bounds pointers are not accessed. Computing an out-of-bounds pointer is not a violation *per se*—indeed, it happens quite often in practice, e.g., at the end of loops.

More precisely, we use different sets of tags for registers (denoted t_v) and memory (t_m). Values in registers are either pointers tagged with a color c or non-pointers tagged \perp . Allocated memory locations are tagged with a pair (c, t_v) , where c is the color of the encompassing region and t_v is the tag of the stored value. Unallocated memory is tagged with the special tag F (free). Programs can directly interact with the monitor by calling (privileged) *monitor services*; for this policy there are only 2 such services: `malloc` and `free`. The `malloc` service first allocates a region as usual, then generates a fresh color c (e.g., by incrementing a counter), initializes the new heap region with $0@c, \perp$ (i.e., the integer 0 tagged with memory tag (c, \perp)), and returns $w@c$, where w is the start address of the region. The `free` service makes sure that the region is currently allocated and tags the whole deallocated region with F . The F tags prevent any remaining pointers to the deallocated region from being used to access it. If a later allocation reuses the same memory, it will be tagged with a different color, so these dangling pointers will still be unusable.

Outside of monitor services, all the propagation and checking of tags is performed using *rules*. While the hardware uses a cache of low-level rules, these can be automatically obtained from a domain-specific language (DSL) of *symbolic rules*. Together with the representation of tags as algebraic datatypes the symbolic rules provide a convenient language for designing micro-policies. Symbolic rules have the form:

$$opcode : \{PC=t_{pc}, CI=t_{ci}, OP_1=t_1, OP_2=t_2, OP_3=t_3\} \rightarrow \{PC'=t_{pc'}, RES=t_r\}$$

which says that the rule matches on a particular instruction *opcode* together with the tags on the program counter (PC), the current instruction (CI), and up to two three operands from registers or memory (OP_1, OP_2, OP_3). If the rule matches, the right-hand side determines how to update the tags on the program counter (PC') and on the result of the operation (RES). The t metavariables above range over symbolic expressions, including variables. We freely omit input fields that are ignored. Returning to our heap memory safety policy, here is the symbolic rule for adding an integer to a pointer:

$$\text{Add} : \{PC=c_{pc}, CI=(c_{pc}, \perp), OP_1=c, OP_2=\perp\} \rightarrow \{PC'=c_{pc}, RES=c\}$$

This rule says that when the current instruction is `Add`, the first operand is a pointer tagged with color c , and the second operand is an integer tagged \perp the result is again tagged c . The color of the PC, c_{pc} , is left unchanged and we additionally require that c_{pc} matches the color of the region from which the instruction was fetched. This ensures that the PC cannot be used to fetch instructions from inaccessible regions. Similarly, the rules for `Load` and `Store` check that the pointer and the referenced location have the same color c . We use descriptive tag names like P (pointer), M (memory), S (source), D (destination), instead of OP_1, OP_2 , and RES .

$$\begin{aligned}
\text{Load : } & \{PC=c_{pc}, CI=(c_{pc}, \perp), P=\mathbf{c}, M=(\mathbf{c}, t_v)\} \rightarrow \{PC'=c_{pc}, D=t_v\} \\
\text{Store : } & \{PC=c_{pc}, CI=(c_{pc}, \perp), P=\mathbf{c}, M=(\mathbf{c}, t'_v), S=t_v\} \rightarrow \{PC'=c_{pc}, M=(\mathbf{c}, t_v)\}
\end{aligned}$$

For Load the tag of the destination register, t_v , is taken from the tag (\mathbf{c}, t_v) of the loaded memory location. For Store the tag of the written memory location is changed from (\mathbf{c}, t'_v) to (\mathbf{c}, t_v) , where t_v is the tag of the word being written.

3.4.6 Tag-based Reference Monitor

The micro-policy machine targeted by our compartmentalizing back end builds on a “symbolic machine” that Azevedo de Amorim et al. [2017, 2015, 2018] used to prove the correctness and security of several micro-policies in Coq. The code generation and static linking parts of the micro-policy back end are much simpler than for the SFI one. The Call and Return instructions are mapped to Jal and Jump. The Alloc instruction is mapped to a monitor service that tags the allocated memory according to the calling component.

A more interesting aspect of this back end is the way memory must be tagged by the (static) loader based on metadata from previous compilation stages. Memory tags are tuples of the form $t_m ::= (t_v, c, cs)$. The tag t_v is for the payload value. The component identifier c , which we call a color, establishes the component that owns the memory location. Our monitor forbids any attempt to write to memory if the color of the current instruction is different from the color of the target location. The set of colors cs identifies all the components that are allowed to call to this location and is by default empty. The value tags used by our monitor distinguish cross-component return addresses from all other words in the system: $t_v ::= \text{Ret}(n) \mid \perp$. To enforce the cross-component stack discipline return addresses are treated as *linear return capabilities*, i.e., unique capabilities that cannot be duplicated [Knight et al. 2012] and that can only be used to return once. This is achieved by giving return addresses tags of the form $\text{Ret}(n)$, where the natural number n represents the stack level to which this capability can return. We keep track of the current stack level using the tag of the program counter: $t_{pc} ::= \text{Level}(n)$. Calls increment the counter n , while returns decrement it. A global invariant is that when the stack is at $\text{Level}(n)$ there is at most one capability $\text{Ret}(m)$ for any level m from 0 up to $n-1$.

Our tag-based reference monitor for compartmentalization is simple; the complete definition is given in Figure 3.12. For Mov, Store, and Load the monitor *copies* the tags together with the values, but for return addresses the linear capability tag $\text{Ret}(n)$ is *moved* from the source to the destination. Loads from other components are allowed but prevented from stealing return capabilities. Store operations are only allowed if the color of the changed location matches the one of the currently executing instruction. Bnz is restricted to the current component. Jal to a different component is only allowed if the color of the current component is included in the allowed entry points; in this case and if we are at some $\text{Level}(n)$ the machine puts the return address in register RA and the monitor gives it tag $\text{Ret}(n)$ and it increments the pc tag to $\text{Level}(n+1)$. Jump is allowed either to the current component or using a $\text{Ret}(n)$ capability,

