KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Sound modular reasoning about security properties of imperative programs

Pieter Agten

Supervisor:
Prof. dr. ir. F. Piessens
Prof. dr. B. Jacobs, co-supervisor

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering Science: Computer Science

June 2015

Sound modular reasoning about security properties of imperative programs

Pieter AGTEN

Examination committee:
Prof. dr. A. Bultheel, chair
Prof. dr. ir. F. Piessens, supervisor
Prof. dr. B. Jacobs, co-supervisor
Prof. dr. ir. W. Joosen
Prof. dr. D. Clarke
Prof. dr. ir. B. Preneel
Prof. dr. M. Maffei
(Saarland University)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering Science: Computer Science

June 2015

Preface

This dissertation is the result of four exciting years conducting research as a member of the DistriNet research group. Of course, this work could not have come to be, without the help of many others. First and foremost, my sincerest gratitude goes out to my supervisor Frank Piessens, who has guided and encouraged me throughout my PhD, and has given me the freedom to explore my interests within the field of software security. His dedication to his students, despite his busy schedule, is truly inspiring.

My sincere thanks also go out to my co-supervisor Bart Jacobs, for his support and for answering all my formal methods-related questions, and to Wouter Joosen for his dedication to our research group. I also want to thank my other jury members for their time and valuable suggestions for improving this dissertation: Adhemar Bultheel, Dave Clarke, Matteo Maffei, and Bart Preneel.

A special thank you to my office colleagues and lunch mates: Jesper Cockx, Francesco Gadaleta, Milica Milutinovic, Jan Tobias Mühlberg, Nick Nikiforakis, Job Noorman, Marco Patrignani, Zubair Rafique, Rula Sayaf, Raoul Strackx, Mathy Vanhoef, Gitte Vanwinckelen, Thomas Vissers, and Frédéric Vogels. We had a lot of fun times and many interesting conversations, both in and out of the office.

During my PhD I've had the privilege of talking to and working with so many outstanding researchers. I especially enjoyed working with my fellow software design, networking and computer architecture teaching assistants and lecturers Yolande Berbers, Tom Holvoet, Danny Hughes, Christophe Huygens, Bart Jacobs, Roel Wuyts, Rafael Bachiller Soler, Mario Henrique Cruz Torres, Willem De Groef, Thomas Heyman, Rinde van Lon, Nelson Matthys, Job Noorman, Willem Penninckx, Andreas Put, Gowri Sankar Ramachandran, Arun Kishore Ramakrishnan, Philippe De Ryck, Klaas Thoelen, Marko van Dooren, Dries Vanoverberghe and Gijs Vanspauwen. I also enjoyed the many interesting conversations I've had with all my other colleagues, of whom I'd

like to explicitly mention Maarten Decat, Lieven Desmet, Dominique Devriese, Adriaan Larmuseau, Jef Maerien, Sam Michiels, Radu Muschevici, Dimitar Mushkov, Minh Ngo, Steven Op de beeck, Koosha Paridel, Davy Preuveneers, José Proença, Bob Reynders, Ilya Sergey, Jan Smans, Amin Timany, Steven Van Acker, Alexander van den Berghe, Tom Van Goethem, Jan Van Haaren, Thomas Winant and Nayyab Zia Naqvi. It is all these people that give our department the amicable atmosphere that it has.

Of course, our department and research group could not function without our chair Ronald Cools and our administrative assistants Liesbet Degent, Fred Jonker, Karin Michiels, Margot Peeters, Anne-Sophie Putseys, Esther Renson, Marleen Somers, Karen Spruyt, Inge Vandenborne and Karen Verresen. Also a big thank you to our DistriNet project office members Katrien Janssens, Ghita Saevels and Annick Vandijck; and to the people that so judiciously manage the department's computer infrastructure: Anita Ceulemans, Jean Huens, Bart Swennen, Greg Vanhove, Kris Wessels and Steven Wittevrouw.

My deepest gratitude goes to my friends and family, in particular my parents Gerard and Mariette, my brother Geert and his wife Marieke, and of course my fiancé Mieke, for all their love and support. One last person that I cannot forget to thank is Ivan Lavigne, for nurturing my passion for programming a long time ago, when I was only just starting high school.

Finally, I would like to thank the following institutions for their financial support.

- The Research Foundation - Flanders (FWO)
- The EU FP7 project NESSoS
- The Intel Lab's University Research Office
- The Research Fund KU Leuven

Abstract

Modern-day imperative programming languages such as C++, C# and Java offer protection facilities such as abstract data types, field access modifiers, and module systems. Such abstractions were mainly designed to enforce software engineering principles such as information hiding and encapsulation, but they can also be used to enforce security properties of programs. Unfortunately, these source-level security properties are typically lost during compilation to low-level machine code. For instance, access to private instance fields is restricted by the programming language's type system at the source-code level, but such restrictions are not in place at the assembly level. This can leave a software module vulnerable to attacks at the assembly level, such as code-injection attacks and kernel-level malware.

In the first part of this dissertation, we present a compilation scheme that is *fully abstract*, which means that the high-level security properties of a software module are maintained after compilation. We formalize this property as preservation of contextual equivalence, which means that two assembly-level compiled modules should only be distinguishable from each other by another module interacting with them, when their original source-level modules can also be distinguished from each other by a source-level module. In other words, a fully abstract compiler ensures that any possible assembly-level interaction is explainable at the source-code level. This effectively reduces the power of an assembly-level attacker to the power of a source-level attacker. To achieve this strong security property, the compiler relies on the presence of a fine-grained, program counter-based memory access protection primitive, as part of the assembly-level target language. We formalize our compilation scheme, prove that it is fully abstract and we show by means of a prototype implementation that the assumed memory access protection primitive can be realized efficiently on modern commodity hardware.

In the second part of this dissertation, we discuss the sound modular verification of imperative programs executing in an unverified context. We focus on Hoare

logic-based software verification techniques, which enable developers to statically prove correctness and safety properties of imperative programs. Unfortunately, the runtime guarantees offered by such verification techniques are relatively limited when the verified codebase is part of a program that also contains unverified code. In particular, unverified code might not behave as assumed by the verifier, leading to failure of all verified assertions that were based on those assumptions. This is particularly troublesome in memory-unsafe languages, where a memory safety error in unverified code can corrupt the runtime state of verified code.

We have developed a series of runtime checks to be inserted at the boundary between the verified and unverified parts of a program, which check that the unverified code behaves as expected. A key problem that we had to solve, is how to ensure that memory errors or malicious code in the unverified part cannot corrupt the state of the verified part. We solve this problem in two steps. Firstly, the inserted boundary checks perform an integrity check on the heap memory owned by the verified code, to ensure that bad writes to heap memory by the unverified part are detected upon (re-)entry to verified code. Secondly, we use the mechanism of fully abstract compilation developed in the first part of this dissertation, for the integrity protection of the verified code's local variables and control flow metadata on the runtime call stack. The combination of these protection measures results in a very strong modular soundness guarantee: no verified assertion in the verified codebase will ever fail at runtime, even if the verified codebase interacts with unverified code. We formalize the developed program transformations, prove that they are sound and precise, and we show by means of micro and macro benchmarks that the performance overhead of the boundary checks is sufficiently low for practical applicability.

Beknopte samenvatting

Moderne programmeertalen zoals C++, C# en Java bieden diverse beschermingsfaciliteiten aan, zoals abstracte gegevenstypes, toegangsbeperkingen voor instantievelden en modulesystemen. Dergelijke abstracties zijn ontworpen om softwareontwerpprincipes te ondersteunen, zoals het afschermen van private informatie en de inkapseling van implementatieaspecten achter een publieke interface. Ze kunnen echter ook gebruikt worden voor het realiseren van beveiligingseigenschappen van computerprogramma's. Helaas gaan zulke eigenschappen typisch verloren bij het vertalen van de broncode van een programma naar machinetaal. Op broncodeniveau wordt bijvoorbeeld de toegang tot private instantievelden beperkt door het typesysteem van de programmeertaal, maar een dergelijke beperking bestaat niet op het niveau van de machinetaal. Dit kan ertoe leiden dat een vertaalde softwaremodule kwetsbaar is voor aanvallen op het niveau van de machinetaal, zoals code-injectieaanvallen en malware die zich in het besturingssysteem manifesteert.

In het eerste deel van dit proefschrift stellen we een vertalingsschema voor dat *volledig abstract* is, wat ruwweg betekent dat de vertaling de beveiligingseigenschappen die op broncodeniveau gelden behoudt tijdens het vertalingsproces. De definitie van volledige abstractie is gebaseerd op het behoud van contextuele equivalenties: twee softwaremodules mogen op machinetaalniveau enkel en alleen van elkaar te onderscheiden zijn door een derde module die met hen kan interageren, wanneer hun oorspronkelijke bronmodules ook van elkaar te onderscheiden zijn door een module op broncodeniveau. Een volledig abstracte vertaler zorgt ervoor dat elke mogelijke interactie op het niveau van de machinetaal uit te leggen is op het niveau van de brontaal. Deze eigenschap verzekert dat een aanvaller die zijn aanval mag schrijven in machinetaal niet krachtiger is dan een aanvaller die zijn code moet schrijven in een veilige hoog-niveau brontaal. Om deze sterke beveiligingseigenschap te verkrijgen, rekent ons vertalingsschema op de aanwezigheid van een beveiligingsprimitief in de doeltaal dat de toegang tot bepaalde geheugengebieden van een proces afschermt op basis van de waarde van

de programmateller. De contributies van dit deel van het proefschrift bestaan uit het formuleren en formaliseren van het vertalingsschema, het bewijzen van de volledige abstractie van de vertaling, en het aantonen door middel van een prototype-implementatie dat het vereiste beveiligingsprimitief efficiënt realiseerbaar is op hedendaagse, alom beschikbare hardware.

In het tweede deel van dit proefschrift bespreken we hoe imperatieve programma's correct modulair geverifieerd kunnen worden, zelfs wanneer ze worden uitgevoerd in een ongeverifieerde omgeving. We richten ons op verificatiemethodes die gebaseerd zijn op Hoarelogica, dewelke ontwikkelaars toelaten om de correctheid van imperatieve programma's te bewijzen op basis van de programmabroncode. Helaas zijn de uitvoeringsgaranties van dergelijke methodes beperkt wanneer de geverifieerde code deel uitmaakt van een programma dat ook ongeverifieerde code bevat. Het is namelijk mogelijk dat de ongeverifieerde code zich niet gedraagt zoals aangenomen tijdens het verificatieproces, wat kan leiden tot het falen van eender welke assertie die gebaseerd is op die aannames. Dit is vooral problematisch bij geheugenonveilige programmeertalen zoals C, waarbij een geheugenfout in ongeverifieerde code de uitvoeringstoestand van geverifieerde code ongeldig kan maken.

Om dit probleem op te lossen, hebben we een reeks controles ontwikkeld die op de grens tussen de geverifieerde en ongeverifieerde code kunnen worden ingevoegd. Deze controles worden uitgevoerd bij elke functieoproep tussen geverifieerde en ongeverificeerde code en verzekeren dat de ongeverifieerde code zich gedraagt zoals aangenomen tijdens de verificatie. Een belangrijk probleem dat hierbij moet worden opgelost, is hoe de controles kunnen verzekeren dat geheugenfouten in het ongeverifieerde deel van het programma de uitvoeringstoestand van het geverifieerde deel niet kunnen beïnvloeden. We lossen dit probleem in twee stappen op. In de eerste stap zorgen we ervoor dat de ingevoegde controles de integriteit controleren van het heapgeheugen dat toegankelijk is voor de geverifieerde code. Hierdoor zullen ongeldige geheugenoperaties uitgevoerd door ongeverifieerde code gedetecteerd worden wanneer de geverifieerde code (terug) wordt binnengetreden. In de tweede stap gebruiken we de volledig abstracte vertaling die ontwikkeld werd in het eerste deel van dit proefschrift, om de lokale variabelen en besturingsstroominformatie van de geverifieerde code op de uitvoerstapel te beschermen. De combinatie van deze twee beschermingsmaatregelen resulteert in een zeer sterke modulaire correctheidsgarantie: geen enkele geverifieerde assertie in de geverifieerde code kan falen tijdens de uitvoering van het programma. De bijdragen van het tweede deel van het proefschrift bestaan uit het formuleren en formaliseren van de voorgestelde controles, het bewijzen dat deze controles correct en precies zijn, en het aantonen door middel van snelheidstesten dat de performantiekost van de controles voldoende laag is voor praktische toepasbaarheid.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software today plays an important role in many areas of society, including commerce, government, industry, education, healthcare, etc. Not only is software ubiquitous, it is also becoming increasingly complex. For instance, recent desktop operating systems are built from tens of millions of lines of code [86], and the software for a modern high-end car is even estimated to have over 100 million lines of code [39]. The impact of software bugs can be significant, since they can lead to exploitable security vulnerabilities such as the recent Heartbleed [91], Shellshock [49], and Ghost [48] bugs, or can even result in catastrophic events such as air plane crashes, radiation overdoses and pipeline explosions [102]. However, the consequences of software bugs do not have to be so spectacular to have a large impact. Consider for instance the number of flights delayed or canceled due to software problems at check-in terminals, and the amount of revenue lost by merchants due to payment system failures. Annual productivity and turnover losses due to software failure are estimated at €1.6 billion in the Netherlands [118] and up to $59.5 billion in the US [82] alone.

1.1 Abstractions in programming languages

To tackle the ever-increasing complexity of software, developers rely on various techniques and tools, such as novel programming languages and methodologies, smart integrated development environments, version control software, continuous integration, etc. However, one of the most important software development techniques is also likely one of the oldest: the technique of *abstraction*. When programmers write code, they write functions or procedures, that are given a

name and (hopefully) some documentation describing the expected input, the intended output, any potential side-effects and possibly the runtime complexity of the function. When a programmer later uses a function, she will rely on the function's name and documentation (i.e., the function's *interface*) to know what that function does. Thus, the programmer has abstracted away *how* the function works, and focuses only on *what* it does. Such abstractions are necessary to hide the complexity of the system under development, since for all but the smallest software systems, it would be infeasible for any developer to know exactly how each function works. From the basic building block of function abstraction, higher-level abstractions can be built, for instance in the form of a software library with a documented application programming interface (API).

Other forms of abstraction that are commonly provided by modern-day, general-purpose programming languages, are execution platform abstractions, module systems, field access modifiers, structured control flow, etc. An example of execution platform abstraction is the fact that a compiled C program that conforms to one of the proper C standards will behave the same on various kinds of execution platforms that feature heterogeneous types of processors, operating systems, and system libraries. The C programmer does not need to worry about how the provided high-level constructs will be translated into low-level assembly code. Similarly, the field access modifier abstraction allows a Java developer to mark a class instance field as private to ensure that access to this field will be restricted to the class's local methods by the type system. Even at the level of assembly code, abstractions are made. For instance, the assembler hides the specific binary encoding of each assembly instruction from the programmer.

Each layer of abstraction hides the complexity of the layer beneath it. But, as a consequence of hiding complexity, each abstraction layer also typically restricts the power of the developer, in some way or another. For instance, at the assembly code level, control can jump from any memory location to any other location, while most high-level programming languages only allow functions to be entered from the top (non-local gotos are an infamous exception to this rule). In case of a software library, the programmer is restricted to using the library's public API and usually cannot access the private functions that are internal to the library. Such restrictions are in place for two reasons: (1) to protect the programmer and provide a convenient and coherent interface to work with, and (2) to ensure the internal consistency of the abstraction. As an example of the latter reason, consider again a compiler for a high-level programming language. When compiling a function, the compiler will assume that the code interacting with this function (i.e., the *context* surrounding it) adheres to the conventions set out by the target platform's application binary interface (ABI). Ensuring correctness, safety and interoperability is much easier in this setting than in the

case where compiled functions must be able to safely interact with arbitrary assembly code. Making this assumption is fair, because normal code, written by well-behaved programmers and compiled by well-behaved compilers, will always adhere to the assumed conventions.

The problem is, however, that most of the abstractions available at the source level, are lost during compilation towards low-level assembly code. This means that code that is injected directly at the assembly level (i.e., hand crafted assembly code, which does not necessarily result from the compilation of valid high-level code), is not bound to the restrictions set out by the high-level abstractions. This is a real issue, as attacks in practice often rely on injecting machine code into a process' address space [41, 42, 122], and kernel-level malware can attack any process in the system at the assembly level. An assembly-level attacker is hence strictly more powerful than a source-level attacker.

This is the problem we are concerned with in Chapter 2 and Chapter 3. In these chapters, we develop a compilation scheme from an imperative high-level language to a low-level assembly target language, that will prevent *any* low-level code from performing actions that could not occur at the source code level. In other words, the power of an assembly-level attacker will be reduced to that of a source-level attacker, because any attack that can be mounted at the low level is explainable at the source-code level. To obtain this strong security property, the compilation scheme relies on a special memory access control mechanism in the target language, which we show to be implementable on commodity hardware available today. We call the compilation scheme *fully abstract*, because it allows all behavior of a system to be understood from studying the source code alone. That is, even in case of code-injection attacks at the assembly level, will the source-level security properties of the system be maintained. This property enforces the principle of source-based reasoning, which was characterized by Baltopoulos and Gordon [17] as the proposition that security properties of a software system should follow from review of the system's source code and its source-level semantics alone, and hence should not depend on details of the compiler or execution platform used to execute the system.

1.2 Sound modular software verification

A different approach for reducing the number of bugs in complex pieces of software, lies in the use of formal software verification techniques. In general, formal software verification denotes any automated or semi-automated technique for checking the validity of certain properties about a rigorously specified model of a software system. In this text however, we will focus primarily on a specific

type of formal software verification, namely the techniques based on the inductive assertion method of Floyd and Hoare [46, 53] for imperative programs. This method involves annotating functions with pre- and postconditions, which relate the original program state before the function has executed to the state after the function has finished execution. Intermediate annotations can also be specified, in order to express further assertions to prove, or to help guide the verifier towards the verification goal. Function contracts and intermediate assertions can express interesting program properties such as memory safety, absence of race conditions and deadlocks, adherence to communication protocols, or even full functional correctness. Program verification in this case corresponds to proving that, if a function is called in any state and with any input values such that its precondition holds, that the function's intermediate assertions hold at the corresponding program points and that its postcondition will hold after the function finishes execution. If the verification of a function using a sound verifier is successful, we can be sure that the source code of the verified function implements the behavior expressed by the function's contract. If the program verifier cannot prove correctness, it can usually give a hint on why it failed. This can either be because there is a bug in the function under verification, or because the verifier simply was not able to prove correctness.

A successfully verified function adheres to its specified contract for *any* possible input that satisfies the precondition. Hence, the correctness guarantees offered by formal verification are significant. Unfortunately, these hard guarantees usually come at a price. Creating or annotating the model of a complex software system and formulating a formal description of the properties to verify, can be a difficult task, requiring substantial human effort and expertise. Although research for reducing the required effort and expertise is ongoing, it remains one of the reasons of why the industrial adoption of formal software verification remains slow [121] in comparison to the integration of formal verification techniques in the hardware industry [44].

The substantial investment required for applying formal verification is also one of the main reasons why it is essential that verification methods are *modular*. That is, it should be possible to soundly verify only a part of an application, leaving the rest of the code unverified. This allows a gradually increasing part of the codebase to be verified over time, rather than requiring the entire codebase to verified in order to have any guarantees at all. The inductive assertion verification methods we discuss in this text satisfy this requirement. Functions that need not be verified can simply be left unannotated, except those that are called from a verified function. Although those functions are unverified, they must be supplied with a pre- and postcondition contract because the verifier needs to know the functions' expected behavior in order to verify their callers. Successful verification using a sound verification method then guarantees that

any execution of the program will comply with the verified specifications, under the critical condition that the unverified functions behave as dictated by their contracts and that they always satisfy the precondition when calling a verified function.

Problems occur when this critical condition is not met. When the assumptions made by the verifier about the behavior of unverified functions turn out to be false, all further proof results that were based upon these assumptions might no longer hold. Hence, if the unverified codebase does not behave as expected, all bets are off. The situation is even worse in memory-unsafe languages such as C, because, even if the unverified codebase behaves according to its contracts in terms of function input-output behavior, a memory-safety error (such as a buffer overflow) in unverified code can corrupt the runtime state of verified code. Thus, the runtime guarantees of a partially verified codebase are more limited than they appear to be at first sight, especially for memory-unsafe languages.

In Chapter 5 and Chapter 6, we address this problem by proposing a program transformation that adds a series of runtime checks at the boundary between verified and unverified code, in order to detect when the unverified codebase does not behave according to its contracts. A key problem that must be solved is how to ensure that memory errors in the unverified codebase cannot corrupt the state of the verified codebase. We propose to solve this problem by having the runtime checks perform an integrity check on the heap memory of the verified codebase: on re-entry to a verified function, we check that the part of the heap "owned" by the verified codebase has not been changed by the unverified function that was called. This ensures that bad writes to the heap performed by the unverified codebase are detected upon re-entry to verified code. Protecting heap memory alone is not enough, however. We must also protect the local variables within the activation records of verified functions, and the control flow metadata on the call stack. Since it is impossible to tamper with activation records at the source code level, we can rely on the fully abstract compilation scheme of Chapter 2 to protect against this kind of misbehavior. The combination of the boundary checks and the fully abstract compilation protection of local variables and control flow result in a strong modular soundness guarantee: no verified assertion in the verified codebase will ever fail at runtime, even if the verified code interacts with unverified code.

1.3 Attacker model and scope

Many system security research papers model the attacker they are protecting against as a so-called *input-providing* attacker, who can interact with the

program under attack by providing input and reading output. An attacker can for instance craft a special string to be given as input to the program under attack, in order to exploit a buffer overflow vulnerability, SQL injection vulnerability, or a logic flaw in the program. The input-providing attacker is a suitable model for an attacker that is trying to subvert a service running on a hardened and well-protected machine.

In this dissertation, however, we use the strictly stronger *in-process* attacker model, which assumes the attacker has the power to load and execute arbitrary machine code in the process executing the program under attack. More specifically, we model the attacker as an arbitrary piece of low-level assembly code that can interact with the code resulting from our translation schemes. An in-process attacker can for instance try to scan memory for secrets, overwrite control-flow data and non-control data of the program under attack, and can even attempt to overwrite other code that was loaded into the process. We place no a priori limitations on the attacker's code, but the interactions that it can engage in with the program under attack are restricted by (1) the memory access control mechanism that we assume is part of our low-level target language and (2) the constraints enforced by the runtime checks inserted by our fully abstract compiler and the boundary checking program translation. The in-process attacker is a suitable model for applications that can be extended at run-time with (binary) plugins, or for situations in which the attacker has already successfully exploited a component that the program under attack is interacting with.

Although this is a very strong attacker model, the following categories of attacks are still out of scope.

- Attacks that exploit logic flaws or other bugs that are present at the source-code level. The goal of our compilation and translation schemes is to preserve source-level semantics when our code is executing in an untrusted low-level environment. Hence, any bugs present at the source-code level will still be present after translation.
- Attacks that depend upon features not modeled in our high- or low-level languages. For instance, our assembly language does not model time nor I/O devices, hence timing- and I/O-based side-channel attacks are excluded from our attacker model.
- Attacks that occur during the bootstrapping of the system. Our model assumes that the program under attack is securely loaded into memory before the attacker has a chance to mount her attack. Hence, attacks against the program before it is loaded (e.g., while it is stored on disk after compilation, or during the loading phase) are excluded.

- Attacks against the availability of the system. Since an attacker can execute arbitrary code, it is trivial for her to enter an infinite loop and hence prevent any program progress.

Finally, it is interesting to consider the kinds of system properties that can be expressed using the techniques we employ. For our fully abstract compilation scheme, we conjecture that it preserves all hypersafety properties [28] of the module to compile, but this is only an intuition that has not been formally pursued in this work. As for the second part of this dissertation, the logics underlying the verification techniques we reason about can refer only to a single execution at a time, and hence are limited to expressing 1-hypersafety properties such as memory safety and functional correctness.

1.4 Other research conducted

The works presented in this dissertation are a subset of the research I have conducted over the past four years. For the contents of this dissertation, I have selected the works of which I am the principal author and which are related to the sound modular reasoning about security properties of imperative programs. Below I give a brief description of the other research I conducted during my PhD.

FAMoS: A Flexible Active Monitoring Service for Wireless Sensor Networks
This paper presents FAMoS, a network traffic monitoring framework for low-end wireless sensor networks. In this framework, each wireless sensor node locally collects data about the number of transmitted and received network packets for different categories of network traffic. The nodes then periodically transmit the collected data to a back-end system for data aggregation and analysis. This paper is based upon the work conducted as part of my Master's thesis, which was performed under supervision of Jef Maerien, Christophe Huygens, and Wouter Joosen.

Publication data:

J. Maerien, P. Agten, C. Huygens, W. Joosen. "FAMoS: A Flexible Active Monitoring Service for Wireless Sensor Networks". In: *DAIS*. 2012, pp. 104–117

Recent Developments in Low-level Software Security This work describes state-of-the-art approaches for securing code written in C-like languages, for two types of attackers. The first type is the interactive attacker, who can interact with the program under attack by providing input and reading output. For instance, an interactive attacker could perform an SQL injection attack. The other type of attacker is the in-process attacker, who can load and execute arbitrary machine code in the process executing the program under attack. The latter attacker model corresponds to the type of attacker we protect against in the works described in this dissertation. This work was an invited paper for the WISTP workshop, jointly co-authored by Nick Nikiforakis, Raoul Strackx, Willem De Groef, Frank Piessens and myself.

Publication data:

P. Agten, N. Nikiforakis, R. Strackx, W. De Groef, F. Piessens. "Recent Developments in Low-level Software Security". In: *Proceedings of the 6th IFIP WG 11.2 International Conference on Information Security Theory and Practice: Security, Privacy and Trust in Computing Systems and Ambient Intelligent Ecosystems*. WISTP'12. Egham, UK: Springer-Verlag, 2012, pp. 1–16

JSand: Complete Client-Side Sandboxing of Third-Party JavaScript Without Browser Modifications This paper presents JSand, a server-driven but client-side enforcing JavaScript sandboxing framework. The framework allows different JavaScript web components included on the same web page to be isolated from each other and from the web page itself. Interactions between different components is still possible, but must adhere to a server-defined policy. For instance, a privacy-aware policy could specify that JavaScript-based advertisements included on a webmail page, are not allowed to access any of the DOM-elements containing the e-mails' subject line and content. This research was conducted in cooperating with Steven Van Acker, Yoran Brondsema, and Phu H. Phung, under supervision of Lieven Desmet and Frank Piessens.

Publication data:

P. Agten, S. Van Acker, Y. Brondsema, P. H. Phung, L. Desmet, F. Piessens. "JSand: complete client-side sandboxing of third-party JavaScript without browser modifications". In: *Proceedings of the 28th Annual Computer Security Applications Conference*. ACSAC '12. Orlando, Florida: ACM, 2012, pp. 1–10

Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base This paper describes Sancus, a hardware-based protected module architecture for networked embedded devices. Sancus implements a program counter-based memory access control scheme similar to the one we describe in Chapter 2, as an extension of the MSP430 microprocessor. The architecture allows the correct, uncompromised execution of a software module to be attested to a remote software provider, and provides authenticated communication between a software module and a software provider. Sancus has a zero-software trusted computing base (TCB), i.e., no software must be trusted in order to achieve the provided security guarantees. The principal author of this work is Job Noorman, who collaborated with me, Wilfried Daniels, Raoul Strackx and Anthony Van Herrewege, under supervision of Christophe Huygens, Bart Preneel, Ingrid Verbauwhede and Frank Piessens.

Publication data:

> J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, F. Piessens. "Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base". In: *Proceedings of the 22nd USENIX Conference on Security*. SEC'13. Washington, D.C.: USENIX Association, 2013, pp. 479–494

Salus: Non-hierarchical Memory Access Rights to Enforce the Principle of Least Privilege This work describes Salus, a Linux kernel modification that provides a program counter-based memory access control scheme similar to the one provided by Sancus. It allows a process to be subdivided into small compartments, each of which is isolated from other compartments. One compartment cannot access the data sections of other compartments and can only jump to specific *entry points* within the code section of other compartments. Interaction between compartments is possible only through function calls, and each compartment is able to authenticate both the compartments that it calls and the compartments that it is called from. The principal author of this work is Niels Avonds, who conducted this research as part of his Master's thesis, which was supervised by Raoul Strackx, me and Frank Piessens.

Publication data:

N. Avonds, R. Strackx, P. Agten, F. Piessens. "Salus: Non-hierarchical Memory Access Rights to Enforce the Principle of Least Privilege". English. In: *Security and Privacy in Communication Networks*. Ed. by T. Zia, A. Zomaya, V. Varadharajan, M. Mao. Vol. 127. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Springer International Publishing, 2013, pp. 252–269

Salus: Kernel Support for Secure Process Compartments This journal paper is an extended version of the work on Salus described above. The main extension consists of adding support for unforgeable references. Under this extension, only knowing a compartment's location in memory is not sufficient for calling it; an additional cryptographic nonce that is specific for each compartment, is required as well. This prevents compromised compartments from being able to scan memory and subsequently access any compartment found by the scan. The principal author of this extended work is Raoul Strackx, who collaborated with me and Niels Avonds, under supervision of Frank Piessens.

Publication data:

R. Strackx, P. Agten, N. Avonds, F. Piessens. "Salus: Kernel Support for Secure Process Compartments". In: *EAI Endorsed Transactions on Security and Safety* 15.3 (Jan. 2015)

Seven Months' Worth of Mistakes: A Longitudinal Study of Typosquatting Abuse This paper presents an empirical study of typosquatting, which is the act of purposefully registering a domain name that is a mistype of a popular domain name. For instance, `www.kuleuben.be` is a typosquatting domain name of `www.kuleuven.be`. Typosquatting has been known and studied for over 15 years, but previous typosquatting studies have always taken a single snapshot of the typosquatting landscape over a limited period of time. This work presents the first content-based *longitudinal* study of typosquatting. The results presented in this work are based upon a data collection experiment of the 500 most popular websites of the Internet, over a period of seven months. The work was conducted in collaboration with Nick Nikiforakis of Stony Brook University, New York, under supervision of Wouter Joosen and Frank Piessens.

Publication data:

P. Agten, W. Joosen, F. Piessens, N. Nikiforakis. "Seven months' worth of mistakes: A longitudinal study of typosquatting abuse". In: *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS 2015)*. Internet Society, Feb. 2015

Secure Compilation to Protected Module Architectures This work describes a fully abstract compilation scheme from a Java-like object-oriented source language, to a target language representing an x86-like processor extended with a program counter-based memory access control mechanism. The work is an extension of the research described in Chapter 2 and Chapter 3, in order to support dynamic memory allocation, dynamic dispatch and exceptions. The principal author of this work is Marco Patrignani, who collaborated with me and Raoul Strackx, under supervision of Bart Jacobs, Dave Clarke and Frank Piessens.

Publication data:

M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, F. Piessens. "Secure Compilation to Protected Module Architectures". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 37.2 (Apr. 2015), 6:1–6:50

1.5 Outline

The rest of this text is structured as follows. In Chapter 2, we discuss in detail the fully abstract compilation scheme referred to in Section 1.1. The contents of this chapter are precise but informal, since the formalization of the secure compilation scheme is left for Chapter 3. In Chapter 4 we give an overview of formal software verification techniques, focusing in particular on the inductive assertion method of Floyd and Hoare, and the verification methods derived from it. Next, in Chapter 5 we discuss in detail the program transformations for the sound modular verification of code executing in an unverified context, as described in Section 1.2. Again, the matter is first discussed informally, and the formalization is left for Chapter 6. Finally, in Chapter 7 we conclude this dissertation by summarizing the obtained results and by taking a step back to reflect upon the work performed and the future work that is left.

Chapter 2

Secure Compilation to Modern Processors

Publication data

P. Agten, R. Strackx, B. Jacobs, F. Piessens. *Secure compilation to modern processors: extended version.* CW Reports CW619. Department of Computer Science, KU Leuven, Apr. 2012

P. Agten, R. Strackx, B. Jacobs, F. Piessens. "Secure Compilation to Modern Processors". In: *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th.* June 2012, pp. 171–185

Pieter Agten is the main contributor to these works, which were conducted under supervision of Frank Piessens and Bart Jacobs. The second author, Raoul Strackx, was responsible for the performance benchmarks and the development of the PMA prototype, which is based on Fides [113].

2.1 Introduction

High-level programming languages such as Java, C#, ML or Haskell offer protection facilities such as abstract data types, the private field modifier, and module systems. Such abstractions have long been used in programming languages, at least since the 1970s [77, 69]. They were mainly designed to enforce

software engineering principles such as information hiding and encapsulation, but they can also be used as building blocks for providing security properties of programs. For instance, declaring a field private in Java can protect the confidentiality of that field towards less trusted code running in the same Java Virtual Machine.

When such protection features are used for the purpose of security, it is important to maintain the resulting security properties when the program is compiled. The classical way to formalize this notion of secure compilation is *full abstraction* [1]. Roughly speaking, compilation from a source language to a target language is fully abstract if the contextual equivalence of source programs implies the contextual equivalence of corresponding target programs and vice versa. In other words, a fully abstract compiler ensures that a source-level context can distinguish two source programs if and only if a target-level context can distinguish the two corresponding target programs.

Full abstraction (and more specifically the preservation of contextual equivalence) is a good candidate for the definition of secure compilation, because the contextual equivalence of programs can express important security properties, such as confidentiality and integrity properties. For instance, the fact that the value of a static field v in a Java class C is confidential can be expressed by saying that class C is contextually equivalent to a class C' that only differs from C in its value for v. Full abstraction entails the preservation of all security properties that can be expressed using contextual equivalence. We conjecture that this class of security properties consists of all hypersafety properties [28] of the program to compile, but this is only an intuition that has not been formally proved.

Unfortunately, it is notoriously hard to securely compile higher-level languages to lower-level languages. Even the compilation of Java to JVM bytecode, or of C# to the .NET intermediate language is known not to be fully abstract [65] – even if for these cases the source and target languages are relatively close. No state-of-the-art compiler of Java-like or ML-like languages towards machine code on classic Von Neumann computer architectures is even close to fully abstract. As a consequence, any security properties the source program might have are possibly lost towards attackers that can interact with the program at machine code level. Unfortunately, this is a real and important issue, as attacks in practice often rely on injecting machine code into a process' address space [42]. Also, kernel-level malware can attack any process in the system at the machine code level.

Recently, however, some important progress has been made. At CSF 2010, Abadi and Plotkin [3] have shown how address space layout randomization can achieve a probabilistic variant of full abstraction when compiling from a

lambda-calculus variant with abstract public and private memory locations towards a lower-level language in which memory locations are numbers. At CSF 2011, Jagadeesan et al. [62] have extended these results to a more expressive programming language.

The main contribution of our work is the proposal of a new secure compilation technique towards low-level machine code. Instead of relying on randomization as Abadi and Plotkin, or Jagadeesan et al., our compilation technique builds on low-level memory access control techniques. It is inspired by recently developed systems for the fine-grained protection of small pieces of applications such as TrustVisor [72], Fides [113] and Intel's SGX extensions [56]. These systems show that it is possible to efficiently implement relatively fine-grained memory access control on modern processors. In this chapter, we show that such fine-grained memory access control can in turn be used to support fully abstract compilation from a simple imperative programming language to machine code.

The remainder of this chapter is structured as follows. First we give an informal overview of our high- and low-level languages and our compilation scheme in Section 2.2. In Section 2.3, we introduce our prototype implementation. We then reflect upon our approach in Section 2.4, and discuss related work in Section 2.5. Finally, we provide a summary of the chapter in Section 2.6. In the next chapter, we will provide a formal definition of our languages and compiler, and we will prove that the compilation scheme is fully abstract.

2.2 Informal overview

This section presents a precise but informal overview of our approach. We first introduce our high-level language, and illustrate by means of examples in that language how security properties can be expressed using contextual equivalence. We then describe the low-level platform with its fine-grained memory access control model. Finally, we describe our compilation scheme. We first describe a basic, straightforward compilation scheme and illustrate that it is not fully abstract by means of counterexamples. We then describe the more involved, fully abstract compilation by discussing how it handles the counterexamples.

2.2.1 High-level language

Our high-level language is a small, single-threaded, procedural language. It supports the basic constructs one would expect of an imperative programming language, including branches, loops and local variables. Indirect function calls

```
module m {
  <(Int,Int)→Unit> listener = null;
  Int value = 0;

  Unit setListener(<(Int,Int)→Unit> l) {
    listener = l;
    return unit;
  }

  Int getValue() {
    return value;
  }

  Unit setValue(Int v) {
    if (listener != null && value != v) {
      listener(value, v);
    }
    value = v;
    return unit;
  }
}
```

Figure 2.1: Example of a module in our high-level imperative language.

are supported through function references (also known as typed function pointers or delegates). For simplicity, the language does not support dynamic memory allocation. The language is type safe; one can prove progress and preservation using standard methods [101].