Nop	$\{PC=t_{pc}, CI=(_, c, _), NI=(_, c, _)\}$	$\rightarrow \{PC'=t_{pc}\}$
Const	$\{PC=t_{pc}, CI=(_, c, _), NI=(_, c, _)\}$	$\rightarrow \{PC'=t_{pc}, D=\perp\}$
BinOp	$\{PC=t_{pc}, CI=(_, c, _), NI=(_, c, _), S_1=_, S_2=_\}$	$\rightarrow \{PC'=t_{pc}, D=\perp\}$
Mov	$\{PC=t_{pc}, CI=(_, c, _), NI=(_, c, _), S=t_v\}$	$\rightarrow \{PC'=t_{pc}, S=\perp, D=t_v\}$
Load	$\{PC=t_{pc}, CI=(_, c, _), NI=(_, c, _), M=(t_v, c, _)\}$	$\rightarrow \{PC'=t_{pc}, M=(\perp, c, _), D=t_v\}$
Load	$\{PC=t_{pc}, CI=(_, c, _), NI=(_, c, _), M=(t_v, c', _)\}$	$\rightarrow \{PC'=t_{pc}, M=(t_v, c', _), D=\perp\}$
Store	$\{PC=t_{pc}, CI=(_, c, _), NI=(_, c, _), M=(_, c, _), S=t_v\}$	$\rightarrow \{PC'=t_{pc}, M=(t_v, c, _), S=\perp\}$
Bnz	$\{PC=t_{pc}, CI=(_, c, _), NI=(_, c, _)\}$	$\rightarrow \{PC'=t_{pc}\}$
Jal	$\{PC=t_{pc}, CI=(_, c, _), NI=(_, c, _), P=_\}$	$\rightarrow \{PC'=t_{pc}, RA=\perp\}$
Jal	$\{PC=Level(n), CI=(_, c, _), NI=(_, c', cs \ni c), P=_\}$	$\rightarrow \{PC'=Level(n+1), RA=Ret(n)\}$
Jump	$\{PC=t_{pc}, CI=(_, c, _), NI=(_, c, _), P=_\}$	$\rightarrow \{PC'=t_{pc}, RA=\perp\}$
Jump	$\{PC=Level(n+1), CI=(_, c, _), NI=(_, c', _), P=Ret(n)\}$	$\rightarrow \{PC'=Level(n), P=\perp\}$

Figure 3.12: Compartmentalization micro-policy rules

but only if we are at $Level(n+1)$; in this case the pc tag is decremented to $Level(n)$ and the $Ret(n)$ capability is destroyed. Instruction fetches are also checked to ensure that one cannot switch components by continuing to execute past the end of a code region. To make these checks as well as the ones for Jal convenient we use the next instruction tag NI directly; in reality one can encode these checks even without NI by using the program counter and current instruction tags [Azevedo de Amorim et al. 2015]. The bigger change compared to the micro-policy mechanism of Azevedo de Amorim et al. [2015] is our overwriting of input tags in order to invalidate linear capabilities in the rules for Mov, Load, and Store. For cases in which supporting this in hardware is not feasible we have also devised a compartmentalization micro-policy that does not rely on linear return capabilities but on linear entry points.

A variant of the compartmentalization micro-policy above was first studied by Juglaret et al. [2015], in an unpublished technical report. Azevedo de Amorim et al. [2015] also devised a micro-policy for compartmentalization, based on a rather different component model. The biggest distinction to Azevedo de Amorim et al.’s work is that our micro-policy enforces the stack discipline on cross-component calls and returns.

3.5 Related Work

Fully Abstract Compilation, originally introduced in seminal work by Abadi [1999], is phrased in terms of protecting two partial program variants written in a *safe* source language, when these are compiled and linked with a malicious target-level context that tries to distinguish the two variants. This original attacker model differs substantially from the one we consider in this work, which protects the trace properties of multiple mutually-distrustful components written in an *unsafe* source language.

In this line of research, Abadi [1999] and later Kennedy [2006] identified failures of full abstraction in the Java and C# compilers. Abadi et al. [2002] proved full abstraction of secure channel implementations using cryptography. Ahmed et al. [Ahmed 2015, Ahmed and Blume 2008, 2011, New et al. 2016] proved the full abstraction of type-preserving compiler passes for functional languages. Abadi and Plotkin [2012] and Jagadeesan et al. [2011] expressed the protection provided by address space layout randomization as a probabilistic variant of full abstraction. Fournet et al. [2013] devised a fully abstract compiler from a subset of ML to JavaScript. More recently, Patrignani et al. [Larmuseau et al. 2015, Patrignani et al. 2015] studied fully abstract compilation to machine code, starting from single modules written in simple, idealized object-oriented and functional languages and targeting a hardware enclave mechanism similar to SGX [Intel].

Modular, Fully Abstract Compilation. Patrignani et al. [2016] subsequently proposed a “modular” extension of their compilation scheme to protecting multiple components from each other. The attacker model they consider is again different from ours: they focus on separate compilation of safe languages and aim to protect linked target-level components that are observationally equivalent to compiled components. This could be useful, for example, when hand-optimizing assembly produced by a secure compiler. In another thread of work, Devriese et al. [2017] proved modular full abstraction by approximate back-translation for a compiler from simply typed to untyped λ -calculus. This work also introduces a complete Coq formalization for the original (non-modular) full abstraction proof of Devriese et al. [2016a].

Beyond Good and Evil. The closest related work is that of Juglaret et al. [2016], who also aim at protecting mutually distrustful components written in an unsafe language. They adapt fully abstract compilation to components, but observe that defining observational equivalence for programs with undefined behavior is highly problematic. For instance, is the partial program “`int buf[5]; return buf[42]`” equivalent to “`int buf[5]; return buf[43]`”? Both encounter undefined behavior by accessing a buffer out of bounds, so at the source level they cannot be distinguished. However, in an unsafe language, the compiled versions of these programs will likely read (out of bounds) different values and behave differently. Juglaret et al. avoid this problem by imposing a strong limitation: a set of components is protected only if it cannot encounter undefined behavior in *any* context. This amounts to a *static* model of compromise: all components that can possibly be compromised during execution have to be treated as compromised from the start. Our aim here is to show that, by moving away from full abstraction and by restricting the temporal scope of undefined behavior, we can support a more realistic *dynamic* compromise model. As discussed below, moving away from full abstraction also makes our secure compilation criterion easier to achieve in practice and to prove at scale.

Robust Safety Property Preservation. Our criterion builds on the *RSC* criterion proposed in §2.2.3, where we studied several secure compilation criteria that are similar to fully abstract

compilation, but that are phrased in terms of preserving hyperproperties [Clarkson and Schneider 2010] (rather than observational equivalence) against an adversarial context. In particular, *RSC* is equivalent to preservation of *robust safety*, which has been previously employed for the model checking of open systems [Kupferman and Vardi 1999], the analysis of security protocols [Gordon and Jeffrey 2004], and compositional verification [Swasey et al. 2017].

Though *RSC* is a bit less extensional than fully abstract compilation (since it is stated in terms of execution traces), it is easier to achieve. In particular, because it focuses on safety instead of confidentiality, the code and data of the protected program do not have to be hidden, allowing for more efficient enforcement, e.g., there is no need for fixed padding to hide component sizes, no cleaning of registers when passing control to the context (unless they store capabilities), and no indirection via integer handlers to hide pointers; cross-component reads can be allowed and can be used for passing large data. We believe that in the future we can obtain a more practical notion of data (but not code) confidentiality by adopting the hypersafety preservation criterion of §2.3.3.

While *RSC* serves as a solid base for our work, the challenges of protecting unsafe components from each other are unique to the setting of this chapter, since, like full abstraction, *RSC* is about protecting a partial program written in a *safe* source language against low-level contexts. Our contribution is extending *RSC* to reason about the dynamic compromise of components with undefined behavior, taking advantage of the execution traces to detect the compromise of components and to rewind the execution along the same trace.

Proof Techniques. In §2.2.3 we observe that, to prove *RSC*, it suffices to back-translate finite execution prefixes, and in §2.3.3 we propose such a proof for a stronger criterion where multiple such executions are involved. In recent concurrent work, Patrignani and Garg [2018] also construct such a proof for *RSC*. The main advantages of the RSC_{MD}^{DC} proof from this chapter are that (1) it applies to unsafe languages with undefined behavior and (2) it directly reuses a compiler correctness result à la CompCert. For safe source languages or when proof reuse is not needed our proof could be further simplified.

Even as it stands though, our proof technique is simple and scalable compared to previous full abstraction proofs. While many proof techniques have been previously investigated [Abadi and Plotkin 2012, Abadi et al. 2002, Ahmed and Blume 2008, 2011, Devriese et al. 2017, Fournet et al. 2013, Jagadeesan et al. 2011, New et al. 2016], fully abstract compilation proofs are notoriously difficult, even for very simple languages, with apparently simple conjectures surviving for decades before being finally settled [Devriese et al. 2018]. The proofs of Juglaret et al. [2016] are no exception: while their compiler is similar to the one in §3.4, their full abstraction-based proof is significantly more complex than our RSC_{MD}^{DC} proof. Both proofs give semantics to partial programs in terms of traces, as was proposed by Jeffrey and Rathke [2005a] and adapted to low-level target languages by Patrignani and Clarke [2015]. However, in our setting the partial semantics is given a one line generic definition and is related to the complete one by two simulation proofs, which is simpler than proving a “trace semantics” fully abstract.

Verifying Low-Level Compartmentalization. Recent successes in formal verification have focused on showing correctness of low-level compartmentalization mechanisms based on software fault isolation [Morrisett et al. 2012, Zhao et al. 2011] or tagged hardware [Azevedo de Amorim et al. 2015]. That work only considers the correctness of low-level mechanisms in isolation, not how a secure compilation chain makes use of these mechanisms to provide security reasoning principles for code written in a higher-level programming language with components. However, more work in this direction seems underway, with Wilke et al. [2017] working on a variant of CompCert with SFI, based on previous work by Kroll et al. [2014]; we believe *RSCC* or *RSC^{DC}* could provide good top-level theorems for such an SFI compiler. In most work on verified compartmentalization [Azevedo de Amorim et al. 2015, Morrisett et al. 2012, Zhao et al. 2011], communication between low-level compartments is done by jumping to a specified set of entry points; the model considered here is more structured and enforces the correct return discipline. Skorstengaard et al. [2018a] have also recently investigated a secure stack-based calling convention for a simple capability machine; they plan to simplify their calling convention using a notion of linear return capability [2018b] that seems similar to the one used in our micro-policy from §3.4.6.

Attacker Models for Dynamic Compromise. While our model of dynamic compromise is specific to secure compilation of unsafe languages, related notions of compromise have been studied in the setting of cryptographic protocols, where, for instance, a participant’s secret keys could inadvertently be leaked to a malicious adversary, who could then use them to impersonate the victim [Backes et al. 2009, Basin and Cremers 2014, Fournet et al. 2007, Gordon and Jeffrey 2005]. This model is also similar to Byzantine behavior in distributed systems [Castro and Liskov 2002, Lamport et al. 1982], in which the “Byzantine failure” of a node can cause it to start behaving in an arbitrary way, including generating arbitrary data, sending conflicting information to different parts of the system, and pretending to be a correct node.

3.6 Conclusion

We introduced *RSCC*, a new formal criterion for secure compilation providing strong security guarantees despite the dynamic compromise of components with undefined behavior. This criterion gives a precise meaning to informal terms like *dynamic compromise* and *mutual distrust* used by proponents of compartmentalization, and it offers a solid foundation for reasoning about security of practical compartmentalized applications and secure compiler chains.

4 Research Plan for the Next 4 Years

The goal of the ERC SECOMP project is to build the first formally secure compartmentalizing compilation chains for realistic programming languages. In particular, we are planning a secure compilation chain starting from programs written in a combination of C and Low* [Protzenko et al. 2017] and targetting a RISC-V architecture extended with micro-policies [Azevedo de Amorim et al. 2015]. In order to ensure high confidence in the security of our compilation chains, we plan to thoroughly test them using property-based testing and eventually formally verify their security using Coq. For measuring and optimizing efficiency we plan to use standard benchmark suites [Henning 2006] and realistic source programs, with miTLS* as the main end-to-end case study. Achieving all this requires overcoming several major conceptual and technological challenges, which constitute the main scientific objectives of this project.

Further Studying Secure Compilation Criteria As illustrated by this thesis, so far most the work on this project has been focused on devising secure compilation criteria based on preserving classes of properties against adversarial contexts [Abate et al. 2018b, Juglaret et al. 2016] and extending these criteria to unsafe languages [Abate et al. 2018a]. Various interesting open problems remain in this space, including fully working out the connection between our secure compilation criteria and fully abstract compilation (see §2.8). Even closer related to our long term goals is extending the component model from chapter 3 with *dynamic component creation*. This would make crucial use of our dynamic compromise model, since components would no longer be statically known, and thus static compromise would not apply at all. We hope that our *RSCC* definition can be adapted to rewind execution to the point at which the compromised component was created, replace the component’s code with the result of our back-translation, and then re-execute. This extension could allow us to move from our current “code-based” compartmentalization model to a “data-based” one [Gudka et al. 2015], e.g., one compartment per incoming network connection.

Formally Secure Compartmentalization for C. We plan to devise a compartmentalizing compilation chain based on the CompCert C compiler and targetting a RISC-V architecture. Scaling up to the whole of C and RISC-V will certainly entail challenges such as defining a variant of C with components and efficiently enforcing compartmentalization all the way down using micro-policies. Targetting a realistic language like C is, however, the only way we can measure and optimize the efficiency of our compilation chain on standard benchmark suites [Henning 2006] and realistic source programs, such as miTLS*. To achieve this, we will build on the solid basis built by this work: the *RSCC* formal security criterion, the scalable proof technique, and the proof-of-concept secure compilation chain from §3.4.

As an interesting first step, we plan to extend our simple compilation chain to allow sharing memory between components. Since we already allow arbitrary reads at the lowest level, it seems appealing to also allow external reads from some of the components' memory in the source. The simplest would be to allow certain static buffers to be shared with all other components, or only with some if we also extend the interfaces. For this extension we need to set the shared static buffers to the right values every time a back-translated component gives up control; for this back-translation needs to look at the read events forward in the back-translated trace prefix. More ambitious would be to allow pointers to dynamically allocated memory to be passed to other components, as a form of read capabilities. This would make pointers appear in the traces and one would need to accommodate the fact that these pointers will vary at the different levels in our compilation chain. Moreover, each component produced by the back-translation would need to record all the read capabilities it receives for later use. Finally, to safely allow write capabilities we could combine compartmentalization with memory safety.