Each high-level program consists of a number of modules, which should be thought of as compilation units that encapsulate private state. Each module consists of private fields and public functions. The supported base types are *Unit*, *Int* and the function reference type $\langle \overline{U} \to T \rangle$. Figure 2.1 illustrates the high-level programming language by showing an example module that encapsulates a value and notifies a listener through an indirect call whenever this value changes.

Execution of a high-level program starts in the `main` function of the module named c. The main function must be typed $\langle \mathit{Unit} \to \mathit{Int} \rangle$. Execution either ends with an integer result n or gets stuck in an infinite loop. When the main function ends with result n, we say the program as a whole ends with result n.

2.2.2 Contextual equivalence and security properties

Modules encapsulate their private state, and hence the internal representation of a module is hidden from outside of its definition. In our language, the private, internal representation of a module consists of its member variables and the implementation of its functions. Its public, external representation consists of the signatures of its functions.

Having this division between the internal and external representation of a module implies that modules can be equivalent from an external point of view, even though they have a different implementation. In other words, two modules might have a different internal representation, but no third module will be able to distinguish them by calling their functions and inspecting their return values. We call any two such modules *contextually equivalent* and we say that a third module C that tries to differentiate them, is a *test context*.

We can use contextual equivalences between modules to express important security properties, such as the confidentiality and integrity of private variables and the integrity of module invariants. This is illustrated by the following examples (the first two examples are taken from [3] but are adapted to our programming language).

Example 2.2.1. *Expressing confidentiality properties*

```
module m {
  Int secret = 0;

  Int m() {
    secret = 0;
    return 0;
  }
}
```

```
module m {
  Int secret = 0;

  Int m() {
    secret = 1;
    return 0;
  }
}
```

These two programs differ only in the value that they store in the `secret` *field. By saying these modules are contextually equivalent, we are effectively saying that no external module can read or deduce the value of the* `secret` *field.*

Example 2.2.2. *Expressing integrity properties*

```
module m {
 Int zero = 0;

 Int m(<Unit→Unit> cb) {
   zero = 0;
   Unit x = cb(unit);
   if (zero == 0)
     return 0;
   else return 1;
 }
}
```

```
module m {
 Int zero = 0;

 Int m(<Unit→Unit> cb) {
   zero = 0;
   Unit x = cb(unit);

   return 0;

 }
}
```

The left module checks whether the `zero` *field was changed during the callback* `cb(unit)`*, and the right module does not. By saying that these module are contextually equivalent, we are expressing that there is no way for the external code that is called through the callback function to modify the* `zero` *field.*

Example 2.2.3. *Expressing module invariants*

```
module m {
  Int min = 0;
  Int max = 0;

  [...]

  Int m() {
    if (min <= max) {
      return 0;
    } else {
      return 1;
    }
  }
}
```

```
module m {
  Int min = 0;
  Int max = 0;

  [...]

  Int m() {

    return 0;



  }
}
```

By saying these two modules are contextually equivalent, we are expressing that no external module can break the invariant `min <= max`*. This is a more general kind of integrity property on the data encapsulated by an module.*

For the high-level language, these contextual equivalences (and their corresponding security properties) clearly hold, because the only way a high-level test context can interact with another module is through function calls and returns. No high-level context C can distinguish the left module from the right module for any of these three examples.

To efficiently execute a high-level program however, it must be compiled into a lower-level assembly-language program. From the viewpoint of an attacker, this low-level language is much more powerful than the high-level language, because there is no type system to make it safe. For instance, an attacker that can inject code to interact with a compiled program at the low level can read and write arbitrary memory locations. As a consequence, none of the contextual equivalences in these three examples would continue to hold at the low level, and hence also the corresponding security properties are lost.

Our objective in this chapter is to define a fully abstract compilation scheme from the high-level language to a realistic low-level assembly language. The compiler must ensure that if any two high-level modules are contextually equivalent, then so are their corresponding low-level translations. We limit ourselves to the contextual equivalence of single modules, and hence we define a context to be an arbitrary test module C, which can be *linked* to a single module under test M. Linking a context C with a module M yields a program $C|M$. If no context C can distinguish two implementations of M then these two implementations are contextually equivalent. The notion of contextual equivalence can be generalized to talk about multiple modules, which can all interact with each other as well as with the test module, but we leave this generalization for future work; many interesting security properties can already be expressed using single modules.

The intuition behind a fully abstract compiler is that it ensures that any security property that can be expressed using contextual equivalence and that holds at the source-code level of a program, also holds at the low level after compilation. The power of a low-level attacker is hence reduced to that of a high-level attacker, because any vulnerability that can be exploited at the low level is explainable at the source-code level.

2.2.3 Low-level language

Our low-level target language models a Von Neumann computer architecture [81] extended with a mechanism for fine-grained, program counter-based memory access control.

The basic machine model consists of a program counter, a register and flags file, and a memory space. The program counter indicates the address of the instruction under execution. The register and flags file contains general purpose registers, a stack pointer register and two flags `ZF` and `SF`, which are set or cleared by the `cmp` instruction and are used by branching instructions. The memory space maps addresses to words that represent both code and data. The supported instructions are shown in Table 2.1.

Table 2.1: The set of instructions that make up the low-level programming language.

`ld` r_d `[`r_s`]`	Load the word at the address pointed to by r_s into r_d.
`st [`r_d`]` r_s	Store the word value of r_s into the address pointed to by r_d.
`ldi` r_d `i`	Load the immediate value `i` into register r_d.
`add` r_d r_s	Store $(r_d + r_s) \bmod 2^{32}$ into r_d.
`sub` r_d r_s	Store $(r_d - r_s) \bmod 2^{32}$ into r_d.
`cmp` r_1 r_2	Calculate $r_1 - r_2$ and set the `ZF` and `SF` flags accordingly.
`jmp` r_i	Jump to the address pointed to by register r_i.
`je` r_i	If the `ZF` flag is set, jump to the address pointed to by r_i.
`jl` r_i	If the `SF` flag is set, jump to the address pointed to by r_i.
`call` r_i	Push the value of the program counter onto the top of the stack and jump to the address pointed to by r_i.
`ret`	Pop a value from the top of the stack and jump to the popped location.
`halt`	Stop execution with the result in register `R0`.

Table 2.2: Read-write-execute memory permissions enforced by the protection scheme of the low-level language.

from \ to	*Protected*			*Unprotected*
	Entry point	*Code*	*Data*	
Protected	`r - x`	`r - x`	`r w -`	`r w x`
Unprotected	`- - x`	`- - -`	`- - -`	`r w x`

So far, the low-level platform is effectively a highly simplified model of the Intel x86 platform. However, in order to support fully abstract compilation, our low-level model must be extended with a protection mechanism. We propose to use a fine-grained, program counter-based memory access control scheme, inspired by existing low-level memory protection systems [72, 113, 56]. In this scheme, the memory address space is logically divided into *protected* and *unprotected* memory, and the former type of memory is further divided into a *code* and a *data* section. Within the code section, a variable number of memory addresses are designated as *entry points*. These addresses are the only points through which execution of code in protected memory can start. Table 2.2 summarizes the access control rules enforced by the protection mechanism. The size and location of each of the memory sections and the location of the entry points are specified by a *memory descriptor*, which can be considered a configuration structure for the low-level language. In Section 2.3 we show how this memory access control scheme can be implemented efficiently on modern commodity hardware.

Execution of a low-level program starts at the first address of unprotected memory. It either ends with an integer result or gets stuck in an infinite loop. To end with a result c, the `halt` instruction must be executed with register `R0` containing the value c. If an invalid memory access attempt is made, the value 0 is placed in register `R0` and execution is halted.

2.2.4 Compilation

We now get to our main result, the fully abstract compilation scheme. We describe the compilation of a single high-level module M. This is sufficient to study full abstraction for our definition of contextual equivalence. High-level contextual equivalence was already defined above: two high-level modules are equivalent if no test module can distinguish them. At the low level, two compiled modules are contextually equivalent if no arbitrary machine code placed in the unprotected area of memory can distinguish the compiled modules.

We introduce our compilation scheme in two steps. First we describe a basic, straightforward compilation that places the code and data of the compiled module in the protected memory area and configures the entry points such that control flow can only enter at the start of each function. This scheme is sound, in the sense that two non-equivalent high-level modules will be compiled into two non-equivalent low-level modules. It also provides some basic protection; for instance, the low-level context cannot just scan memory to find the values of module fields, as this is prevented by the low-level memory access control scheme.

However, we will show by means of counterexamples that the basic scheme fails to be fully abstract. These counterexamples then motivate the final definition of our compilation scheme, for which we prove full abstraction in Chapter 3.

Basic compilation

The compilation of a high-level module M results in a low-level module $M{\downarrow}$, consisting of a partial memory space and a memory descriptor. We should prevent the low-level context from being able to distinguish two modules just by their size, so a constant amount of memory is reserved for each translated module, independent of the actual memory space required. The translated module will be placed in protected memory and the memory descriptor divides the reserved space equally over the code and the data section. The compiler assumes the stack pointer register is set up by the context and is pointing to free space in unprotected memory.

The compilation process consists of translating each field and each function of the input module. To prevent a low-level module from being distinguished by the order of its functions in memory, all functions are first sorted alphabetically. Fields and functions are then given a unique index number starting at 0, based on their order of occurrence. Parameters and local variables are given a function-local index number. For a field v_i, one word of memory is reserved at the ith memory address of the data section. Integer-typed constants are translated to their corresponding numeric value. Unit-typed constants are translated to 0 and the **`null`** function reference is translated to the highest address `0xFFFFFFFF`.

To translate a function body, the compiler processes each high-level statement in turn, translating it into a list of instructions that performs the corresponding operation. Registers `R0` to `R3` are used as general working registers and return values are passed through `R0` as well. The first eight parameters are passed through registers `R4` to `R11` and additional parameters are spilled onto the run-time stack. A prologue is prepended to each translated function body and an epilogue is appended to it. The prologue allocates and initializes a new activation record on the stack, which contains the function's local variables and parameters. The epilogue deallocates this activation record when the function is done. This code is placed in free space in the code section.

In addition to translating each function's body, an entry point is generated for each function as well. The entry point for function f_i is placed at address $(i + 1) * 128$ of the code section. The offset of 128 memory words is chosen arbitrarily, with the only condition that there is enough space between entry points to perform a number of simple operations, as will be described in Section 2.2.4. The code at each entry point consists of two parts: (1) a call to the function's body and (2) a return instruction. When the call to the body returns, the return instruction will simply return control to the location from which the entry point was called.

Because the low-level language allows protected memory to be entered only through one of the entry points, an additional *return entry point* is generated at the first address of the protected code section, to support returning from calls from the module to the context. We name such calls *outcalls*. To perform an outcall, first the actual return address is placed on the stack, followed by the address of the return entry point. Control is then transferred to the context by a `jmp` instruction. When the context returns from the outcall, control will first be transferred to the return entry point, which will then subsequently return back to the actual return address within the caller function.

The compilation scheme as described above ensures that a module is always exited through an outcall, or through the return instruction at the end of an entry point. Therefore, we name the end of each entry point an *exit point*.

Limitations of the basic compilation scheme

The compilation scheme defined so far is not fully abstract, as illustrated by the counterexamples below.

Example 2.2.4. *Stack security*

```
module m {
  Int secret = 0;

  Int f(<Unit→Unit> cb)
  {
    Int x = secret;
    Unit y = cb(unit);
    return 0;
  }
}
```

```
module m {
  Int secret = 1;

  Int f(<Unit→Unit> cb)
  {
    Int x = secret;
    Unit y = cb(unit);
    return 0;
  }
}
```

These two high-level modules are contextually equivalent, because the only difference is the value of the `secret` *field and this value is never exposed to the context. The low-level translations of these modules are not contextually equivalent, however, because an attacker can read the value of* `x` *during the callback* `cb(unit)`*. An attacker is able to do this, because local variables are placed on the runtime stack (which is in unprotected memory) as part of a function call, and a low-level attacker can read all unprotected memory. Since the variable* `x` *contains the value of* `secret`*, the attacker can distinguish the two modules.*

The above example shows that the current compilation scheme does not entail the confidentiality or integrity of the runtime stack, and this enables attackers to read and write local variables of the module under test. Attackers can use this vulnerability to read secrets from the stack, similar to a buffer-overread attack [115], or they can even tamper with control flow by overwriting a return address, similar to a classic return address clobbering attack [42].

Example 2.2.5. *Illegal function pointers*

```
1  module m {                                  module m {
2    Int v = 1;                                  Int v = 1;
3
4    Int f(<Unit→Unit> cb)                       Int f(<Unit→Unit> cb)
5    {                                           {
6      Unit x = cb(unit);                          Unit x = cb(unit);
7      v += 1;                                     v -= 1;
8      v -= 1;                                     v += 1;
9      return v;                                   return v;
10   }                                           }
11 }                                         }
```

These two high-level modules are contextually equivalent, because in both modules the function f *always returns 1. A low-level attacker can differentiate their low-level translations, however, by giving the address of the instruction corresponding to line 8 as the callback address* cb*. In this case, the left module will decrement* v *without first incrementing it, while the right module will increment* v *without first decrementing it. This will result in* v *having a value of* 0 *in the left module and* 2 *in the right. This attack is similar to a real-world return-oriented programming attack [106].*

Example 2.2.6. *Information leakage*

```
module m {                                   module m {
  Int f() {                                    Int f() {
    Int x = 0;                                   Int x = 1;

    if (x == 0) {                                if (x == 0) {
      return 0;                                    return 0;
    } else {                                     } else {
      return 0;                                    return 0;
    }                                            }
  }                                            }
}                                            }
```

These two high-level modules are contextually equivalent, because in both modules the function f *always returns 0. A low-level attacker can differentiate their translations, however, due to the equality test in the condition of the if-statement. This test sets the* ZF *flag in the first module and clears it in the second. This example illustrates that the flags register can leak information. Information can also be leaked through the general purpose registers* R0 *to* R11 *or through the stack pointer register* SP*.*

Example 2.2.7. *Illegal program values*

```
module m {
  Unit f(Unit x) {
    return unit;
  }
}
```

```
module m {
  Unit f(Unit x) {
    return x;
  }
}
```

These two high-level modules are contextually equivalent, because the only legal value of type Unit is `unit`*. The low-level translations of these modules are not contextually equivalent, because a low-level attacker can supply any 32-bit value for parameter* `x` *when calling* `f`*.*

As there is no real purpose for having `Unit`-typed function parameters, the above example might seem to be contrived, without ever occurring in the real world. However, this problem is actually similar to a full abstraction failure for the .NET C# compiler reported by Kennedy [65], where the boolean type is two valued in C# but is byte valued in the .NET virtual machine.

Secure compilation

In this section, we describe an enhanced compilation scheme, which adds a number of runtime checks at strategic places in the protected module's code, to correct the full abstraction failures described above. When these runtime checks detect that the context is performing an action that cannot possibly correspond to some action expressible in the high-level language, they will end execution with return value 0. We will show in Chapter 3 that this enhanced compilation scheme is fully abstract.

Stack security The compiler must ensure the confidentiality and integrity of variables and control structures on the run-time stack. Instead of storing the entire stack in unprotected memory, it is split into an unprotected stack in unprotected memory and a secure stack in the data section of protected memory. The protected module places its activation records exclusively on the secure stack, where they are protected from the context.

At each low-level entry point, the stack must be switched to the secure stack and the spilled parameters for the function call (if any) must be copied from the unprotected to the secure stack. At each exit point, the stack must be restored to its previous address in unprotected memory. To implement these stack switches, the compiler uses a *shadow stack pointer* variable in the data

section. It is initialized to a fixed address somewhere in the middle of the protected data section, which is the base address of the secure stack. At each entry and exit point, code is added to swap the value of the stack pointer register with the value of this shadow field.

An outcall from the protected module to unprotected memory is performed by first pushing the actual return address onto the secure stack. Next, the stack must be switched to the unprotected stack and the address of the return entry point and any spilled parameters must be pushed onto it. Control can then be transferred to the context by a jump. When the outcall returns, control will first be transferred to the return entry point, which switches back to the secure stack and subsequently returns back to the actual return address in the original protected function. Because data is written to the unprotected stack as part of making an outcall, the compilation scheme must ensure that the location of the unprotected stack (i.e., the value of the `SP` register) is valid before any stack modifications are performed. That is, code must be added to check that the address of the unprotected stack lies outside of the protected memory region, for otherwise parts of protected memory might get overwritten. If this check fails, it means the context is trying to tamper with the protected module and hence execution should be terminated by placing the value 0 into `R0` and executing the halt instruction.

To prevent the context from tampering with control flow by jumping to the return entry point when there is no outcall to return from, the compiler initializes the first location of the secure stack to the address of a procedure that writes 0 to `R0` and then halts execution. The return entry point will jump to this address if it is called when there is no outcall to return from.

Because the first half of the data section is now reserved for the secure stack, the memory space for a field v_i will now be located at the ith address of the *second half* of the data section. Figure 2.2 illustrates this memory layout. Notice that this layout also ensures that an overflow of the secure stack will result in a memory access violation, because the code section is non-writable.

Illegal function pointers The compilation scheme must ensure control flow integrity on jumps from the protected module to an externally supplied address. Such a jump occurs at each indirect call and at each exit point. For an indirect call, a valid destination address is either (1) an address outside of the module's memory bounds, or (2) the address of one of the module's own functions with a correct signature. For an exit point, only addresses outside of the module's memory bounds are valid. A call or return to the address `0xFFFFFFFF` is also not allowed, because it corresponds to the `null` function reference.

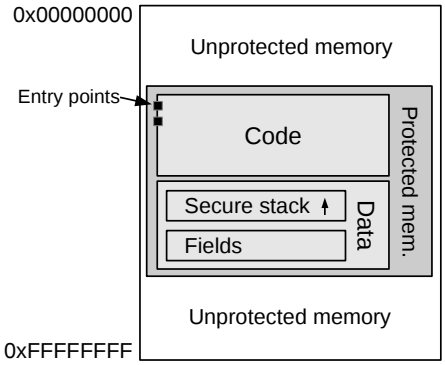


Figure 2.2: Memory layout of the enhanced secure compilation scheme.

The compiler will add runtime checks for these conditions at each indirect call and exit point. A check for the first type of addresses is straightforward to implement, while a check for the second type of addresses is more complicated, because it requires function signature information to be available at runtime. An alternative solution is to simply forbid indirect calls from a module to itself in the high-level language. In that case, the run-time check for illegal function pointers must only check that each indirect call destination lies outside of the protected module's memory bounds. We select the alternative solution, because it does not pose any fundamental restrictions on function calls, since any indirect call to a local function `f` can be replaced by an indirect call to a wrapper function in the context that calls `f`.

Information leakage In the high-level language, the only way for two modules to communicate, is through function calls, function arguments and return values. The compiler must ensure that a low-level attacker cannot use any other communication channels, as this might leak information that should be kept private to the protected module.

The low-level language inherently provides three ways to exchange information: (1) through unprotected memory, (2) through the register and flag file, and (3) by jumping to or calling memory locations. The first method is already restricted, because the only data that is exchanged through unprotected memory consists of spilled function arguments and return values that are copied from the unprotected stack to the secure stack and vice versa. Since this copying is controlled by code in the protected module (generated by the secure compiler), no information other than the information available at the high level can be leaked through unprotected memory. The compiler must constrain the other

communication methods as follows:

- The flags register must be cleared at each outcall and exit point.
- Every general purpose register except `R0` must be cleared at each exit point.
- Every general purpose register not used for passing a parameter must be cleared at each outcall.

Note that the `SP` register does not convey any module-private information, because it is restored to the location of the unprotected stack whenever control leaves the protected module. The above measures ensure that the only information that can be observed from the protected module by the context is also available at the high level.

Illegal program values The compiler must ensure that all low-level memory locations corresponding to high-level fields and variables of the protected module contain only values for which there is a corresponding high-level value of the correct type. The only type in our programming language for which this could be problematic is *Unit*, because it is the only type for which there is no corresponding high-level value for every possible 32-bit low-level value. In particular, `unit` is the only valid high-level value, for which the corresponding low-level value is 0. Hence, the compiler must add a runtime check at each entry point, to ensure that the value of any *Unit*-typed parameter is 0. The same check is added at each outcall to a function with return type *Unit*. If the check fails, the value 0 is placed into register `R0` and the `halt` instruction is subsequently executed.

2.3 Implementation

Our secure compiler relies on the fine-grained, program counter-based memory access control scheme described in Section 2.2.3. A prerequisite for the practical application of this compiler is that this memory access control scheme has an efficient real-world implementation. Efficiency is important, as the main motivation to compile a language is performance; if one does not care about performance, one can simply interpret the high-level program, which is safe assuming the interpreter and underlying operating system are safe. At least three types of implementations are possible: (1) a hardware implementation, (2) a software implementation based on the virtualization support offered by modern processors and (3) a software kernel-level implementation.

An important difference between these three types is the size of the trusted computing base (TCB). A small TCB gives better assurance that there are no vulnerabilities in the implementation that could be exploited to bypass the access control scheme. Recent research [114, 40, 85, 56] has proposed hardware modifications to processors designed for use in embedded systems, to implement a memory access control scheme that is very similar to the one required by our compiler. Such hardware implementations have a very small TCB, since only the hardware itself needs to be trusted. Unfortunately, these modified processors are not yet available for end-consumers at the time of writing, although [56] is expected to be integrated into commodity off-the-shelf processors soon. Conversely, a virtualization-based implementation is supported by currently available commodity hardware and, as we will show below, can also achieve good performance while having a small TCB. A kernel-level implementation does not require virtualization support and hence can even run on older hardware, but this solution would include the entire kernel in its TCB. Given that kernel-level malware is a realistic threat on internet-connected computers today, a kernel-level implementation will most likely not provide sufficient security. This is also an additional reason (besides performance) of why an interpreter would not provide a good solution.

Given these conditions, we followed the second implementation strategy, which is inspired by implementation techniques used in other security architectures that support fine-grained isolation of pieces of application logic [74, 72]. The key idea is to build on the virtualization support offered by current-day commodity hardware. The essence of the memory protection scheme is that the permissions for accessing certain regions of memory depend on the current value of the program counter, as shown in Table 2.2. However, as can be seen from this table, the memory permissions change only when the protected code region is entered or exited. We can trap such entries and exits using a small hypervisor, and reconfigure the standard hardware memory access control unit (MMU) as necessary. We first describe the overall architecture of this implementation, then report on the size of its TCB and finally we provide performance benchmarks.

2.3.1 Architecture

Our prototype implementation is based on a small hypervisor that runs two virtual machines, called the Legacy VM and the Secure VM (see Figure 2.3). Both VMs have the same view of physical memory, but have different memory access permission configurations.

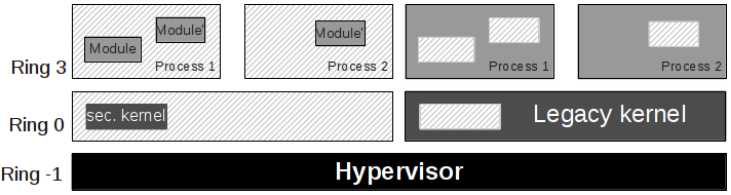


Figure 2.3: The architecture of our prototype is based on two virtual machines: the Legacy VM and the Secure VM.

Legacy VM

The Legacy VM executes all legacy applications and other code in unprotected memory. Using virtualization techniques, this virtual machine is able to execute commodity operating systems and legacy applications without any modification. From the point of view of the Legacy VM, the only difference compared to running on bare hardware is that certain memory locations are inaccessible. More specifically, two memory regions are inaccessible to the Legacy VM: (1) the memory region reserved for the hypervisor and (2) the *protected* memory region as defined in our low-level machine model. Whenever an access to these memory locations is attempted from the Legacy VM, execution traps to the hypervisor.

Hypervisor

The hypervisor serves two simple purposes. First, it offers a coarse-grained memory protection: it prevents *any* code executing in the Legacy VM from directly accessing protected modules (as discussed above) and it prevents both the Legacy VM and the Secure VM from accessing the hypervisor itself.

Secondly, the hypervisor implements a simple scheduling algorithm. When the Legacy VM calls an entry point in the protected module, execution traps to the hypervisor, which then schedules the Secure VM. Execution only returns to the Legacy VM when the protected module returns or performs an outcall to unprotected memory.

Secure VM

The Secure VM can access all memory, with the exception of the memory containing the hypervisor. The fine-grained memory access control mechanism

Table 2.3: The TCB of our PMA prototype consists of only 7K lines of C and assembly code.

Trusted Computing Base (Lines of Code)			
VMM	***Security kernel***	***Shared***	***Total***
1,045	1,947	4,167	7,159

is implemented by a security kernel running in this VM. When a request is received from the hypervisor to execute a function in the protected module, the requested entry point is checked against a list of valid entry points provided by the protected module's memory descriptor. If this check passes, the MMU is set up to allow memory access to the protected module's memory region and execution then proceeds from the entry point that was called. The MMU is set up to disallow execution of unprotected code, such that a page fault will be generated when execution tries to jump back out of the protected module. This page fault will be handled by the security kernel, which will let execution return to the Legacy VM.

2.3.2 Trusted Computing Base

An important factor for the security assurance provided by this system is the size of its TCB. Table 2.3 shows the code size of the different parts of our prototype's TCB, as measured by SLOCCount [120]. Only the hypervisor (VMM) and the security kernel are trusted. They contain 1,045 and 1,947 lines of C and assembly code respectively, and they share an additional 4,167 lines of code. This totals the size of the implementation of the TCB to only 7,159 lines of code, which is at least 4 orders of magnitude less than the TCB of a typical operating system [86].

2.3.3 Performance

We performed two benchmarks to quantify the performance of the memory access control implementation. First, we measured the impact on the overall system. Next, we measured the cost of transitioning between unprotected and protected memory.

All our experiments were performed on a Dell Latitude E6510, a mid-end consumer laptop equipped with an Intel Core i5 560M processor running at 2.67 GHz and 4 GiB of RAM. Due to limitations of our prototype, we had to disable all but one core in the laptop's BIOS. An unmodified version of

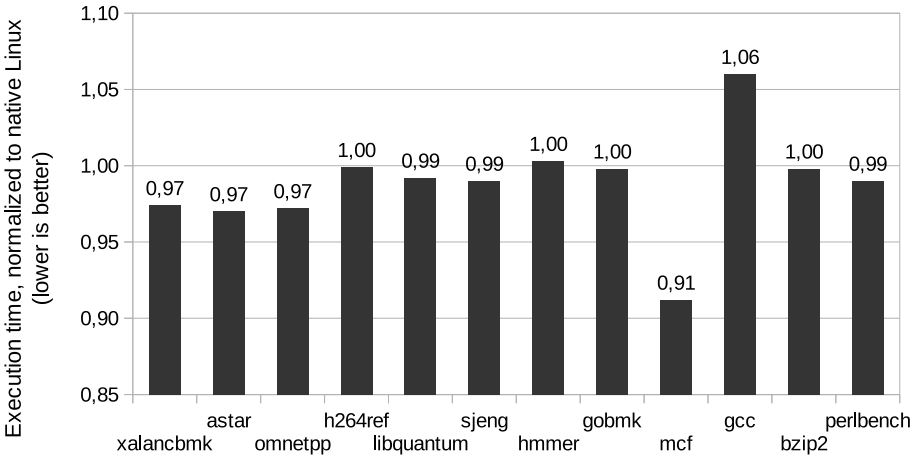


Figure 2.4: The SPECint 2006 benchmarks indicate our prototype implementation has a very low overhead on legacy applications running in unprotected memory.

KUbuntu 10.10 running the 2.6.35-22-generic x86_64 kernel was used as the operating system.

System-wide performance cost

Our implementation uses a small hypervisor, which affects the performance of both the protected module as well as all legacy code. To measure the performance impact of the hypervisor on legacy applications running in unprotected memory, we ran the SPECint 2006 benchmarks. Figure 2.4 displays the results, which show that all applications have an overhead of less than 3.28%, with the exception of the `mcf` application (10.36%). We attribute the performance increase of gcc to caching effects.

As our implementation does not require any computation when the protected module is not under execution, this performance overhead can be attributed completely to the hardware virtualization support. We expect that as this support matures, performance overhead will be reduced further. Note that our hypervisor can be unloaded when it is no longer required, reducing the overhead to 0%.

Table 2.4: Overhead of executing a protected module call vs. executing a kernel-level driver call (in μs).

	Module	***Driver***	***Overhead***
Entry	4.35	0.05	81×
Round trip	6.58	0.07	88×

Micro benchmarks

To measure the impact of crossing the protected/unprotected memory boundary, we implemented a *PingPong* protected module that immediately returns control back to the caller after being called. Using a hardware high-frequency timestamp, we calculated both the time to enter the module as the round trip time. Each test was executed 100,000 times. The results (see Table 2.4) show an overhead of 81× and 88× respectively, compared to a similar module implemented as a driver. Hence, a transition between the two protection domains in our system is about 85 times slower than calling a driver in the operating system. This overhead is mainly caused by the fact that each time the protected/unprotected memory boundary is crossed, a different virtual machine needs to be scheduled. This measurement is an upper bound: as the hardware implementations of virtualization mature, the performance cost of VM transitions will decrease further. Nevertheless, even this upper bound seems small enough to lead to a negligible overhead on the overall application when the protected module is relatively large. For instance, a sensible application design would be to put cryptographic code in the protected module. Calls to this crypto module would cost a few microseconds of performance overhead, which is negligible if the cryptographic operations are computationally intensive.

Conclusions

Our benchmarks indicate that our fine-grained, program counter-based memory access control scheme can be implemented efficiently on commodity hardware. At the same time, it should be clear that the main contribution of this text lies in the description of the secure compilation scheme, and a mature implementation with rigorous micro and macro benchmarks is out of scope.

2.4 Discussion

In this section, we first discuss our choice of using full abstraction as the definition of secure compilation. We then discuss limitations of our high- and low-level languages.

2.4.1 Security by fully abstract compilation

As illustrated by the examples in Section 2.2.2, full abstraction can be used as the definition of secure compilation because the preservation of contextual equivalence expresses important security properties. Full abstraction ensures that programs that are safe at the source-code level remain safe after compilation. Though, one could argue that full abstraction is too restrictive as a definition of secure compilation. For instance, sorting functions alphabetically before compilation to hide the order of a module's functions in memory is required to achieve full abstraction, yet unnecessary if we are only interested in providing confidentiality and integrity of data and in maintaining control flow integrity.

There are two ways to deal with this problem: (1) we can use an entirely different definition for secure compilation that is not based on full abstraction or contextual equivalences, or (2) we can adjust our source and/or target language to better match our desired definition of security. As an example of the latter, consider again the issue described above, where we unnecessarily hide the order of functions in memory. We could add an operation to the high-level language that, when given two functions of a module, returns whether or not the first one is defined before the second one. This would make the high-level language more powerful, without compromising its safety, and it would do away with the need to hide the order of the functions at the low level in order to achieve full abstraction.

2.4.2 Language limitations

The high-level language as described in this paper has a number of limitations as compared to modern programming languages. Most noticeably, it does not have support for multiple interacting modules or dynamic memory allocation. These limitations were removed in the works of Patrignani et al. [99, 96, 94], which extend the compilation scheme described in this chapter. Further extensions, such as support for dynamic dispatch and exceptions, are also discussed in these works.

Multiple modules

Extending our languages to support multiple interacting modules that are part of a single trust domain is straightforward. As these modules trust each other, they can be placed together in protected memory and can share a single secure stack and return entry point.

However, in a more realistic situation, each module could be in its own trust domain, i.e., each module only trusts itself. This situation is more complicated, as each module would require its own protected memory area, including its own secure stack. The memory access protection scheme would have to take this into account and a number of changes to the compilation scheme would have to be made as well. For instance, the stack switch on entry and exit of a module would have to be modified, because spilled parameters and return addresses of calls between two protected modules cannot be written on either of their stacks, as neither stack is accessible by both modules. Furthermore, new attack vectors might exist due to the increased complexity of multiple interacting modules. For instance, an attacker could try to make two low-level modules interact in ways that could never occur at the high level, leading to undefined behavior that is dependent on the specific implementation of a module. New compiler measures would have to be installed to protect against these new attacks.

Dynamic memory allocation

For simplicity, our high-level language does not support dynamic memory allocation. However, Patrignani et al. have developed a fully abstract compilation scheme for an object-oriented source language with support for dynamic allocation of objects [99, 96, 94], as an extension of this work.

Support for dynamic memory allocation is implemented in these works as follows. A predetermined amount of protected data memory is reserved as heap space, similar to the memory reserved for the secure stack. Because this memory is protected, an attacker will be unable to access heap records directly. This is necessary to maintain full abstraction, since direct access to heap records would violate the high-level encapsulation properties of an object. Construction of a new object of a class of the protected module is performed by a factory method in the protected code section.

Since objects are now created dynamically, references to these object will need to be passed around for other objects to access them. Supporting such references without breaking full abstraction is nontrivial, since they might leak information about the order in which objects are allocated and possibly about their size, thereby breaking full abstraction. Patrignani et al. solve this problem by

maintaining a hidden object identity map in the protected module. This data structure maps a unique number (an object identifier) to each protected object passed to the context. Instead of passing the memory address of the object to the context, as would be the case in a standard compiler, the generated object identifier is passed instead. By choosing the identifiers such that the first object exposed to the context receives identifier 1, the second exposed object receives identifier 2, etc., no information about the order or size of object allocations is leaked. Other changes to the compilation scheme that are required for maintaining full abstraction, include more thorough runtime type checks of argument and return values of class methods.

2.5 Related work

There is a large amount of research on secure compilation to machine code, but in most works the emphasis is on hardening the compilation of unsafe languages such as C to protect against exploitation of the compiled program by feeding it malicious input. Younan et al. [122] give an extensive survey. Some notable examples that can provide formal guarantees include control-flow integrity (CFI) [2], and obfuscation [103].

In our work, attackers can do much more than just supply malicious input; attackers can execute arbitrary code in the low-level language. The idea to formalize secure compilation to lower-level languages as full abstraction (and thus protect against this more powerful type of attacker) was pioneered by Abadi [1]. In that paper, Abadi illustrates this idea in two settings: the compilation of Java to bytecode, and the implementation of secure channels in the pi-calculus by means of cryptographic protocols. The second setting, proving the soundness of cryptographic implementations, has received a lot of attention but it is less related to the work reported in this paper. The first setting, secure compilation to lower-level languages, was studied in the context of compilation to .NET bytecode by Kennedy [65]. However, for compilation to low-level code with natural number addressing for memory, only very recently Abadi and Plotkin [3] have shown that Address Space Layout Randomization (ASLR) is a sufficiently strong software protection technique to achieve fully abstract compilation in a probabilistic sense. Jagadeesan et al. [62] extended the results of Abadi and Plotkin to a richer programming language with dynamic memory allocation, first class and higher order references and unstructured control flow. Instead of relying on randomization as the fundamental protection mechanism, our work shows that program counter-dependent low-level memory access control is also a sufficiently strong protection mechanism to achieve fully abstract compilation. Another notable application of full abstraction can be

found in the recent work of Fournet et al. [47], who present a fully abstract compiler from an ML-like language with higher-order functions and references to JavaScript.

The fact that fine-grained memory isolation can be achieved with reasonable performance overhead, was shown by a second line of related research that influenced our work. Several authors have proposed security architectures with a closely related low-level isolation mechanism. While there are significant differences between these security architectures and our own prototype (in for instance the implementation techniques used), their access control mechanisms are comparable. For example, Nizza [110], Flicker [74], TrustVisor [72] and SICE [15] also provide isolation of small pieces of application logic. The memory access control mechanisms enforced by these architectures are a special case of our model, where accesses to unprotected memory from modules is not allowed. P-MAPS [107], Fides [113], Sancus [85] and Intel's SGX [56], do allow access to unprotected memory. All these papers are systems papers: they report on working systems without providing formal security guarantees. It is likely that several of these proposed security architectures could be low-level target platforms for a secure compilation process as we developed in this paper. In addition, these papers provide evidence that the low-level memory access control we need in our model is efficiently implementable on today's computer platforms.

Finally, as reported above, the secure compilation scheme described in this chapter has been extended by Patrignani et al. [99, 96, 94], to support dynamic memory allocation, dynamic dispatch, exceptions and other advanced programming language features.

2.6 Summary

Protection facilities in high-level programming languages can be used to enforce confidentiality and integrity properties for the data managed by one component towards other (potentially malicious) components that it interacts with. Maintaining such security properties after compilation to a low-level language requires some protection features in the low-level language as well. Randomization is one such protection feature that is known to be strong enough to support secure compilation. We have shown in this paper that program counter-based memory access control schemes, such as those offered by state-of-the-art protected module architectures, are also suitable as low-level protection mechanism. We developed a model of such a low-level platform, and have shown how a procedural high-level language can be securely compiled to this platform. We have also shown that the low-level platform is realistic, in the sense that it

can already be implemented on today's commodity computers with acceptable performance. We believe that the presented secure compilation technique can help address the pervasive threat of kernel-level malware that we still see today.