Dynamic Privilege Notions The compilation chain from §3.4 used a very simple notion of interface to statically restrict the privileges of components. This could, however, be extended with dynamic notions of privilege such as capabilities and history-based access control [Abadi and Fournet 2003]. In one of its simplest form, allowing pointers to be passed between components and then used to write data, as discussed above, would already constitute a dynamic notion of privilege, that is not captured by the static interfaces, but nevertheless needs to be enforced to achieve *RSCC*, in this case using some form of memory safety.

Memory Safety for C We also plan to enforce memory safety for C and its interactions with untrusted RISC-V assembly. This will protect buggy programs from malformed inputs that would normally trigger a memory safety violation. Enforcing memory safety requires changes to the C compiler and a sophisticated micro-policy, which extends our simple heap memory safety policy [Azevedo de Amorim 2017, Azevedo de Amorim et al. 2015, Dhawan et al. 2015a] to additionally deal with unboxed structs, stack allocation [Roessler and DeHon 2018], byte addressing, unaligned memory accesses, custom allocators, etc. We plan to build an extension of CompCert [Leroy 2009b] that is memory safe. To verify security we will target both properties describing the absence of spatial (e.g., buffer overflows) and temporal (e.g., use after free, double free) memory safety violations [Nagarakatte et al. 2010] and our higher-level reasoning principles enabled by memory safety [Azevedo de Amorim et al. 2018].

In a follow up step we will obtain stronger properties by combining memory safety and compartmentalization. This will give us a fine-grained object-capability model [Watson et al. 2015a] on a fully generic tagged architecture and will enable compartmentalized applications that are much more granular and thus more secure than using currently-deployed isolation techniques [Gudka et al. 2015, Reis and Gribble 2009, Yee et al. 2010].

Verifying Compartmentalized Applications. It would also be interesting to build verification tools based on the source reasoning principles provided by *RSCC* and to use these tools to analyze the security of practical compartmentalized applications. Effective verification on top

of *RSCC* will, however, require good ways for reasoning about the exponential number of dynamic compromise scenarios. One idea is to do our source reasoning with respect to a variant of our partial semantics, which would use nondeterminism to capture the compromise of components and their possible successive actions. Correctly designing such a partial semantics for a complex language is, however, challenging. Fortunately, our *RSCC* criterion provides a more basic, low-TCB definition against which to validate any fancier reasoning tools, like partial semantics, program logics [Jia et al. 2015], logical relations [Devriese et al. 2016b], etc.

Secure compilation of Low^* to C using Components, Contracts, and Sealing We also plan to devise a secure compilation chain from Low^* to C. Low^* programs are verified with respect to Hoare-style pre- and post- conditions to achieve correctness and use the F^* module system (i.e., data abstraction and parametricity) to achieve confidentiality of secret data, even against certain side-channels. These high-level abstractions will have to be protected at the C level, and while compartmentalization will offer a first barrier of defense, more work will be needed. We plan to enforce specifications by turning them into dynamic contracts and parametricity by relying on dynamic sealing. We hope that micro-policies can help us implement both contracts and sealing efficiently.

Micro-policies for C Micro-policies operate at the lowest machine-code level. While this is appropriate for devising secure C compilers, we also want our secure Low^* to C compiler to directly make use of micro-policies in order to efficiently enforce the high-level abstractions of Low^* . Moreover, we want a general solution that is not tied to our compilation chain, but instead allows arbitrary programs in C to benefit from efficient programmable tag-based monitoring. Exposing micro-policies in C and then translating them down is challenging, because the structure of programs in these languages is different than that of machine code.

We will extend the semantics of C with support for tag-based reference monitoring. These tag-based monitors—i.e., high-level micro-policies—will be written in rule-based domain-specific languages (DSLs) inspired by our rule format for micro-policies monitoring machine code [Azevedo de Amorim et al. 2014, 2015, Dhawan et al. 2015a]. Some parts of the micro-policy DSLs for C and machine code will be similar: for instance, we want a simple way to define the structure of tags using algebraic datatypes, sets, and maps. The kinds of tags differs from level to level though: at the machine code level we have register, program counter, and memory tags, while in C we could replace register tags with value and procedure tags. The way tags are checked and propagated also differs significantly between levels. At the machine-code level, propagation is done via rules that are invoked on each instruction, while in C we have many different operations that can be monitored, e.g., primitive operations, function calls and returns etc. Moreover, the tags of C values could be propagated automatically as values are copied around, without needing to write explicit rules for that. Finally, we want to automatically translate micro-policies for C to micro-policies for machine-code.

Secure micro-policy composition Our secure compilation chains require composing many different micro-policies. For instance, we need to simultaneously enforce isolation of mutual

distrustful components and memory safety for some of the components. Recent microarchitectural optimizations enable us to efficiently enforce multiple micro-policies simultaneously [Dhawan et al. 2015a], by taking tags to be tuples, where each tag component is handled by a different sub-policy. Yet composing isolation and memory safety is non-trivial, since each of them has its own view on memory, and a naive composition would be dysfunctional, for instance dynamically allocating in the memory of the wrong component. While this problem can be fixed by changing the code of the composed micro-policies, the bigger conceptual difficulty is in composing specifications and security proofs. Secure composition principles are badly needed, since verifying each composed micro-policy from scratch does not scale. Secure composition is, however, very difficult to achieve in our setting, because micro-policies can directly influence the monitored code by answering to direct calls and by raising exceptions, and thus one micro-policy’s observable behavior can break the other micro-policy’s guarantees.

We will study several techniques for composing micro-policy specifications and proofs, with the composite policies needed by our secure compilation chains as the main motivating examples. First, we will investigate layering micro-policies, by choosing an order among them and constructing a sequence of abstract machines, each of which “virtualizes” the tagging mechanisms in the hardware so that further micro-policies can be implemented on top. We will then use ideas from monad transformers and algebraic effects to allow the micro-policies to be verified separately and layered in any order. Finally, we will investigate other forms of composition, for instance, those in which each micro-policy specifies how its tags should be affected by the interactions of the other policies with the monitored code.

Preserving Confidentiality and Hypersafety It would be interesting to extend our *RSCC* security criterion and enforcement mechanisms from robustly preserving safety to confidentiality and hypersafety (§2.3.3). For this we need to control the flow of information at the target level—e.g., by restricting direct reads and read capabilities, cleaning registers, etc. This becomes very challenging though, in a realistic attacker model in which low-level contexts can observe time. While at first we could assume that low-level contexts cannot exploit side-channels, an interesting challenge will be to try to extend our enforcement to also protect against timing side-channels. In this context, we could investigate preserving various K-Safety Hyperproperties such as nonmalleable information flow control [Cecchetti et al. 2017], timing-sensitive noninterference [Rafnsson et al. 2017], and cryptographic “constant time” [Barthe et al. 2018] (i.e. secret independent timing).