Chapter 3

Secure Compilation to Modern Processors: Formalization

Publication data

P. Agten, R. Strackx, B. Jacobs, F. Piessens. *Secure compilation to modern processors: extended version.* CW Reports CW619. Department of Computer Science, KU Leuven, Apr. 2012

P. Agten, R. Strackx, B. Jacobs, F. Piessens. "Secure Compilation to Modern Processors". In: *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th.* June 2012, pp. 171–185

Pieter Agten is the main contributor to these works, which were conducted under supervision of Frank Piessens and Bart Jacobs. The second author, Raoul Strackx, was responsible for the performance benchmarks and the development of the PMA prototype, which is based on Fides [113]

3.1 Introduction

In this chapter, we formalize the compilation scheme defined in the previous chapter and prove that it is fully abstract. We have divided our efforts in the most

efficient way between developing a thorough implementation of the compiler (see Appendix A) and rigorously formalizing it and its security properties. For this reason, the theorems and lemmas that we believe contribute most to convincing the reader of the correctness of our results are proved in full detail, while some details on the proofs of the more straightforward statements have been omitted. To make this difference in rigor explicit, the less rigorous proofs are named *proof sketches* instead of proofs.

The further outline of this chapter is as follows. In section Section 3.2, we give a formal definition of contextual equivalence. We then formalize our high- and low-level programming languages in Section 3.3. In Section 3.4, we describe our compiler, and in Section 3.5 we introduce program execution traces. Finally, we give our full abstraction proof in Section 3.6 and provide a summary of the chapter in Section 3.7.

3.2 Contextual equivalence

We will consider programs that consist out of two modules: a test context C and a test subject M. Programs of this form are written as $C|M$. The context is always modeled as a single module, because from the viewpoint of the subject, the entire context is a (potentially malicious) black box of which the internal structure is irrelevant.

The execution of a program $C|M$ is written as $C|M \longrightarrow^* n$ if it ends with integer result n and as $C|M \longrightarrow^* \lightning$ if it diverges. We will now first formally define what it means for two modules to be contextually equivalent.

Definition 3.2.1 (Contextual equivalence)**.** *For any two modules M_1 and M_2, we say M_1 and M_2 are contextually equivalent, and write $M_1 \simeq M_2$ iff*

$$\forall C.\ C|M_1 \longrightarrow^* r \iff C|M_2 \longrightarrow^* r$$

This definition holds for both low- and high-level programs, and the program result r can be an integer, the **fault** outcome, or it can be $\lightning$ if the programs do not terminate.

3.3 Language definitions

In this section we formally define the high-level source language and the low-level target language of our compiler. We start with the high-level language in Section 3.3.1 and discuss the low-level language in Section 3.3.2.

$$
\begin{array}{rll}
P & ::= & C|M & \textit{programs} \\
C, M & ::= & \textbf{module}\ m\ \{\overline{V}\ \overline{F}\} & \textit{modules} \\
V & ::= & T\ x = v & \textit{variables} \\
F & ::= & T\ f(T\ x)\ \{\textbf{vars}\ \overline{V};\ S;\ \textbf{ret}\ x\} & \textit{functions} \\
T & ::= & \textbf{Unit} \mid \textbf{Int} \mid \langle T \to T \rangle & \textit{types} \\
S & ::= & x := E \mid m.x := E \mid \textbf{if}\ B\ \textbf{then}\ S\ \textbf{else}\ S \mid & \textit{statements} \\
 & & \quad x := E(E) \mid \textbf{exit}\ E \mid S; S & \\
B & ::= & E = E \mid E < E \mid !\, B & \textit{booleans} \\
E & ::= & x \mid m.x \mid v \mid E + E \mid E - E & \textit{expressions} \\
v & ::= & n \mid m.f \mid \text{unit} & \textit{values}
\end{array}
$$

Figure 3.1: Syntax definition for the high-level language. The metavariable m ranges over module names, x over variable names, f over function names, and n over natural numbers. We implicitly assume all variable and function names are unique.

3.3.1 High-level language

The syntax of the high-level language is defined in Figure 3.1. Figure 3.2 shows the language's typing rules, and Figure 3.3 shows its operational semantics. In these figures, $[\![E]\!]_s$ denotes the evaluation of expression E under the store s (which we assume has a mapping for each of the free variables of E), using the modulo 2^{32} interpretation of the available arithmetic operators. The notation $s[x \mapsto n]$ denotes the partial function that maps x to n and all other arguments y in the domain of s to $s(y)$.

We make one simplification to the formal high-level language compared to the language used in the previous chapter, which is that we require each function to have exactly one parameter. This change simplifies the formal notation and the compiler, because it prevents parameters from having to be spilled onto the runtime stack, while only slightly reducing the expressivity of the language.

Notice that the high-level language does not contain an explicit looping construct such as a while- or for-loop. These constructs have been omitted in favor of

$$\frac{\begin{array}{cc}\mathrm{sig}(M) \vdash C : \mathrm{OK_m} & \mathrm{sig}(C) \vdash M : \mathrm{OK_c} \\ \mathrm{name}(C) \neq \mathrm{name}(M) & \mathrm{name}(C).\mathrm{main} : \langle \mathbf{Unit} \to \mathbf{Int} \rangle \in \mathrm{sig}(C)\end{array}}{C|M : \mathrm{OK}} \text{ T-Prog}$$

$$\frac{M = \mathbf{module}\ m\ \{\overline{V}\ \overline{F}\} \qquad \Gamma \vdash \overline{V} : \mathrm{OK_x} \qquad \Gamma \cup \mathrm{sig}(\overline{V}, m) \cup \mathrm{sig}(M) \vdash \overline{F} : \mathrm{OK_x}}{\Gamma \vdash M : \mathrm{OK_x}} \text{ T-Mod} \qquad \frac{\Gamma \vdash B : \mathrm{OK_x}}{\Gamma \vdash !\, B : \mathrm{OK_x}} \text{ T-Not}$$

$$\frac{\Gamma \vdash v : T}{\Gamma \vdash T\ x = v : \mathrm{OK_x}} \text{ T-VarDecl} \qquad \frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{ T-Var} \qquad \frac{m.x : T \in \Gamma}{\Gamma \vdash m.x : T} \text{ T-Field}$$

$$\frac{}{\Gamma \vdash n : \mathbf{Int}} \text{ T-Int} \qquad \frac{}{\Gamma \vdash \mathrm{unit} : \mathbf{Unit}} \text{ T-Unit} \qquad \frac{m.f : \langle T \to T' \rangle \in \Gamma}{\Gamma \vdash m.f : \langle T \to T' \rangle} \text{ T-FRef}$$

$$\frac{\Gamma \vdash E : \mathbf{Int} \qquad \Gamma \vdash E' : \mathbf{Int}}{\Gamma \vdash E + E' : T} \text{ T-Sum} \qquad \frac{\Gamma \vdash E : \mathbf{Int} \qquad \Gamma \vdash E' : \mathbf{Int}}{\Gamma \vdash E - E' : T} \text{ T-Diff}$$

$$\frac{\Gamma \cup \mathrm{sig}(\overline{V}) \cup \{x' : T'\} \vdash S : \mathrm{OK_x} \qquad \mathrm{sig}(\overline{V}) \vdash x : T}{\Gamma \vdash T\ f(T'\ x')\ \{\mathbf{vars}\ \overline{V};\ S;\ \mathbf{ret}\ x\} : \mathrm{OK_x}} \text{ T-Func}$$

$$\frac{\Gamma \vdash E : \langle T \to T' \rangle \qquad \Gamma \vdash x : T' \qquad \Gamma \vdash E' : T}{\Gamma \vdash x := E(E') : \mathrm{OK_x}} \text{ T-Call}$$

$$\frac{\Gamma \vdash E : \mathbf{Int}}{\Gamma \vdash \mathbf{exit}\ E : \mathrm{OK_c}} \text{ T-Exit} \qquad \frac{\Gamma \vdash B : \mathrm{OK_x} \qquad \Gamma \vdash S : \mathrm{OK_x} \qquad \Gamma \vdash S' : \mathrm{OK_x}}{\Gamma \vdash \mathbf{if}\ B\ \mathbf{then}\ S\ \mathbf{else}\ S' : \mathrm{OK_x}} \text{ T-If}$$

$$\frac{\Gamma \vdash x : T \qquad \Gamma \vdash E : T}{\Gamma \vdash x := E : \mathrm{OK_x}} \text{ T-Assn} \qquad \frac{\Gamma \vdash S : \mathrm{OK_x} \qquad \Gamma \vdash S' : \mathrm{OK_x}}{\Gamma \vdash S;\ S' : \mathrm{OK_x}} \text{ T-Seq}$$

$$\frac{\Gamma \vdash E : T \qquad \Gamma \vdash E' : T}{\Gamma \vdash E = E' : \mathrm{OK_x}} \text{ T-Eq} \qquad \frac{\Gamma \vdash E : \mathbf{Int} \qquad \Gamma \vdash E' : \mathbf{Int}}{\Gamma \vdash E < E' : \mathrm{OK_x}} \text{ T-Lt}$$

$$\mathrm{name}(\mathbf{module}\ m\ \{\ldots\}) = m$$

$$\mathrm{sig}(\mathbf{module}\ m\{\overline{V}\ \overline{F}\}) = \{m.f : \langle T' \to T \rangle \mid T\ f(T'\ x)\ \{\ldots\} \in \overline{F}\}$$

$$\mathrm{sig}(\overline{V}) = \{x :\ T \mid T\ x = v \in \overline{V}\} \qquad \mathrm{sig}(\overline{V}, m) = \{m.x :\ T \mid T\ x = v \in \overline{V}\}$$

Figure 3.2: Typing rules for our high-level language.

$$\frac{C = \textbf{module } c\ \{\overline{V}\ \overline{F}\} \qquad \text{bodies}(C)(\text{c.main}) = (-, \overline{V}'', S, x) \qquad M = \textbf{module } m\ \{\overline{V}'\ \overline{F}'\} \qquad g = \text{init}(\overline{V}) \cup \text{init}(\overline{V}')}{C|M \longrightarrow \text{fc}(C|M) \vdash \langle g, [(c, \text{init}(\overline{V}''))],\ S; \textbf{ret } x\rangle}\ \text{E-Prog}$$

$$\frac{v = [\![E]\!]_{g \cup s}}{\Sigma \vdash \langle g, (m, s)::\overline{s},\ x := E\rangle \longrightarrow \Sigma \vdash \langle g, (m, s[x \mapsto v])::\overline{s},\ \textbf{skip}\rangle}\ \text{E-Var} \qquad \frac{v = [\![E]\!]_{g \cup s}}{\Sigma \vdash \langle g, (m, s)::\overline{s},\ m.x := E\rangle \longrightarrow \Sigma \vdash \langle g[m.x \mapsto v], (m, s)::\overline{s},\ \textbf{skip}\rangle}\ \text{E-Field}$$

$$\frac{[\![B]\!]_{g \cup s} = \text{true}}{\Sigma \vdash \langle g, (m, s)::\overline{s},\ \textbf{if } B \textbf{ then } S \textbf{ else } S'\rangle \longrightarrow \Sigma \vdash \langle g, (m, s)::\overline{s},\ S\rangle}\ \text{E-IfTrue}$$

$$\frac{[\![B]\!]_{g \cup s} = \text{false}}{\Sigma \vdash \langle g, (m, s)::\overline{s},\ \textbf{if } B \textbf{ then } S \textbf{ else } S'\rangle \longrightarrow \Sigma \vdash \langle g, (m, s)::\overline{s},\ S'\rangle}\ \text{E-IfFalse}$$

$$\frac{\text{validCall}(m, E, g \cup s) \qquad [\![E]\!]_{g \cup s} = m'.f \qquad \Sigma(m'.f) = (x', \overline{V}, S, x'') \qquad [\![E']\!]_{g \cup s} = v \qquad s' = \{x' \mapsto v\} \cup \text{init}(\overline{V})}{\Sigma \vdash \langle g, (m, s)::\overline{s},\ x := E(E')\rangle \longrightarrow \Sigma \vdash \langle g, (m', s')::(m, s)::\overline{s},\ S; x := \textbf{ret } x''\rangle}\ \text{E-Call}$$

$$\frac{\neg\text{validCall}(m, E, g \cup s)}{\Sigma \vdash \langle g, (m, s)::\overline{s},\ x := E(E')\rangle \longrightarrow 0}\ \text{E-CallInvalid}$$

$$\frac{s'' = s'[x \mapsto s(x')]}{\Sigma \vdash \langle g, (m, s)::(m', s')::\overline{s},\ x := \textbf{ret } x'\rangle \longrightarrow \Sigma \vdash \langle g, (m', s'')::\overline{s},\ \textbf{skip}\rangle}\ \text{E-Ret}$$

$$\frac{\Sigma \vdash \langle g, \overline{s},\ S\rangle \longrightarrow \Sigma \vdash \langle g', \overline{s}',\ S'\rangle}{\Sigma \vdash \langle g, \overline{s},\ S; S''\rangle \longrightarrow \Sigma \vdash \langle g', \overline{s}',\ S'; S''\rangle}\ \text{E-Seq} \qquad \frac{}{\Sigma \vdash \langle g, \overline{s},\ \textbf{skip}; S\rangle \longrightarrow \Sigma \vdash \langle g, \overline{s},\ S\rangle}\ \text{E-Skip}$$

$$\frac{v = [\![E]\!]_{g \cup s}}{\Sigma \vdash \langle g, (m, s)::\overline{s},\ \textbf{exit } E\rangle \longrightarrow v}\ \text{E-Exit} \qquad \frac{}{\Sigma \vdash \langle g, [(m, s)],\ \textbf{ret } x\rangle \longrightarrow s(x)}\ \text{E-End}$$

$$\text{init}(\overline{V}) = \{x \mapsto v \mid T\ x = v \in \overline{V}\} \qquad \text{init}(\overline{V}, m) = \{m.x \mapsto v \mid T\ x = v \in \overline{V}\}$$

$$\text{validCall}(m, E, s) = \exists m', f.\ (E = m'.f) \vee ([\![E]\!]_s = m'.f \wedge m' \neq m)$$

$$\text{fc}(C|M) = \text{bodies}(C) \cup \text{bodies}(M))$$

$$\text{bodies}(\textbf{module } m\ \{\overline{V}\ \overline{F}\}) = \{m.f \mapsto \text{mbody}(F) \mid F \in \overline{F}\}$$

$$\text{mbody}(T\ f(T'\ x)\ \{\textbf{vars } \overline{V};\ S;\ \textbf{ret } x'\}) = (x, \overline{V}, S, x')$$

Figure 3.3: Small-step operational semantics of our high-level language.

using recursion. Also notice that the protected module is not allowed to use the **exit** statement. This is because allowing the module to do so would break full abstraction, since it makes a module that calls **exit** appear equivalent to a module that enters an infinite loop, at the level of execution traces (see Section 3.5.1). Finally, remember that our language disallows indirect calls between functions of the same module, as discussed in Section 2.2.4.

Typing judgments take the form $\Gamma \vdash K : T$, where Γ is the typing context, K is the programming construct to be typed and T is a type or $\mathrm{OK_x}$ if the construct does not have a type but has been checked to conform to the language's typing rules. The x in $\mathrm{OK_x}$ is a metavariable that is either c while type checking the context or m when type checking the protected module. We make this distinction in order to be able to reject protected modules that use the **exit** statement, while still allowing the context to use this statement.

The transition relation $\longrightarrow$ relates *configurations*, which are four-tuples $\Sigma \vdash \langle g, \overline{s}, S \rangle$, representing the current execution state. The first element of those tuples is the function context Σ, which maps function names to tuples $(x, \overline{V}, S, x')$, containing those functions' parameter name x, local variable declarations $\overline{V}$, body S and return variable name x'. The function context is never modified during execution. The second element g is the global store, which maps module field names to values. The third element $\overline{s}$ is a stack of activation records, each of which is a pair (m, s), where m is a module name and s is a local store mapping variable names to values. The record at the top of the stack is the *current* activation record, which designates the module name and local store of the function under execution. Keeping track of the module name associated with each activation record is not strictly necessary for defining the operational semantics of the high-level language, but it will facilitate the definition of *traces* in Section 3.5.1. The fourth and final configuration element S is simply the program under execution. All inference rules of the language are deterministic, and hence any configuration either diverges or results in exactly one end state.

3.3.2 Low-level language

The low-level language models a basic Von Neumann computer architecture [81], extended with a fine-grained, program counter-based memory access control scheme. Its syntax has already been described in Table 2.1 and its formal semantics are defined in Figure 3.4 and Figure 3.5.