Bibliography

- M. Abadi. Protection in programming-language translations. *Secure Internet Programming*. 1999. 13, 14, 19, 42, 45, 48, 77, 78
- M. Abadi and C. Fournet. Access control based on execution history. *NDSS*. The Internet Society, 2003. 82
- M. Abadi and J. Planul. On layout randomization for arrays and functions. *POST*. 2013. 48
- M. Abadi and G. D. Plotkin. On protection by layout randomization. *ACM TISSEC*, 15(2):8, 2012. 13, 14, 42, 43, 48, 69, 78, 79
- M. Abadi, C. Fournet, and G. Gonthier. Secure implementation of channel abstractions. *Information and Computation*, 174(1):37–83, 2002. 13, 14, 42, 43, 69, 78, 79
- M. Abadi, B. Blanchet, and C. Fournet. The applied pi calculus: Mobile values, new names, and secure communication. *J. ACM*, 65(1):1:1–1:41, 2018. 44
- C. Abate, A. Azevedo de Amorim, R. Blanco, A. N. Evans, G. Fachini, C. Hrițcu, T. Laurent, B. C. Pierce, M. Stronati, and A. Tolmach. When good components go bad: Formally secure compilation despite dynamic compromise. *CCS*. 2018a. 7, 12, 14, 19, 45, 81
- C. Abate, R. Blanco, D. Garg, C. Hrițcu, M. Patrignani, and J. Thibault. Journey beyond full abstraction: Exploring robust property preservation for secure compilation. *arXiv:1807.04603*, 2018b. 7, 11, 48, 58, 60, 81
- P. Agten, R. Strackx, B. Jacobs, and F. Piessens. Secure compilation to modern processors. *CSF*. 2012. 48
- P. Agten, B. Jacobs, and F. Piessens. Sound modular verification of C code executing in an unverified context. *POPL*. 2015. 43, 48
- D. Ahman, C. Hrițcu, K. Maillard, G. Martínez, G. Plotkin, J. Protzenko, A. Rastogi, and N. Swamy. Dijkstra monads for free. *POPL*. 2017. 8
- D. Ahman, C. Fournet, C. Hrițcu, K. Maillard, A. Rastogi, and N. Swamy. Recalling a witness: Foundations and applications of monotonic state. *PACMPL*, 2(POPL):65:1–65:30, 2018. 8
- A. Ahmed. Verified compilers for a multi-language world. *SNAPL*. 2015. 14, 19, 42, 69, 78
- A. Ahmed and M. Blume. Typed closure conversion preserves observational equivalence. *ICFP*. 2008. 14, 19, 42, 43, 69, 78, 79

- A. Ahmed and M. Blume. An equivalence-preserving CPS translation via multi-language semantics. *ICFP*. 2011. 14, 19, 42, 43, 69, 78, 79
- B. Alpern and F. B. Schneider. Defining liveness. *IPL*, 21(4):181–185, 1985. 17, 18, 21, 22, 23, 26, 28, 29
- K. Asanović and D. A. Patterson. Instruction sets should be free: The case for RISC-V. Technical Report UCB/EECS-2014-146, EECS Department, University of California, Berkeley, 2014. 12
- A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. *ESORICS*. 2008. 15, 26, 28
- A. Azevedo de Amorim. *A methodology for micro-policies*. PhD thesis, University of Pennsylvania, 2017. 74, 76, 82
- A. Azevedo de Amorim, N. Collins, A. DeHon, D. Demange, C. Hrițcu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach. A verified information-flow architecture. *POPL*. 2014. 6, 7, 74, 83
- A. Azevedo de Amorim, M. Dénès, N. Giannarakis, C. Hrițcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach. Micro-policies: Formally verified, tag-based security monitors. *Oakland S&P*. 2015. 7, 10, 12, 19, 46, 66, 74, 76, 77, 80, 81, 82, 83
- A. Azevedo de Amorim, N. Collins, A. DeHon, D. Demange, C. Hrițcu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach. A verified information-flow architecture. *Journal of Computer Security (JCS); Special Issue on Verified Information Flow Security*, 24(6):689–734, 2016. 6, 7, 12
- A. Azevedo de Amorim, C. Hrițcu, and B. C. Pierce. The meaning of memory safety. In *7th International Conference on Principles of Security and Trust (POST)*, 2018. 47, 74, 76, 82
- M. Backes, C. Hrițcu, and M. Maffei. Type-checking zero-knowledge. *CCS*. 2008. 44
- M. Backes, M. P. Grochulla, C. Hrițcu, and M. Maffei. Achieving security despite compromise using zero-knowledge. *CSF*. 2009. 80
- M. Backes, C. Hrițcu, and M. Maffei. Union and intersection types for secure protocol implementations. *TOSCA (precursor of POST)*. 2011. Invited paper. 44
- D. Baelde, S. Delaune, and L. Hirschi. A reduced semantics for deciding trace equivalence. *LMCS*, 13(2), 2017. 30, 44
- G. Barthe, B. Grégoire, and V. Laporte. Secure compilation of side-channel countermeasures: the case of cryptographic “constant-time”. *CSF*. 2018. 14, 19, 84
- D. A. Basin and C. Cremers. Know your enemy: Compromising adversaries in protocol analysis. *TISSEC*, 17(2):7:1–7:31, 2014. 80

- B. Beurdouche, K. Bhargavan, F. Kiefer, J. Protzenko, E. Rescorla, T. Taubert, M. Thomson, and J.-K. Zinzindohoue. HACL* in Mozilla Firefox: Formal methods and high assurance applications for the web. *Real World Crypto Symposium*, 2018. 13
- K. Bhargavan, B. Bond, A. Delignat-Lavaud, C. Fournet, C. Hawblitzel, C. Hritcu, S. Ishtiaq, M. Kohlweiss, R. Leino, J. Lorch, K. Maillard, J. Pang, B. Parno, J. Protzenko, T. Ramanananandro, A. Rane, A. Rastogi, N. Swamy, L. Thompson, P. Wang, S. Zanella-Béguelin, and J.-K. Zinzindohoue. Everest: Towards a verified, drop-in replacement of HTTPS. *SNAPL*, 2017a. 8
- K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Pan, J. Protzenko, A. Rastogi, N. Swamy, S. Zanella Béguelin, and J. K. Zinzindohoue. Implementing and proving the TLS 1.3 record layer. *IEEE Security & Privacy*, 2017b. 8, 10, 13
- A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting applications into reduced-privilege compartments. *USENIX NSDI*, 2008. 10, 19, 46
- M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *TOCS*, 20(4):398–461, 2002. 80
- E. Cecchetti, A. C. Myers, and O. Arden. Nonmalleable information flow control. *CCS*. 2017. 28, 84
- V. Cheval, V. Cortier, and S. Delaune. Deciding equivalence-based properties using constraint solving. *TCS*, 492:1–39, 2013. 17, 30, 35, 44, 45
- V. Cheval, S. Kremer, and I. Rakotonirina. DEEPSEC: Deciding equivalence properties in security protocols theory and practice. *S&P*. 2018. 30, 44
- D. Chisnall, C. Rothwell, R. N. M. Watson, J. Woodruff, M. Vadera, S. W. Moore, M. Roe, B. Davis, and P. G. Neumann. Beyond the PDP-11: Architectural support for a memory-safe C abstract machine. *ASPLOS*. 2015. 10, 46
- K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ICFP*. 2000. 8
- M. R. Clarkson and F. B. Schneider. Hyperproperties. *JCS*, 18(6):1157–1210, 2010. 14, 15, 17, 21, 26, 28, 29, 43, 79
- M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, M. Negro, C. Liebchen, M. Qunaibit, and A. Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. *CCS*. 2015. 9
- R. De Nicola and M. Hennessy. Testing equivalences for processes. *TCS*, 34:83–133, 1984. 31
- S. Delaune and L. Hirschi. A survey of symbolic methods for establishing equivalence-based properties in cryptographic protocols. *JLAMP*, 87:127–144, 2017. 30, 44