Compared to the model described in the previous chapter, we only make the following simplification. We assume a fixed memory descriptor $\delta = (b, s_c, s_d, n)$, where $b = \texttt{0x1000000}$ is the base address of the protected memory region,

$$\frac{r_0 = \{r \mapsto 0 \mid r \in \mathit{Regs} \cup \mathit{Flags}\}}{C|M \longrightarrow \langle 0, r_0, C|M\rangle}\ \text{E-Prog} \qquad \frac{m(p) = \texttt{halt}}{\langle p, r, m\rangle \longrightarrow r(\texttt{R0})}\ \text{E-Halt}$$

$$\frac{m(p) = \texttt{ld d [s]} \qquad \mathit{validJmp}_\delta(p, \mathit{inc}(p)) \qquad \mathit{validRead}_\delta(p, r(s))}{\langle p, r, m\rangle \longrightarrow \langle \mathit{inc}(p), r[d \mapsto m(r(s))], m\rangle}\ \text{E-Ld}$$

$$\frac{m(p) = \texttt{st [d] s} \qquad \mathit{validJmp}_\delta(p, \mathit{inc}(p)) \qquad \mathit{validWrite}_\delta(r(d),)}{\langle p, r, m\rangle \longrightarrow \langle \mathit{inc}(p), r, m[r(d) \mapsto r(s)]\rangle}\ \text{E-St}$$

$$\frac{m(p) = \texttt{ldi d i} \quad \mathit{validJmp}_\delta(p, \mathit{inc}(p))}{\langle p, r, m\rangle \longrightarrow \langle \mathit{inc}(p), r[d \mapsto i], m\rangle}\ \text{E-Ldi} \qquad \frac{m(p) = \texttt{jmp r}_1 \quad \mathit{validJmp}_\delta(p, r(r_l))}{\langle p, r, m\rangle \longrightarrow \langle r(r_l), r, m\rangle}\ \text{E-Jmp}$$

$$\frac{m(p) = \texttt{add d s} \qquad \mathit{validJmp}_\delta(p, \mathit{inc}(p)) \qquad v = r(d) + r(s) \mod 2^{32}}{\langle p, r, m\rangle \longrightarrow \langle \mathit{inc}(p), r[d \mapsto v], m\rangle}\ \text{E-Add}$$

$$\frac{m(p) = \texttt{sub d s} \qquad \mathit{validJmp}_\delta(p, \mathit{inc}(p)) \qquad v = r(d) - r(s) \mod 2^{32}}{\langle p, r, m\rangle \longrightarrow \langle \mathit{inc}(p), r[d \mapsto v], m\rangle}\ \text{E-Sub}$$

$$\frac{m(p) = \texttt{cmp x y} \qquad \mathit{validJmp}_\delta(p, \mathit{inc}(p)) \quad z = (r(x) = r(y)) \qquad s = (r(x) < r(y))}{\langle p, r, m\rangle \longrightarrow \langle \mathit{inc}(p), r[\texttt{ZF} \mapsto z, \texttt{SF} \mapsto s], m\rangle}\ \text{E-Cmp}$$

$$\frac{m(p) = \texttt{je d} \qquad r(\texttt{ZF}) = \text{true} \quad \mathit{validJmp}_\delta(p, r(d))}{\langle p, r, m\rangle \longrightarrow \langle r(d), r, m\rangle}\ \text{E-JeTrue} \qquad \frac{m(p) = \texttt{je d} \qquad r(\texttt{ZF}) = \text{false} \quad \mathit{validJmp}_\delta(p, \mathit{inc}(p))}{\langle p, r, m\rangle \longrightarrow \langle \mathit{inc}(p), r, m\rangle}\ \text{E-JeFalse}$$

$$\frac{m(p) = \texttt{jl d} \qquad r(\texttt{SF}) = \text{true} \quad \mathit{validJmp}_\delta(p, r(d))}{\langle p, r, m\rangle \longrightarrow \langle r(d), r, m\rangle}\ \text{E-JlTrue} \qquad \frac{m(p) = \texttt{jl d} \qquad r(\texttt{SF}) = \text{false} \quad \mathit{validJmp}_\delta(p, \mathit{inc}(p))}{\langle p, r, m\rangle \longrightarrow \langle \mathit{inc}(p), r, m\rangle}\ \text{E-JlFalse}$$

$$\frac{m(p) = \texttt{call d} \qquad s = r(\texttt{SP}) - 1 \mod 2^{32} \quad \mathit{validJmp}_\delta(p, r(d)) \qquad \mathit{validWrite}_\delta(p, r(s))}{\langle p, r, m\rangle \longrightarrow \langle r(d), r[\texttt{SP} \mapsto s], m[r(s) \mapsto \mathit{inc}(p)]\rangle}\ \text{E-Call}$$

$$\frac{m(p) = \texttt{ret} \qquad s = r(\texttt{SP}) + 1 \mod 2^{32} \quad \mathit{validJmp}_\delta(p, m(r(\texttt{SP})))}{\langle p, r, m\rangle \longrightarrow \langle m(r(\texttt{SP})), r(\texttt{SP} \mapsto s), m\rangle}\ \text{E-Ret}$$

$$\mathit{inc}(p) = p + 2 \mod 2^{32}$$

Figure 3.4: Operational semantics of our low-level assembly language.

$$
\begin{aligned}
\delta &= (\texttt{0x1000000}, \texttt{0x4000000}, \texttt{0x4000000}, 16) \\
\mathit{validJmp}_\delta(p, p') &= (\mathit{internalJmp}_\delta(p, p') \ \vee \mathit{entryJmp}_\delta(p, p') \vee \mathit{exitJmp}_\delta(p, p')) \wedge \\
&\quad\ p' \mod 2 = 0 \\
\mathit{internalJmp}_\delta(p, p') &= (\mathit{unprotected}_\delta(p) \wedge \mathit{unprotected}_\delta(p')) \vee \\
&\quad\ (\mathit{code}_\delta(p) \wedge \mathit{code}_\delta(p')) \\
\mathit{entryJmp}_\delta(p, p') &= \mathit{unprotected}_\delta(p) \wedge \mathit{entryPoint}_\delta(p') \\
\mathit{exitJmp}_\delta(p, p') &= \mathit{protected}_\delta(p) \wedge \mathit{unprotected}_\delta(p') \\
\mathit{protected}_\delta(p) &= \exists b, s_c, s_d.\ \delta = (b, s_c, s_d, -) \wedge b < p \wedge p < (b + s_c + s_d) \\
\mathit{unprotected}_\delta(p) &= \neg \mathit{protected}_\delta(p) \\
\mathit{entryPoint}_\delta(p) &= \exists m \in \mathbb{N}^+, b, n.\ \delta = (b, -, -, n) \ \wedge p = b + m \times 128 \wedge m < n \\
\mathit{retEntryPoint}_\delta(p) &= \exists b.\ \delta = (b, -, -, -) \ \wedge p = b \\
\mathit{validRead}_\delta(p, l) &= \mathit{unprotected}_\delta(l) \vee \mathit{protected}_\delta(p) \\
\mathit{validWrite}_\delta(p, l) &= \mathit{unprotected}_\delta(l) \vee (\mathit{protected}_\delta(p) \wedge \mathit{data}_\delta(l)) \\
\mathit{code}_\delta(l) &= \exists b, s_c, s_d.\ \delta = (b, s_c, s_d, -) \wedge b \leq l \wedge l < (b + s_c) \\
\mathit{data}_\delta(l) &= \exists b, s_c, s_d.\ \delta = (b, s_c, s_d, -) \wedge b + s_c \leq l \wedge l < (b + s_c + s_d)
\end{aligned}
$$

Figure 3.5: Auxiliary definitions for the program counter-based memory access control scheme of our low-level assembly language.

$s_c = \texttt{0x4000000}$ is the size of the protected code region, $s_d = \texttt{0x4000000}$ is the size of the protected data region, and $n = 16$ is the number of entry points in the protected module. Each entry point is spaced 128 words apart, starting at the module's base address. The return entry point is always the first entry point. If fewer than 16 entry points are required, the compiler should ensure that extraneous entry points simply halt the machine. We assume that the compiler returns an error when the fixed number of entry points or the fixed size of the provided memory regions is insufficient for the module under compilation.

Evaluation starts from a program $C|M$, which is the result of linking C and M. We do not give a formal description of the linking procedure, as it would distract from the main purpose of this chapter, but informally it means that (1) the

resulting memory region consists of C, with the memory region from `0x1000000` to `0x8FFFFFFF` overwritten by the contents of M, and (2) any unresolved call targets are replaced by the memory address of the corresponding function.

The initial program $C|M$ transitions into a three-tuple $\langle p, r, m\rangle$ that models a processor, where p is the program counter, r is the register and flags file, and m is the main memory. The register and flag file contains 12 general purpose registers `R0` to `R11`, a stack pointer register `SP`, and two flags: the zero flag `ZF` and the sign flag `SF`. The stack is used for return addresses and local variables and grows down, i.e. from high to low memory addresses. The flags are set or cleared by the `cmp` instruction and are used by branching instructions. The memory space is a function mapping positive natural numbers to words which represent both data and code. Addresses, words and registers are 32 bits wide, instructions are 64 bits wide and memory is addressed in multiples of 32 bits. Any conditional or unconditional jump target must be a multiple of 2, such that it is impossible to jump to the middle of an instruction.

Initially the program counter and all flags and registers are 0, and the main memory is taken from the starting program $C|M$. From then on, the processor will start processing instructions, thereby modifying the registers, flags and memory space along the way. The processor implements the program counter-based memory access control scheme by checking whether each memory read, memory write and control flow jump (even "jumps" from one instruction to the next) adheres to the permissions of Table 2.2. If at any point in the execution there is no transition rule from Figure 3.4 that applies to the current state, we assume that the processor identifies this as a fault. In that case, we write $\langle p, r, m\rangle \longrightarrow \textbf{fault}$.

In this work, we do not consider side channel attacks that could trivially break full abstraction in practice. This is reflected in the low-level processor model by the lack of features that would provide such side channels, such as a real-time clock, caches and I/O devices.

3.4 Compiler

Recall that the compilation of a high-level module M results in a low-level module $M{\downarrow}$. We have formalized this compilation process in the form of an OCaml implementation, which can be found in Appendix A. The full source code, including a number of examples, can be downloaded at `http://people.cs.kuleuven.be/~pieter.agten/csf2012/`. The compiler takes as input a high-level module in a slightly different format than presented in Section 3.3.1. The syntax of the high-level modules taken as input by this compiler looks more

like a typed assembly language than the syntax presented in Section 3.3.1. This should be considered only a cosmetic change that doesn't change the expressivity or safety of the language, but lets the compiler focus on the essentials of the translation (i.e., ensuring that the compilation is fully abstract), instead of on basic compilation issues that you would find in any compiler.

3.5 Traces

Inspired by Jeffrey and Rathke's technique of using program *traces* for proving full abstraction for Java Jr [63], we now introduce traces for our high- and low-level programming languages. Traces let us reason about executions at a higher level of abstraction, by showing only the interactions between a program's context C and its subject M, and hiding the internal computation in either component. For instance, if we have a high-level execution in which the context C first performs a number of internal computations, then calls the function $m.f$ with argument v in the module M, which subsequently does some internal computations and then returns the value v', then the corresponding high-level trace (so far) will be $\mathbf{call}\ f(v)? \cdot \mathbf{ret}\ v'!$.

We will show that the traces we define are fully abstract. That is, we will define them such that $M_1 \simeq M_2 \iff \mathrm{Tr}(M_1) = \mathrm{Tr}(M_2)$, where $\mathrm{Tr}(M)$ is the set of all possible traces generated by M when interacting with some unknown context. In Section 3.6, we will show that whenever $\mathrm{Tr}(M_1{\downarrow}) \neq \mathrm{Tr}(M_2{\downarrow})$, that $\mathrm{Tr}(M_1) \neq \mathrm{Tr}(M_2)$. Because our traces are fully abstract, this implies that $M_1 \simeq M_2 \implies M_1{\downarrow} \simeq M_2{\downarrow}$, which is the core of our full abstraction theorem.

3.5.1 High-level traces

Figure 3.6 defines a labeled transition relation $\Theta \xrightarrow{\overline{a_h}}{}\!\!\!\!\!\rightarrow_h \Theta'$ between configurations Θ and Θ', which can either be a starting state $C|M$, or an intermediate or final state $\Sigma \vdash \langle g, s, S\rangle$. The trace of (the execution of) a high-level program $C|M$ is defined as the sequence of basic actions $\overline{a_h}$ generated by this transition relation, starting from the initial state $C|M$. Each high-level basic action a_h is either:

- a call '$\mathbf{call}\ f(v)?$' from C to M, where f is the name of the function that was called and v is the value passed as argument;
- a return '$\mathbf{ret}\ v!$' from M to C, where v is the return value;
- an outcall '$\mathbf{call}\ f(v)!$' from M to C, where f is the name of the function that was called and v is the value passed as argument;

$$\frac{C|M \longrightarrow \Sigma \vdash \langle g, \overline{s},\ S\rangle}{C|M \overset{\tau}{\Longrightarrow}_h C|M \vdash \langle \mathrm{name}(c), \overline{s},\ S\rangle}\ \textsc{Tr-Init}$$

$$\frac{\mathrm{fc}(C|M) \vdash \langle g, (m,s)::\overline{s},\ S\rangle \longrightarrow \Sigma \vdash \langle g', (m,s')::\overline{s}',\ S'\rangle}{C|M \vdash \langle g, (m,s)::\overline{s},\ S\rangle \overset{\tau}{\Longrightarrow}_h C|M \vdash \langle g', (m,s')::\overline{s}',\ S'\rangle}\ \textsc{Tr-Internal}$$

$$\frac{\begin{array}{c}\mathrm{fc}(C|M) \vdash \langle g, (m,s)::\overline{s},\ x := E(E');\ S\rangle \longrightarrow \Sigma \vdash \langle g', (c,s')::\overline{s}',\ S'\rangle \\ [\![E]\!]_s = m.f \qquad [\![E']\!]_s = v \qquad c = \mathrm{name}(C) \qquad m = \mathrm{name}(M)\end{array}}{C|M \vdash \langle g, (m,s)::\overline{s},\ x := E(E');\ S\rangle \overset{\mathbf{call}\ f(v)?}{\Longrightarrow}_h C|M \vdash \langle g', (c,s')::\overline{s}',\ S'\rangle}\ \textsc{Tr-InCall}$$

$$\frac{\begin{array}{c}\mathrm{fc}(C|M) \vdash \langle g, (c,s)::\overline{s},\ x := E(E');\ S\rangle \longrightarrow \Sigma \vdash \langle g', (m,s')::\overline{s}',\ S'\rangle \\ [\![E]\!]_s = c.f \qquad [\![E']\!]_s = v \qquad c = \mathrm{name}(C) \qquad m = \mathrm{name}(M)\end{array}}{C|M \vdash \langle g, (c,s)::\overline{s},\ x := E(E');\ S\rangle \overset{\mathbf{call}\ f(v)!}{\Longrightarrow}_h C|M \vdash \langle g', (m,s')::\overline{s}',\ S'\rangle}\ \textsc{Tr-OutCall}$$

$$\frac{\begin{array}{c}\mathrm{fc}(C|M) \vdash \langle g, (m,s)::(c,s')::\overline{s},\ x := \mathbf{ret}\ x';S\rangle \longrightarrow \Sigma \vdash \langle g', (c,s'')::\overline{s},\ S'\rangle \\ s(x') = v \qquad c = \mathrm{name}(C) \qquad m = \mathrm{name}(M)\end{array}}{C|M \vdash \langle g, (m,s)::(c,s')::\overline{s},\ x := \mathbf{ret}\ x';S\rangle \overset{\mathbf{ret}\ v!}{\Longrightarrow}_h C|M \vdash \langle g', (c,s'')::\overline{s},\ S'\rangle}\ \textsc{Tr-Return}$$

$$\frac{\begin{array}{c}\mathrm{fc}(C|M) \vdash \langle g, (c,s)::(m,s')::\overline{s},\ x := \mathbf{ret}\ x';S\rangle \longrightarrow \Sigma \vdash \langle g', (m,s'')::\overline{s},\ S'\rangle \\ s(x') = v \qquad c = \mathrm{name}(C) \qquad m = \mathrm{name}(M)\end{array}}{C|M \vdash \langle g, (c,s)::(m,s')::\overline{s},\ x := \mathbf{ret}\ x';S\rangle \overset{\mathbf{ret}\ v!}{\Longrightarrow}_h C|M \vdash \langle g', (m,s'')::\overline{s},\ S'\rangle}\ \textsc{Tr-RetBack}$$

$$\textsc{Tr-Refl}\ \frac{}{\Theta \overset{\epsilon}{\longrightarrow\!\!\!\!\rightarrow}_h \Theta} \qquad \textsc{Tr-Silent}\ \frac{\Theta \overset{\tau}{\Longrightarrow}_h \Theta'}{\Theta \overset{\epsilon}{\longrightarrow\!\!\!\!\rightarrow}_h \Theta'} \qquad \textsc{Tr-Action}\ \frac{\Theta \overset{\alpha}{\Longrightarrow}_h \Theta'}{\Theta \overset{[\alpha]}{\longrightarrow\!\!\!\!\rightarrow}_h \Theta''} \qquad \textsc{Tr-Trans}\ \frac{\Theta \overset{\overline{\alpha}}{\longrightarrow\!\!\!\!\rightarrow}_h \Theta' \qquad \Theta' \overset{\overline{\alpha}'}{\longrightarrow\!\!\!\!\rightarrow}_h \Theta''}{\Theta \overset{\overline{\alpha\alpha'}}{\longrightarrow\!\!\!\!\rightarrow}_h \Theta''}$$

Figure 3.6: Definition of traces for our high-level imperative language.

- a return-back '**ret** v?' from C to M, where v is the return value.

Notice that interactions from the context to the protected module are annotated with a '?', while interactions from the protected module to the context are annotated with a '!'. Based on the described labeled transition relation, we can define the set of traces of a high-level module as follows.

Definition 3.5.1 (High-level trace set)**.** *The set of traces of a high-level module M is defined as:*

$$\mathrm{Tr}^{\mathrm{H}}(M) = \{\overline{\alpha} \mid \exists C, \Theta.\ C|M \overset{\overline{\alpha}}{\longrightarrow\!\!\!\!\rightarrow}_h \Theta\}$$

From this definition, we can define high-level trace equivalence.

Definition 3.5.2 (High-level trace equivalence)**.** *We say two high-level modules* M_1 *and* M_2 *are* trace equivalent *when their trace sets are equal:*

$$M_1 \simeq_{HT} M_2 \text{ iff } \mathrm{Tr}^{\mathrm{H}}(M_1) = \mathrm{Tr}^{\mathrm{H}}(M_2)$$

The following lemma states that the high-level traces as defined in Figure 3.6 are fully abstract.

Lemma 3.5.1 (Full abstraction of high-level traces)**.**

$$M_1 \simeq_{HT} M_2 \iff M_1 \simeq M_2$$

Proof sketch. In the high-level language, all implementation aspects of a module are hidden and the only way for a context to interact with a module is through function calls to the module. The information exchanged between a module and a context is (1) the names of the functions called, (2) function call arguments and (3) the function call return values. Hence if there is a context that is able to distinguish M_1 from M_2, it can only do so based on these three elements. Since each of these elements is included in the traces of a module, the trace sets of these two modules will differ and hence the $\Longrightarrow$ direction of the lemma follows. Conversely, if $\mathrm{Tr}(M_1) \neq \mathrm{Tr}(M_2)$, then at some point M_1 reacts differently to a function call than M_2 and this means there must be some context C that can trigger this difference by interacting with M_1 and M_2. □

3.5.2 Low-level traces

Figure 3.7 defines the labeled transition relation $\Theta \xrightarrow{\overline{a_l}}{\!\!\!\!\rightarrow}_l \Theta'$ for low-level traces, based on the relation presented in [97]. Considering all possible interactions between a module and a context, each low-level basic action a_l is either:

- a call '**call** $p(r)$?' from C to M, where p is the call target address in M and r is the register and flag file;
- a return '**ret** $p(r)$!' from M to C, where p is the return address in C and r is the register and flag file;
- an outcall '**call** $p(r', p')$!' from M to C, where p is the call target address in C, r' is the register and flag file, and p' is the return address in M;
- a return-back '**ret** r' from C to M, where r is the register and flag file;
- a jump '**jmp** $p(r)$!' from M to C, where p is the jump destination and r is the register and flag file;

$$\frac{C|M \longrightarrow \langle p, r, m\rangle}{C|M \stackrel{\tau}{\Longrightarrow}_l \langle p, r, m\rangle}\ \textsc{Tr-Init} \qquad \frac{\langle p, r, m\rangle \longrightarrow \langle p', r', m'\rangle \quad \mathit{internalJmp}_\delta(p, p')}{\langle p, r, m\rangle \stackrel{\tau}{\Longrightarrow}_l \langle p', r', m'\rangle}\ \textsc{Tr-Internal}$$

$$\frac{\langle p, r, m\rangle \longrightarrow \langle p', r', m'\rangle \quad \mathit{exitJmp}_\delta(p, p') \quad m(p) = \texttt{call d}}{\langle p, r, m\rangle \stackrel{\mathbf{call}\ p'(r', p)!}{\Longrightarrow}_l \langle p', r', m'\rangle}\ \textsc{Tr-OutCall}$$

$$\frac{\langle p, r, m\rangle \longrightarrow \langle p', r', m'\rangle \quad \mathit{exitJmp}_\delta(p, p') \quad m(p) = \texttt{ret}}{\langle p, r, m\rangle \stackrel{\mathbf{ret}\ p'(r')!}{\Longrightarrow}_l \langle p', r', m'\rangle}\ \textsc{Tr-Return}$$

$$\frac{\langle p, r, m\rangle \longrightarrow \langle p', r', m'\rangle \quad \mathit{exitJmp}_\delta(p, p') \quad m(p) \notin \{\texttt{call d}, \texttt{ret}\}}{\langle p, r, m\rangle \stackrel{\mathbf{jmp}\ p'(r')!}{\Longrightarrow}_l \langle p', r', m'\rangle}\ \textsc{Tr-OutJmp}$$

$$\frac{\langle p, r, m\rangle \longrightarrow \langle p', r', m'\rangle \quad \mathit{entryJmp}_\delta(p, p') \quad \neg \mathit{retEntryPoint}_\delta(p)}{\langle p, r, m\rangle \stackrel{\mathbf{call}\ p'(r')?}{\Longrightarrow}_l \langle p', r', m'\rangle}\ \textsc{Tr-InCall}$$

$$\frac{\langle p, r, m\rangle \longrightarrow \langle p', r', m'\rangle \quad \mathit{entryJmp}_\delta(p, p') \quad \mathit{retEntryPoint}_\delta(p')}{\langle p, r, m\rangle \stackrel{\mathbf{ret}\ r'?}{\Longrightarrow}_l \langle p', r', m'\rangle}\ \textsc{Tr-RetBack}$$

$$\frac{\langle p, r, m\rangle \longrightarrow \langle p', r', m'\rangle \quad m(p) = \texttt{st [d] s} \quad \mathit{code}_\delta(p) \quad \mathit{unprotected}_\delta(r(d))}{\langle p, r, m\rangle \stackrel{\mathbf{write}\ (r(d), r(s))!}{\Longrightarrow}_l \langle p', r', m'\rangle}\ \textsc{Tr-WriteOut}$$

$$\frac{\langle p, r, m\rangle \longrightarrow \langle p', r', m'\rangle \quad m(p) = \texttt{ld d [s]} \quad \mathit{code}_\delta(p) \quad \mathit{unprotected}_\delta(r(d))}{\langle p, r, m\rangle \stackrel{\mathbf{read}\ m(r(s))?}{\Longrightarrow}_l \langle p', r', m'\rangle}\ \textsc{Tr-ReadIn}$$

$$\frac{}{\Theta \stackrel{\epsilon}{\longrightarrow\!\!\!\!\rightarrow}_l \Theta}\ \textsc{Tr-Refl} \qquad \frac{\Theta \stackrel{\tau}{\Longrightarrow}_l \Theta'}{\Theta \stackrel{\epsilon}{\longrightarrow\!\!\!\!\rightarrow}_l \Theta'}\ \textsc{Tr-Silent} \qquad \frac{\Theta \stackrel{\alpha}{\Longrightarrow}_l \Theta'}{\Theta \stackrel{[\alpha]}{\longrightarrow\!\!\!\!\rightarrow}_l \Theta''}\ \textsc{Tr-Action} \qquad \frac{\Theta \stackrel{\overline{\alpha}}{\longrightarrow\!\!\!\!\rightarrow}_l \Theta' \quad \Theta' \stackrel{\overline{\alpha}'}{\longrightarrow\!\!\!\!\rightarrow}_l \Theta''}{\Theta \stackrel{\overline{\alpha\alpha'}}{\longrightarrow\!\!\!\!\rightarrow}_l \Theta''}\ \textsc{Tr-Trans}$$

Figure 3.7: Definition of traces for our low-level assembly language.

- a write-out '**write** (a, v)!', when M writes value v at address a which is in C;
- a read-in '**read** v?', when M reads a value v at some memory location of C.

The set of traces of a low-level module is defined in terms of these basic actions, as follows.

Definition 3.5.3 (Low-level trace set)**.** *The set of traces of a low-level module M is defined as:*

$$\mathrm{Tr}^{\mathrm{L}}(M) = \{\overline{\alpha} \mid \exists C, \Theta.\ C|M \xrightarrow{\overline{\alpha}}\!\!\!\!\!\rightarrow_l \Theta\}$$

As we did for high-level traces, we now define low-level trace equivalence.

Definition 3.5.4 (Low-level trace equivalence)**.** *We say two low-level modules M_1 and M_2 are* trace equivalent *when their trace sets are equal:*

$$M_1 \simeq_{LT} M_2 \text{ iff } \mathrm{Tr}^{\mathrm{L}}(M_1) = \mathrm{Tr}^{\mathrm{L}}(M_2)$$

Following the same structure as the section above, we should now argue that these low-level traces are fully abstract. Unfortunately, it turns out that these traces are *not* fully abstract. More specifically, the traces are complete in the sense that they capture all observable actions, but they are unsound because they also capture unobservable actions. Thus, if two low-level modules have the same sets of traces, then they are contextually equivalent, but the opposite statement does not hold.

Lemma 3.5.2 (Completeness of low-level traces)**.** *For any two low-level modules M_1 and M_2, we have:*

$$M_1 \simeq_{LT} M_2 \implies M_1 \simeq M_2$$

Proof sketch. In the low-level language, there are inherently three ways for a context C and a module M to communicate: (1) through unprotected memory, (2) through the register and flag file, and (3) by jumping to or calling memory locations. We will argue that the low-level traces capture each of these interactions. The context can initiate communication in the following ways.

- By calling or jumping to an entry point of M, potentially passing information through the register and flag file. All of this information, including the address that was called, is captured in the Tr-InCall rule.

- By returning to the return entry point from an outcall, again potentially passing information through the register and flag file. This information is captured in the Tr-RetBack rule.

The program counter-based memory protection rules of the low-level language prevent the context from jumping to anywhere else in the protected code section, and from accessing the protected data section. Of course, it is possible that M reads information from unprotected memory in response to a call or return-back, but this will be captured in the Tr-ReadIn rule.

Similarly, the module M can initiate communication as follows.

- By making an outcall to some location in C. The information transferred by this action is (1) the call target location p', (2) the values of the register and flag file r', and (3) the original value of the program counter p, because $inc(p)$ is pushed onto the top of the stack in unprotected memory (see rule E-Call in Figure 3.4). Each of these pieces of information is captured in the Tr-OutCall rule.
- By jumping to some location in C. This case is similar to the previous one, except for the fact that the `jmp` instruction does not place the return address on the stack. Hence the information transferred consists of the call target location p' and the values of the register and flag file r', which are both captured in the Tr-OutJmp rule. Note that it is not possible for the execution to "run off the end" of the protected code section into unprotected memory, because the protected code section is followed by the protected data section, which is non-executable and hence would cause the processor to fault.
- By returning from a call made by C. The information transferred in this case is the return location p' and the register and flag file r'. The original program counter is not disclosed in this case, because it is not written into memory. All information transferred is captured in the Tr-Return rule.
- By writing some value into unprotected memory. The information transferred in this case is the address of the memory write and the value that was written. Both pieces of information are capture in the Tr-WriteOut rule.
- By reading some value from unprotected memory. The information transferred in this case is the value that was read. This information is captured in the Tr-ReadIn rule.

The above items describe all possible ways for a module and a context to communicate, and all of them are captured in the low-level traces. Hence, if the trace sets of two modules M_1 and M_2 are equal, they communicate with the context in exactly the same ways, and since a context C can only distinguish M_1 from M_2 by communicating with them, no context will be able to tell them apart. □

Lemma 3.5.3 (Unsoundness of low-level traces)**.** *For low-level modules M_1 and M_2, we have in general:*

$$M_1 \simeq M_2 \implies M_1 \simeq_{LT} M_2$$

Proof. It is easy to prove this lemma by using a counterexample. Consider a low-level module M_1 that reads the value v at memory location 1 of the context and subsequently returns 0, and a low-level module M_2 that reads the value v' at memory location 2 of the context and also subsequently returns 0. The observable behavior of these two modules is the same, hence we have $M_1 \simeq M_2$. However, assuming $v \neq v'$, the trace sets of these two modules are not equal because the read values v and v' show up in the low-level traces. Hence we have $M_1 \not\simeq_{LT} M_2$. □

As argued above, the problem is that the low-level traces capture too much: they not only capture all observable behavior of a module, they also capture unobservable actions. Read-ins and write-outs are the offending actions, since they are not always observable by the context. Patrignani and Clarke [98] have studied this problem in depth and propose two possible solutions, based on earlier observations by Curien [34]. The first solution consists of modifying the trace semantics, such that the traces capture more precisely what is communicated between the module and the context. In particular, a read-in or write-out label should only be generated for *observable* read-ins or write-outs, which is not trivial to achieve for our low-level language. The other solution is more straightforward and consists of preventing or avoiding read-ins and write-outs from occurring altogether. In the next section, we will show that this solution is a natural fit for our compilation scheme.

3.5.3 Revised low-level traces

As argued above, the low-level traces are not fully abstract because they not only capture all observable actions, but certain unobservable read-ins and write-outs as well. In this section, we will show that, while in general a low-level module can perform any of the low-level actions a_l, the low-level modules that

result from the compilation of a high-level module using our secure compiler are more reserved in the kinds of interactions they engage in. This allows us to create a revised low-level trace semantics that is both simpler and fully abstract for securely compiled modules (it will not be fully abstract in general, for arbitrary modules, however). We make the following simplifications for our revised low-level traces.

- A securely compiled module never performs any reads from unprotected memory. Hence, **read** actions will never show up in low-level traces of a securely compiled module and thus we can discard them for our revised low-level traces.
- A securely compiled module never performs an outcall directly, but always passes through a function (named `outcall_helper` in the formalization in Appendix A) that (1) switches from the secure stack to the public stack, (2) places the address of the return entry point on the stack, (3) clears all flags and registers except the argument passing register `R4` and the `SP` register, and finally (4) jumps to the call target using a `jmp` instruction. The `call` instruction is never used. Hence, the **call** $p(r', p')!$ action will never show up in any low-level trace of a compiled module, but instead we will see a **write** $(s, v)!$ followed by a **jmp** $p(v')$, where s is the top of the public stack, v is the address of the return entry point, p is the call target destination and v' is the value of the argument passed in `R4`. For our revised low-level traces, we can hence discard the original **call** $p(r', p')!$ action and we can replace sequences **write** $(s, v)! \cdot$ **jmp** $p(v')$ by a new action **call** $p(v')$. This new call action contains the same information as the sequence it replaces, even though it does not mention s or v. This is because s is always the top of the public stack (which is controlled completely by the context) and v is the address of the return entry point, which is constant for all protected modules and hence does not carry any information.
- Apart from the case described above, a securely compiled module never performs any write or jump to unprotected memory. Hence **write** and **jmp** actions never show up in low-level traces, except in the above case, where we have replaced them by the new **call** $p(v')$ action. Thus, we can discard **jmp** and **write** actions for our revised low-level traces.
- When a function of a securely compiled module returns, it will first return to the entry point through which it was called, which (1) switches from the secure stack back to the public stack, (2) checks that the return address (which is on the top of the stack) is a valid address in unprotected memory, (3) clears all flags and registers except `R0` and `SP`, and finally (4) returns

back to unprotected code. Hence, for our revised low-level traces, we can replace each **ret** $p(r)!$ action by a **ret** $v!$ action, where v is the value of `R0`. We can discard the return location p, because it is controlled completely by the context and hence does not represent any information transferred from the module to the context. We also do not need to incorporate the value of `SP` into our traces, because it will point to the top of the public stack, which is also controlled completely by the context.

- When a securely compiled module receives a call from the context, the module never reads any of the values passed through the register and flag file, *except* the value of `R4`, which contains the call argument, and the value of `SP`, which is simply stored somewhere in protected memory such that the stack can be switched to the secure stack and later restored to the public stack. Thus, in our revised low-level traces, we can simplify **call** $p(r')?$ actions to the format **call** $p(v)?$, where p is the call target address and v is value of `R4`.
- When a context returns back to a securely compiled module after an outcall, the protected module never reads any of the values passed through the register and flag file, *except* (1) the value of `R0`, which contains the return value, and (2) the value of `SP`, which is only stored temporarily, as described above. Thus, in our revised low-level traces, we can simplify **ret** $r?$ actions to the format **ret** $v?$, where v is value of `R0`.

Figure 3.8 formally defines these revised low-level traces and we give the definitions for trace sets and equivalence below.

Definition 3.5.5 (Revised low-level trace set)**.** *The revised set of traces of a securely compiled low-level module $M{\downarrow}$ is defined as:*

$$\mathrm{Tr}^{\mathrm{R}}(M{\downarrow}) = \{\overline{\alpha} \mid \exists C, \Theta.\ C|M{\downarrow} \overset{\overline{\alpha}}{\Longrightarrow}_r \Theta\}$$

Definition 3.5.6 (Revised low-level trace equivalence)**.** *We say two securely compiled low-level modules $M_1{\downarrow}$ and $M_2{\downarrow}$ are trace equivalent when their revised trace sets are equal:*

$$M_1{\downarrow} \simeq_{RT} M_2{\downarrow} \text{ iff } \mathrm{Tr}^{\mathrm{R}}(M_1{\downarrow}) = \mathrm{Tr}^{\mathrm{R}}(M_2{\downarrow})$$

Lemma 3.5.4 (Full abstraction of revised low-level traces)**.** *For two securely compiled low-level modules $M_1{\downarrow}$ and $M_2{\downarrow}$, we have:*

$$M_1{\downarrow} \simeq_{RT} M_2{\downarrow} \iff M_1{\downarrow} \simeq M_2{\downarrow}$$

Proof sketch. The above paragraphs have argued why no information is lost when using the revised low-level traces for securely compiled modules,

$$\frac{C|M \longrightarrow \langle p,r,m\rangle}{C|M \overset{\tau}{\Longrightarrow}_r \langle p,r,m\rangle}\ \textsc{Tr-Init} \qquad \frac{\langle p,r,m\rangle \longrightarrow \langle p',r',m'\rangle \quad \mathit{internalJmp}_\delta(p,p')}{\langle p,r,m\rangle \overset{\tau}{\Longrightarrow}_r \langle p',r',m'\rangle}\ \textsc{Tr-Internal}$$

$$\frac{\langle p,r,m\rangle \longrightarrow \langle p',r',m'\rangle \quad \mathit{exitJmp}_\delta(p,p') \quad m(p) = \texttt{jmp d} \quad v = r(\texttt{R4})}{\langle p,r,m\rangle \overset{\mathbf{call}\ p'(v)!}{\Longrightarrow}_r \langle p',r',m'\rangle}\ \textsc{Tr-OutCall}$$

$$\frac{\langle p,r,m\rangle \longrightarrow \langle p',r',m'\rangle \quad \mathit{exitJmp}_\delta(p,p') \quad m(p) = \texttt{ret} \quad v = r(\texttt{R0})}{\langle p,r,m\rangle \overset{\mathbf{ret}\ v!}{\Longrightarrow}_r \langle p',r',m'\rangle}\ \textsc{Tr-Return}$$

$$\frac{\langle p,r,m\rangle \longrightarrow \langle p',r',m'\rangle \quad \mathit{entryJmp}_\delta(p,p') \quad \neg \mathit{retEntryPoint}_\delta(p) \quad v = r(\texttt{R4})}{\langle p,r,m\rangle \overset{\mathbf{call}\ p'(v)?}{\Longrightarrow}_r \langle p',r',m'\rangle}\ \textsc{Tr-InCall}$$

$$\frac{\langle p,r,m\rangle \longrightarrow \langle p',r',m'\rangle \quad \mathit{entryJmp}_\delta(p,p') \quad \mathit{retEntryPoint}_\delta(p') \quad v = r(\texttt{R0})}{\langle p,r,m\rangle \overset{\mathbf{ret}\ v?}{\Longrightarrow}_r \langle p',r',m'\rangle}\ \textsc{Tr-RetBack}$$

$$\textsc{Tr-Refl}\ \frac{}{\Theta \overset{\epsilon}{\longrightarrow\!\!\!\!\rightarrow}_r \Theta} \qquad \textsc{Tr-Silent}\ \frac{\Theta \overset{\tau}{\Longrightarrow}_r \Theta'}{\Theta \overset{\epsilon}{\longrightarrow\!\!\!\!\rightarrow}_r \Theta'} \qquad \textsc{Tr-Action}\ \frac{\Theta \overset{\alpha}{\Longrightarrow}_r \Theta'}{\Theta \overset{[\alpha]}{\longrightarrow\!\!\!\!\rightarrow}_r \Theta''} \qquad \textsc{Tr-Trans}\ \frac{\Theta \overset{\overline{\alpha}}{\longrightarrow\!\!\!\!\rightarrow}_r \Theta' \quad \Theta' \overset{\overline{\alpha}'}{\longrightarrow\!\!\!\!\rightarrow}_r \Theta''}{\Theta \overset{\overline{\alpha\alpha'}}{\longrightarrow\!\!\!\!\rightarrow}_r \Theta''}$$

Figure 3.8: Definition of revised traces for our low-level assembly language. These traces are only fully abstract for *securely compiled* low-level modules.

compared to the original low-level traces. Hence we have $M_1{\downarrow} \simeq_{RT} M_2{\downarrow} \implies M_1{\downarrow} \simeq_{LT} M_2{\downarrow}$ and based on Lemma 3.5.2, we thus have $M_1{\downarrow} \simeq_{RT} M_2{\downarrow} \implies M_1{\downarrow} \simeq M_2{\downarrow}$. The soundness of the revised low-level traces follows from the fact that any action that is part of a revised low-level trace is now observable in the corresponding low-level execution. This is because control is transferred from one component to the other (the context or the module) as part of each action, and any call destination, argument value or return value included in the action is directly observable by the destination component in the low-level execution. Hence we also have $M_1{\downarrow} \simeq M_2{\downarrow} \implies M_1{\downarrow} \simeq_{RT} M_2{\downarrow}$ and thus the lemma holds. □

3.6 Full abstraction

We can now start to prove our full abstraction theorem, but before we begin, there is one important assumption that we must point out, which is that we assume that the secure stack of the low-level modules under consideration never overflows. The limited size of the secure stack could be seen as a failure of full abstraction, because it reveals information about the number of local variables used by a module, which is not visible at the high level. However, we will abstract from this problem, because revealing this information does not pose a security risk in practice.