- D. Devriese, M. Patrignani, and F. Piessens. Fully-abstract compilation by approximate back-translation. *POPL*, 2016a. 14, 19, 37, 39, 43, 78
- D. Devriese, F. Piessens, and L. Birkedal. Reasoning about object capabilities with logical relations and effect parametricity. *EuroS&P*. 2016b. 83
- D. Devriese, M. Patrignani, F. Piessens, and S. Keuchel. Modular, fully-abstract compilation by approximate back-translation. *LMCS*, 13(4), 2017. 14, 43, 48, 69, 78, 79
- D. Devriese, M. Patrignani, and F. Piessens. Parametricity versus the universal type. *PACMPL*, 2(POPL):38:1–38:23, 2018. 14, 43, 79
- U. Dhawan and A. DeHon. Area-efficient near-associative memories on FPGAs. *FPGA*, 2013. 6
- U. Dhawan, A. Kwon, E. Kadric, C. Hrițcu, B. C. Pierce, J. M. Smith, G. Malecha, G. Morrisett, T. F. Knight, Jr., A. Sutherland, T. Hawkins, A. Zyxfryx, D. Wittenberg, P. Trei, S. Ray, G. Sullivan, and A. DeHon. Hardware support for safety interlocks and introspection. In *SASO Workshop on Adaptive Host and Network Security*, 2012. 6
- U. Dhawan, C. Hrițcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. Knight, Jr., B. C. Pierce, and A. DeHon. Architectural support for software-defined metadata processing. *ASPLOS*. 2015a. 74, 82, 83, 84
- U. Dhawan, C. Hrițcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. Knight, Jr., B. C. Pierce, and A. DeHon. Architectural support for software-defined metadata processing. *ASPLOS*. 2015b. 7, 66, 74
- V. D’Silva, M. Payer, and D. X. Song. The correctness-security gap in compiler optimization. *S&P Workshops*. 2015. 19
- G. J. Duck and R. H. C. Yap. EffectiveSan: Type and memory error detection using dynamically typed C/C++. *PLDI*, 2018. 9, 47
- Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson. The matter of Heartbleed. *IMC*. 2014. 9, 13
- A. El-Korashy, S. Tsampas, M. Patrignani, D. Devriese, D. Garg, and F. Piessens. Compiling a secure variant of C to capabilities. Dagstuhl Seminar 18201 on Secure Compilation, 2018. 19
- J. Engelfriet. Determinacy implies (observation equivalence = trace equivalence). *TCS*, 36: 21–25, 1985. 17, 30, 33, 35, 45
- A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. *IEEE S&P*, 2019. 13
- I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. To appear at CCS, 2015. 9

- J. S. Fenton. Memoryless subsystems. *The Computer Journal*, 17(2):143–147, 1974. 28
- R. Focardi and R. Gorrieri. A taxonomy of security properties for process algebras. *JCS*, 3(1): 5–34, 1995. 17
- C. Fournet, A. D. Gordon, and S. Maffei. A type discipline for authorization policies. *ACM Trans. Program. Lang. Syst.*, 29(5):25, 2007. 80
- C. Fournet, N. Swamy, J. Chen, P. Dagand, P. Strub, and B. Livshits. Fully abstract compilation to JavaScript. *POPL*. 2013. 14, 42, 43, 48, 69, 78, 79
- J. A. Goguen and J. Meseguer. Security policies and security models. *S&P*, 1982. 15, 26
- A. Gollamudi and C. Fournet. Building secure SGX enclaves using F*, C/C++ and X64. 2nd Workshop on Principles of Secure Compilation (PriSC), 2018. 10, 46
- A. D. Gordon and A. Jeffrey. Typing correspondence assertions for communication protocols. *TCS*, 300(1-3):379–409, 2003. 53
- A. D. Gordon and A. Jeffrey. Types and effects for asymmetric cryptographic protocols. *JCS*, 12(3-4):435–483, 2004. 15, 24, 44, 53, 79
- A. D. Gordon and A. Jeffrey. Secrecy despite compromise: Types, cryptography, and the pi-calculus. *CONCUR*. 2005. 80
- N. Grimm, K. Maillard, C. Fournet, C. Hrițcu, M. Maffei, J. Protzenko, T. Ramananandro, A. Rastogi, N. Swamy, and S. Zanella-Béguelin. A monadic framework for relational verification: Applied to information security, program equivalence, and optimizations. *CPP*. 2018. 8
- K. Gudka, R. N. M. Watson, J. Anderson, D. Chisnall, B. Davis, B. Laurie, I. Marinos, P. G. Neumann, and A. Richardson. Clean application compartmentalization with SOAAP. *CCS*. 2015. 10, 11, 19, 46, 50, 81, 82
- A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. F. Bastien. Bringing the web up to speed with WebAssembly. *PLDI*, 2017. 10, 46
- I. Haller, Y. Jeon, H. Peng, M. Payer, C. Giuffrida, H. Bos, and E. van der Kouwe. TypeSan: Practical type confusion detection. *CCS*, 2016. 9, 47
- C. Hathhorn, C. Ellison, and G. Rosu. Defining the undefinedness of C. *PLDI*. 2015. 9, 47
- Heartbleed. The Heartbleed bug. <http://heartbleed.com/>, 2014. 47
- J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4): 1–17, 2006. 12, 81
- C. Hrițcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett. All your IFCEException are belong to us. *Oakland S&P*. 2013a. 6

- C. Hrițcu, J. Hughes, B. C. Pierce, A. Spector-Zabusky, D. Vytiniotis, A. Azevedo de Amorim, and L. Lampropoulos. Testing noninterference, quickly. *ICFP*. 2013b. 6, 7, 8
- C. Hrițcu, L. Lampropoulos, A. Spector-Zabusky, A. Azevedo de Amorim, M. Dénès, J. Hughes, B. C. Pierce, and D. Vytiniotis. Testing noninterference, quickly. *JFP*, 26:e4 (62 pages), 2016. 6, 7, 8
- Intel. Software guard extensions (SGX) programming reference, 2014. 10, 42, 46, 78
- ISO/IEC. ISO/IEC 9899:2011 - programming languages – C, 2011. 47
- R. Jagadeesan, C. Pitcher, J. Rathke, and J. Riely. Local memory via layout randomization. *CSF*. 2011. 14, 42, 43, 69, 78, 79
- A. Jeffrey and J. Rathke. Java Jr: Fully abstract trace semantics for a core Java language. *ESOP*. 2005a. 36, 43, 44, 62, 63, 69, 71, 79
- A. Jeffrey and J. Rathke. A fully abstract may testing semantics for concurrent objects. *TCS*, 338(1-3):17–63, 2005b. 43, 44
- L. Jia, S. Sen, D. Garg, and A. Datta. A logic of programs with interface-confined code. *CSF*. 2015. 83
- Y. Juglaret, C. Hrițcu, A. A. de Amorim, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach. Towards a fully abstract compiler using micro-policies: Secure compilation for mutually distrustful components. *CoRR*, abs/1510.00697, 2015. 7, 11, 46, 77
- Y. Juglaret, C. Hrițcu, A. Azevedo de Amorim, B. Eng, and B. C. Pierce. Beyond good and evil: Formalizing the security guarantees of compartmentalizing compilation. *CSF*, 2016. 7, 12, 14, 35, 44, 48, 49, 50, 52, 56, 58, 59, 60, 65, 69, 71, 78, 79, 81
- J. Kang, Y. Kim, C.-K. Hur, D. Dreyer, and V. Vafeiadis. Lightweight verification of separate compilation. *POPL*, 2016. 20, 60, 62
- A. Kennedy. Securing the .net programming model. *Theoretical Computer Science*, 364(3):311–317, 2006. 35, 42, 78
- D. Kilpatrick. Privman: A library for partitioning applications. *USENIX FREENIX*. 2003. 10, 19, 46
- T. F. Knight, Jr., A. DeHon, A. Sutherland, U. Dhawan, A. Kwon, and S. Ray. SAFE ISA (version 3.0 with interrupts per thread), 2012. 76
- R. Krebbers. *The C Standard Formalized in Coq*. PhD thesis, Radboud University Nijmegen, 2015. 9, 47
- J. Kroll, G. Stewart, and A. Appel. Portable software fault isolation. *CSF*. 2014. 80

- R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: a verified implementation of ML. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*. 2014. 10, 15, 18, 21
- O. Kupferman and M. Y. Vardi. Robust satisfaction. *CONCUR*, 1999. 15, 44, 79
- L. Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley, 2002. 18, 21
- L. Lamport and F. B. Schneider. Formal foundation for specification and verification. In *Distributed Systems: Methods and Tools for Specification, An Advanced Course*, 1984. 14, 15, 18, 19, 21, 48, 53
- L. Lamport, R. E. Shostak, and M. C. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982. 80
- L. Lampropoulos, D. Gallois-Wong, C. Hrițcu, J. Hughes, B. C. Pierce, and L. Xia. Beginner’s Luck: A language for random generators. *POPL*. 2017. 8
- L. Lampropoulos, Z. Paraskevopoulou, and B. C. Pierce. Generating good generators for inductive relations. *PACMPL*, 2(POPL):45:1–45:30, 2018. 8
- A. Larmuseau, M. Patrignani, and D. Clarke. A secure compiler for ML modules. *APLAS*, 2015. 14, 42, 48, 78
- C. Lattner. What every C programmer should know about undefined behavior #1/3. LLVM Project Blog, 2011. 47
- J. Lee, Y. Kim, Y. Song, C. Hur, S. Das, D. Majnemer, J. Regehr, and N. P. Lopes. Taming undefined behavior in LLVM. *PLDI*, 2017. 67
- X. Leroy. Formal verification of a realistic compiler. *CACM*, 52(7):107–115, 2009a. 10, 13, 15, 17, 18, 20, 21, 33, 43, 45, 47, 49, 50, 54, 57, 62, 63, 71
- X. Leroy. Formal verification of a realistic compiler. *CACM*, 52(7):107–115, 2009b. 82
- X. Leroy and S. Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *JAR*, 41(1):1–31, 2008. 13, 49, 67
- Z. Manna and A. Pnueli. *Temporal verification of reactive systems: safety*. Springer Science & Business Media, 2012. 18, 21
- G. Martínez, D. Ahman, V. Dumitrescu, N. Giannarakis, C. Hawblitzel, C. Hrițcu, M. Narasimhamurthy, Z. Paraskevopoulou, C. Pit-Claudel, J. Protzenko, T. Ramananandro, A. Rastogi, and N. Swamy. Meta-F*: Proof automation with SMT, tactics, and metaprograms. arXiv:1803.06547, 2018. 8
- J. McLean. Proving noninterference and functional correctness using traces. *Journal of Computer Security*, 1(1):37–58, 1992. 15, 26, 28

- B. Montagu, B. C. Pierce, and R. Pollack. A theory of information-flow labels. *CSF*. 2013. 6
- G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan. RockSalt: better, faster, stronger SFI for the x86. *PLDI*. 2012. 72, 73, 80
- E. Mullen, D. Zuniga, Z. Tatlock, and D. Grossman. Verified peephole optimizations for Comp-Cert. *PLDI*, 2016. 66
- S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. CETS: compiler enforced temporal safety for C. *ISMM*. 2010. 9, 82
- S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Everything you want to know about pointer-based checking. *SNAPL*. 2015. 9
- G. Neis, C. Hur, J. Kaiser, C. McLaughlin, D. Dreyer, and V. Vafeiadis. Pilsner: a compositionally verified compiler for a higher-order imperative language. *ICFP*. 2015. 20
- M. S. New, W. J. Bowman, and A. Ahmed. Fully abstract compilation via universal embedding. *ICFP*, 2016. 14, 17, 19, 36, 38, 39, 42, 43, 69, 78, 79
- Z. Paraskevopoulou, C. Hrițcu, M. Dénès, L. Lampropoulos, and B. C. Pierce. Foundational property-based testing. *ITP*. 2015. 8, 66, 73
- M. Patrignani and D. Clarke. Fully abstract trace semantics for protected module architectures. *CL*, 42:22–45, 2015. 43, 62, 63, 69, 71, 79
- M. Patrignani and D. Garg. Secure compilation and hyperproperty preservation. *CSF*. 2017. 44
- M. Patrignani and D. Garg. Robustly safe compilation. *CoRR*, abs/1804.00489, 2018. 14, 43, 45, 60, 62, 79
- M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens. Secure compilation to protected module architectures. *TOPLAS*, 2015. 14, 19, 35, 36, 42, 48, 69, 78
- M. Patrignani, D. Devriese, and F. Piessens. On modular and fully-abstract compilation. *CSF*. 2016. 11, 14, 35, 36, 42, 43, 46, 48, 69, 78
- M. Patrignani, A. Ahmed, and D. Clarke. Formal approaches to secure compilation: A survey of fully abstract compilation and related work. *ACM Computing Surveys*, 2019. 14, 42, 45
- J. T. Perconti and A. Ahmed. Verifying an open compiler using multi-language semantics. *ESOP*. 2014. 20
- J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hrițcu, K. Bhargavan, C. Fournet, and N. Swamy. Verified low-level programming embedded in F*. *PACMPL*, 1(ICFP):17:1–17:29, 2017. 8, 10, 12, 13, 81
- N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *12th USENIX Security Symposium*. 2003. 10, 19, 46

- W. Rafnsson, L. Jia, and L. Bauer. Timing-sensitive noninterference through composition. In M. Maffei and M. Ryan, editors, *Principles of Security and Trust - 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. 2017. 84
- J. Regehr. A guide to undefined behavior in C and C++, part 3. Embedded in Academia blog, 2010. 47, 57, 65
- C. Reis and S. D. Gribble. Isolating web programs in modern browser architectures. *EuroSys*. 2009. 10, 19, 46, 82
- N. Roessler and A. DeHon. Protecting the stack with metadata policies and tagged hardware. *IEEE S&P*. 2018. 82
- A. W. Roscoe. CSP and determinism in security modelling. *S&P*. 1995. 28
- A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003. 15, 26
- A. Sabelfeld and D. Sands. A PER model of secure information flow in sequential programs. *HOSC*, 14(1):59–91, 2001. 15, 28
- F. Schneider. *On Concurrent Programming*. Texts in Computer Science. Springer New York, 1997. 18, 21
- L. Simon, D. Chisnall, and R. J. Anderson. What you get is what you C: Controlling side effects in mainstream C compilers. *EuroS&P*. 2018. 19
- L. Skorstengaard, D. Devriese, and L. Birkedal. Reasoning about a machine with local capabilities - provably safe stack and return pointer management. *ESOP*, 2018a. 10, 46, 80
- L. Skorstengaard, D. Devriese, and L. Birkedal. Enforcing well-bracketed control flow and stack encapsulation using linear capabilities. 2nd Workshop on Principles of Secure Compilation (PriSC), 2018b. 80
- G. Stewart, L. Beringer, S. Cuellar, and A. W. Appel. Compositional CompCert. *POPL*. 2015. 20
- G. T. Sullivan, S. Chiricescu, A. DeHon, D. Demange, S. Iyer, A. Kliger, G. Morrisett, B. C. Pierce, H. Reubenstein, J. M. Smith, A. Thomas, J. Tov, C. M. White, and D. Wittenberg. SAFE: A clean-slate architecture for secure systems. In *Proceedings of the IEEE International Conference on Technologies for Homeland Security*, 2013. 6
- N. Swamy, C. Hrițcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Zanella-Béguelin. Dependent types and multi-monadic effects in F*. *POPL*. 2016. 8, 10
- D. Swasey, D. Garg, and D. Dreyer. Robust and compositional verification of object capability patterns. *PACMPL*, 1(OOPSLA):89:1–89:26, 2017. 15, 24, 44, 56, 79

- L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. *IEEE S&P*. 2013. 9, 13, 47, 48, 73
- G. Tan. Principles and implementation techniques of software-based fault isolation. *FTSEC*, 1(3):137–198, 2017. 10, 19, 24, 46
- S. Tsampas, A. El-Korashy, M. Patrignani, D. Devriese, D. Garg, and F. Piessens. Towards automatic compartmentalization of C programs on capability machines. *FCS*, 2017. 11, 46
- N. van Ginkel, R. Strackx, J. T. Muehlberg, and F. Piessens. Towards safe enclaves. *HotSpot*, 2016. 11, 46
- T. Van Strydonck, D. Devriese, and F. Piessens. Linear capabilities for modular fully-abstract compilation of verified code. 2nd Workshop on Principles of Secure Compilation (PriSC), 2018. 11, 46
- N. Vasilakis, B. Karel, N. Roessler, N. Dautenhahn, A. DeHon, and J. M. Smith. BreakApp: Automated, flexible application compartmentalization. *NDSS*. 2018. 50
- R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *SOSP*, 1993. 10, 19, 24, 46, 72
- X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. *SOSP*. 2013. 47
- R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. H. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. *S&P*. 2015a. 82
- R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. H. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. *IEEE S&P*. 2015b. 10, 43, 46, 66
- P. Wilke, F. Besson, S. Blazy, and A. Dang. CompCert for software fault isolation. Secure Compilation Meeting (SCM), 2017. 80
- T. Y. C. Woo and S. S. Lam. A semantic model for authentication protocols. *IEEE S&P*. 1993. 53
- G. Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014. 25
- B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. *CACM*, 53(1):91–99, 2010. 10, 46, 82
- A. Zakinthinos and E. S. Lee. A general theory of security properties. *S&P*. 1997. 17

- S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. *CSFW*. 2003. 15, 26, 28
- L. Zhao, G. Li, B. D. Sutter, and J. Regehr. ARMor: Fully verified software fault isolation. *EMSOFT*. 2011. 80
- J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. HACL*: A verified modern cryptographic library. *CCS*, 2017. 8, 10, 13

Appendix

The results presented in this habilitation have previously appeared in a series of research papers that are appended below. I have substantially contributed to each of these papers, which I co-authored with my students and several external collaborations.

Carmine Abate, Roberto Blanco, Deepak Garg, Cătălin Hrițcu, Marco Patrignani, and Jérémy Thibault. Journey beyond full abstraction: Exploring robust property preservation for secure compilation. *arXiv:1807.04603*, July 2018b.

Carmine Abate, Arthur Azevedo de Amorim, Roberto Blanco, Ana Nora Evans, Guglielmo Fachini, Cătălin Hrițcu, Théo Laurent, Benjamin C. Pierce, Marco Stronati, and Andrew Tolmach. When good components go bad: Formally secure compilation despite dynamic compromise. In *25th ACM Conference on Computer and Communications Security (CCS 2018)*, pages 1351–1368, October 2018a. (Acceptance rate: 134/809=16.6).

Yannis Juglaret, Cătălin Hrițcu, Arthur Azevedo de Amorim, Boris Eng, and Benjamin C. Pierce. Beyond good and evil: Formalizing the security guarantees of compartmentalizing compilation. In *29th IEEE Symposium on Computer Security Foundations (CSF)*, pages 45–60. IEEE Computer Society Press, July 2016. (Acceptance rate: 31/87=0.36).

Arthur Azevedo de Amorim, Maxime Dénès, Nick Giannarakis, Cătălin Hrițcu, Benjamin C. Pierce, Antal Spector-Zabusky, and Andrew Tolmach. Micro-Policies: Formally verified, tag-based security monitors. In *36th IEEE Symposium on Security and Privacy (Oakland S&P)*, pages 813–830. IEEE Computer Society, May 2015. (Acceptance rate: 55/420=0.13).

Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Cătălin Hrițcu, David Pichardie, Benjamin C. Pierce, Randy Pollack, and Andrew Tolmach. A verified information-flow architecture. *Journal of Computer Security (JCS); Special Issue on Verified Information Flow Security*, 24(6):689–734, December 2016. (Supersedes POPL 2014 paper with the same name; Acceptance rate: 51/220=0.23)