Full abstraction breaks down into two parts: soundness and completeness.

Theorem 3.6.1 (Soundness)**.** *For any two high-level modules M_1 and M_2, we have:*

$$M_1{\downarrow} \simeq M_2{\downarrow} \Rightarrow M_1 \simeq M_2$$

The soundness theorem states that if two low-level modules behave identically, then their high-level sources must behave identically as well. Or, in other words, if two high-level modules behave differently, then their compilations behave differently as well. This essentially means the compiler is correct (but not necessarily secure), since otherwise two modules that have different high-level semantics are translated to low-level modules that behave the same, and hence at least one of those translations behaves incorrectly. Proving this theorem is nontrivial, but since the focus of this chapter is on proving the *security* of the compiler, we do not prove the soundness theorem.

Theorem 3.6.2 (Completeness)**.** *For any two high-level modules M_1 and M_2, we have:*

$$M_1 \simeq M_2 \Rightarrow M_1{\downarrow} \simeq M_2{\downarrow}$$

To prove the completeness theorem, we will prove the equivalent statement $M_1{\downarrow} \not\simeq M_2{\downarrow} \implies M_1 \not\simeq M_2$. The general idea is as follows. From Lemma 3.5.4, we know that $M_1{\downarrow} \not\simeq M_2{\downarrow} \implies M_1{\downarrow} \not\simeq_{RT} M_2{\downarrow}$ and hence, by the definition of revised low-level traces, there is some low-level context C_l such that $C_l|M_1{\downarrow} \longrightarrow^* r$ and $C_l|M_2{\downarrow} \not\longrightarrow^* r$. Let $\overline{a_1}$ and $\overline{a_2}$ be the traces of $C_l|M_1{\downarrow}$ and $C_l|M_2{\downarrow}$ respectively, then we know $\overline{a_1} \neq \overline{a_2}$. We will describe an algorithm that, given any such high-level M_1, M_2, and revised low-level traces $\overline{a_1}$ and $\overline{a_2}$ as input, will construct a high-level C_h such that $C_h|M_1 \longrightarrow^* r$ and $C_h|M_2 \not\longrightarrow^* r$. The existence of this algorithm proves the completeness theorem.

The algorithm relies on the following two propositions.

Proposition 3.6.1. *If we number the actions of a high-level or revised low-level trace starting at 0, then $a^{(i)}$ is a call or a return-back (i.e., a ?-type action)*

when i is even, and $a^{(i)}$ is an outcall or a return (i.e., a !*-type action) when i is odd.*

Proof. The proof goes by induction on i. For $i = 0$, the proposition certainly holds, because execution starts in the `main` function of the context and hence action number 0 can only be a call. Assuming that the proposition holds for $i = n$, we now prove it holds for $i = n + 1$. If n is even, then $a^{(n)}$ is a call or return-back. In either case, control will be in the protected module M after the call or return-back and hence, the next action can only be a return or outcall respectively. The case for when n is odd is symmetrical. □

Proposition 3.6.2. *For any low-level C, $M_1{\downarrow}$, and $M_2{\downarrow}$, the revised low-level traces $\overline{a_1}$ and $\overline{a_2}$ generated by $C|M_1{\downarrow}$ and $C|M_2{\downarrow}$ respectively, differ for the first time at an odd-numbered (i.e.,* !*-type) action.*

Proof. Suppose the first pair of differing actions is even-numbered. By Proposition 3.6.1, all even-numbered actions are calls or return-backs, which originate from C. The information carried by these actions is the target address p and the argument value v for a call, or the return value v for a return-back. At least one of these components must be different for the actions to be different, but this means some state of C must already be different in the two executions, which can only be because of a prior difference in the actions performed (because both executions start from the same context C). This is a contradiction and hence our assumption that the first pair of differing actions is even-numbered must be false. □

3.6.1 Lifting low-level values

The algorithm that we will use to prove our completeness theorem requires a function for 'lifting' low-level values to a corresponding high-level value. The input to this function consists of (1) the expected high-level type T of the value to lift, (2) the low-level value v to lift, and (3) a *function table* τ mapping (*address*, *function type*) pairs to high-level function names. Each entry of this function table will correspond to a function of the high-level module that is under construction by the algorithm. The lifting function is defined as follows.

$$\mathsf{lift}(\mathit{Unit}, v, \tau) = \textbf{if } v = 0 \textbf{ then } \mathit{unit} \textbf{ else Fail}$$

$$\mathsf{lift}(\mathit{Int}, v, \tau) = v$$

$$\begin{array}{l}\mathsf{lift}(\langle T \to T' \rangle, v, \tau) = \\ \quad \textbf{match } \tau(v, \langle T \to T' \rangle) \textbf{ with} \\ \quad\quad f \textbf{ then } c.f \\ \quad\quad \epsilon \textbf{ then } T'\ f'(T\ x)\ \{\textbf{vars } T'\ x' = \mathsf{default}(T);\ \textbf{ret } x'\} \\ \quad \textbf{where} \\ \quad\quad \mathsf{default}(\mathit{Unit}) = \mathit{unit} \\ \quad\quad \mathsf{default}(\mathit{Int}) = 0 \\ \quad\quad \mathsf{default}(\langle T \to T' \rangle) = \mathit{null}\end{array}$$

We assume the function name f' in the case where $T = \langle T' \to T'' \rangle$ is a fresh function name not yet occurring in the high-level module under construction.

3.6.2 Algorithm

We are now ready to define the algorithm for proving our completeness theorem. It takes as input two high-level modules M_1 and M_2 and non-equal revised low-level traces $\overline{a_1}$ and $\overline{a_2}$, generated by the executions of $C_l|M_1{\downarrow}$ and $C_l|M_2{\downarrow}$ for some low-level context C_l. From this, the algorithm will construct a high-level context C such that the (high-level) trace of $C|M_1$ differs from the trace of $C|M_2$, thereby proving that $M_1 \not\approx_{HT} M_2$. The idea behind the algorithm is that the created context C will try to "mimic" the revised low-level execution traces $\overline{a_1}$ and $\overline{a_2}$ at the high level. The created context must perform ?-type actions such that, at every step of the execution of $C|M_1$ and $C|M_2$, the high-level state of M_1 and M_2 is equivalent to the low-level state of $M_1{\downarrow}$ and $M_2{\downarrow}$ in the execution of $C_l|M_1{\downarrow}$ and $C_l|M_2{\downarrow}$ respectively. This will ensure that, at the execution point where $M_1{\downarrow}$ performs a different action than $M_2{\downarrow}$, their high-level counterparts M_1 and M_2 also perform different actions. While describing the algorithm, we will at each point argue why the algorithm ensures that the state of M_1 and M_2 remains equivalent to the state of $M_1{\downarrow}$ and $M_2{\downarrow}$ respectively and why the !-type actions generated by M_1 and M_2 have the same format as those generated by $M_1{\downarrow}$ and $M_2{\downarrow}$ respectively.

As internal variables, the algorithm uses an integer *step counter* variable i, a stack of *return locations* $\overline{r}$ and a *current function name* f_c. The algorithm will iterate over the pairs of basic actions of the $\overline{a_1}$ and $\overline{a_2}$ given as input, and the step counter will indicate the pair of actions that is currently being handled.

The top of the stack of return locations will indicate the precise location in the context under construction where M_1 and M_2 will return to on the next return action. Finally, the current function name simply indicates the name of the function of C that is currently being constructed by the algorithm.

The algorithm will alternate between a *construction* mode and an *execution* mode, and will increment i by 1 at each alternation. The algorithm will start in construction mode and hence it will handle ?-type actions in construction mode and odd-numbered !-type actions in execution mode. In construction mode, the algorithm will add statements to the context under construction C, and in execution mode, the algorithm will be executing functions of M_1 and M_2. Hence we assume the algorithm contains an interpreter for the high-level language, following the evaluation rules of Figure 3.3. The algorithm always maintains the following invariant: $\forall j < i : a_1^{(j)} = a_2^{(j)}$. Thus, it finishes when it reaches the pair of basic actions that differ. According to Proposition 3.6.2, this will be at an odd-numbered, !-type action.

Initialization

C is initialized to:

```
module c {
  Int step = 0;

  Int main() {
    vars;
    ret 0;
  }
}
```

The initial return location stack $\overline{r}$ is empty, the current function f_c is set to `main` and the step counter i is set to 0. The algorithm must also initialize the function table τ. It does this by scanning M_1, M_2, to create a list of all function reference literals and their corresponding type. For each distinct function reference $c.f$ found (where c is the name of the context C) with type $\langle T \to T' \rangle$, a new function $T'\ f'(T\ x)\ \{\mathbf{vars}\ T'\ x' = \mathsf{default}(T);\ \mathbf{ret}\ x'\}$ is created in C and an entry $(p, \langle T \to T' \rangle) \mapsto c.f$ is added to τ, where p is the low-level memory location of the start of the newly added function after compilation of C. This initialization step is required to ensure that when the algorithm sees the low-level value p being exchanged between C and M_1 or M_2 as part of a low-level interaction, that it can consistently exchange the correct corresponding function $c.f$ at the high level.

Construction mode

Whenever the algorithm is in this mode, i is even, so by Proposition 3.6.2 we have $a_1^{(i)} = a_2^{(i)}$. We shall refer to these actions simply as $a^{(i)}$. The algorithm will add a block of code at the end of the current function f_c, right before its return statement. The code to add depends on the kind of $a^{(i)}$ we are dealing with.

- If $a^{(i)} = \textbf{call}\ p(v)?$, the algorithm will first determine which high-level function corresponds to the called address p. This address *must* correspond to the address of one of the entry points of $M_1{\downarrow}$ and $M_2{\downarrow}$, for otherwise a **call** ? action would not have been generated (see rule Tr-InCall of Figure 3.8). For the same reason, the called entry point cannot be the return entry point. Since the mapping of entry points to functions is deterministic, the algorithm can determine which high-level function f corresponds to the p that was called. When the algorithm knows f, it also knows the type T of the function's argument and its return type T'. It now has enough information to call the lifting function for the argument value: $v{\uparrow} = \mathsf{lift}(T, v, \tau)$. There are four possible outcomes for this lifted value:

  The Fail outcome - This can only happen when $T = \mathit{Unit}$ and $v \neq 0$. This means the low-level context that generated this action has performed an illegal operation: it has specified an invalid value for a function parameter, as described in Section 2.2.4. However, our secure compiler was designed to handle this situation by halting execution with outcome 0. This contradicts the given fact that $\overline{a_1} \neq \overline{a_2}$, and hence we can conclude that this case cannot occur.

  A new function $T''\ f'(T'\ x)\ \{\dots\}$ - This means that $T = \langle T' \to T'' \rangle$ for some T' and T'', and that (v, T) was not yet present in the function table τ. It also means that v is an address somewhere outside of the protected memory region, for otherwise the protected module would halt the execution with outcome 0 right after this call, again contradicting the fact that $\overline{a_1} \neq \overline{a_2}$. When the algorithm encounters this case, it will add the new function definition $v{\uparrow}$ to the module under construction C and it will update the function table by mapping (v, T) to the name of the new function $c.f'$. This ensures that the next time the algorithm tries to lift (v, T), it will end up using the *same* high-level function. This is important because protected modules are allowed to compare function pointers of the same type, and hence we must ensure that whenever we see two equal values in the actions of the revised low-level traces $\overline{a_1}$ and $\overline{a_2}$

that we are mimicking, that the corresponding high-level values are also equal.

An existing function name $c.f$ - This means that $T = \langle T' \to T'' \rangle$ for some T' and T'', and (v, T) was already present in the function table τ.

A integer value v - This means that either $T = \mathit{Unit}$ and $v = 0$, or that $T = \mathit{Int}$.

In any case, the algorithm adds the following block of code to f_c.

```
if (c.step = i) {
  c.step = c.step + 1;
  T' r = m.f(v↑);
                    ⇐
}
```

In this piece of code, i refers to the current step counter value, m is $\text{name}(M_1) = \text{name}(M_2)$, and T, T', f and $v\uparrow$ are as described above. The arrow indicates the return location l_r. We use a bit of syntactic sugar here for the format of the if-statement and for declaring a new variable r inline, instead of in the initial **vars** declaration at the start of f_c. We also assume that r is a fresh variable name and that the `step` field has an unlimited integer range (we can simulate an arbitrarily large range using multiple 32-bit variables).

After adding this block of code to f_c, the algorithm increments its internal step counter i, pushes l_r onto the return location stack $\overline{r}$ and then switches to *execution* mode, passing the location of the called function name $m.f$ in M_1 and in M_2 as arguments to this mode.

- If $a^{(i)} = \textbf{ret}\ v?$, the algorithm will add the following block of code to f_c, right before its return statement.

```
if (c.step = i) {
  c.step = c.step + 1;
  T x = v↑;
  ret x;
}
```

Here, i again refers to the current step counter value, and $v\uparrow = \mathsf{lift}(T, v, \tau)$. The type T to use for lifting is the return type of the current function f_c. For the same reason as described above, the lifting function cannot fail (i.e., if $T = \mathit{Unit}$, v must be 0).

After adding this block of code to f_c, the algorithm increments its internal step counter i, pops two locations l_1 and l_2 from the return stack $\overline{r}$ and switches to *execution* mode, passing l_1 and l_2 as arguments to this mode.

Execution mode

In this mode, the algorithm will execute high-level statements of M_1 and M_2 using an embedded interpreter based on Figure 3.3. The step counter i will always be odd when the algorithm is in this mode and hence the current action will be a !-type action. This mode always executes after construction mode, and receives as arguments from that mode two locations l_1 and l_2 in M_1 and M_2 respectively, indicating where the execution should start or continue. However, before starting execution, the algorithm first checks if the current low-level actions $a_1^{(i)}$ and $a_2^{(i)}$ both exist. At least one of them must exist, for otherwise we have a contradiction with the fact that $\overline{a_1} \neq \overline{a_2}$.

If $a_1^{(i)}$ exists, the algorithm runs its interpreter from location l_1 until it encounters a high-level outcall **call** $c.f(v)!$ or return **ret** $v!$ to C. If $a_2^{(i)}$ exists, the algorithm does the same for location l_2. We shall call these high-level actions $h_1^{(i)}$ and $h_2^{(i)}$ respectively. After reaching the outcall or return, the interpreter pauses, thereby saving the execution state, so that it can continue execution using the same variable and field values the next time the algorithm enters the execution mode.

If only $a_1^{(i)}$ exists, this means the second execution trace has ended. In this case, the algorithm decides its next action based on the form of $h_1^{(i)}$.

- If $h_1^{(i)} =$ **ret** $v!$, the algorithm first pops a location r_l from the return location stack $\overline{r}$. This is the code location in C that M_1 will return to. It then simply adds the statement `exit(1)` at that location and ends.
- If $h_1^{(i)} =$ **call** $c.f(v)!$, the algorithm adds the following code block at the end of function $c.f$ of C, right before its return statement.

  ```
  if (c.step = i) {
    exit(1);
  }
  ```

  The algorithm ends after adding this code.

If only $a_2^{(i)}$ exists, the algorithm behaves symmetrically to the case where only $a_1^{(i)}$ exists. If both $a_1^{(i)}$ and $a_2^{(i)}$ exist, the algorithm decides its next action based on the forms of $h_1^{(i)}$ and $h_2^{(i)}$.

- If $h_1^{(i)} =$ **ret** $v_1!$ and $h_2^{(i)} =$ **ret** $v_2!$, the algorithm first pops the return location l_r from the top of the return location stack $\overline{r}$. The current

function f_c is set to the function enclosing this location. The algorithm then checks whether the values of v_1 and v_2 are equal. If so, the algorithm adds the statement '`step += 1`' at location l_r, increments its internal step counter i and returns to construction mode. However, if the values are different, the algorithm adds the following code at location l_r:

```
if (r = v1) {
  exit(1);
} else {
  exit(2);
}
```

The algorithm then ends after adding this block of code.

- If $h_1^{(i)} = \textbf{call}\ c.f_1(v_1)!$ and $h_2^{(i)} = \textbf{call}\ c.f_2(v_2)!$, the algorithm decides its next action based on whether $f_1 = f_2$ and $v_1 = v_2$.

  If $f_1 = f_2 \equiv f$ and $v_1 = v_2$, the algorithm adds the following code right before the final return statement in $c.f$:

  ```
  if (c.step = i) {
    c.step = c.step + 1;
  }
  ```

  The algorithm then pushes the locations in M_1 and M_2 right after the outcalls onto the return location stack, sets f as the new current function f_c and increments its internal step counter i, before returning to construction mode.

  If $f_1 = f_2 \equiv f$, but $v_1 \neq v_2$, the algorithm adds the following code right before the final return statement in $c.f$, where `x` refers to f's parameter name.

  ```
  if (c.step = i) {
    if (x = v1) {
      exit(1);
    } else {
      exit(2);
    }
  }
  ```

  The algorithm then ends at this point.

  If $f_1 \neq f_2$, the algorithm adds the following code to both $c.f_1$ and $c.f_2$, right before their return statements, where *num* is 1 in f_1 and 2 in f_2.

  ```
  if (c.step = i) {
    exit(num);
  }
  ```

The algorithm ends at this point.

- If h_1 and h_2 are two different kinds of actions (i.e., one is an outcall **call** $c.f(v_1)$! and the other is a return **ret** v!), the algorithm first pops the return location l_r from the return stack and adds the statement `exit(1)` at that location in C. The algorithm then adds the following code right before the final return statement in $c.f$.

```
if (c.step = i) {
  exit(2);
}
```

The algorithm then ends at this point.

3.6.3 Correctness proof

We will now argue why the algorithm described above generates a high-level context C that can distinguish the given M_1 from M_2. The argumentation is based on a correspondence between high- and low-level states.

Definition 3.6.1 (High- and low-level state equivalence)**.** *We say that the state $\langle g, \overline{s}, S\rangle$ of a high-level module M is equivalent to the state $\langle p, r, m\rangle$ of its low-level translation $M{\downarrow}$ in the context of the high- and low-level executions of $C_h|M$ and $C_l|M{\downarrow}$ respectively (for some arbitrary C_h and C_l), when the following properties hold.*

- *The program counter p either points somewhere in unprotected memory, or it points to the first instruction of the translation (by the secure compiler) of S in $M{\downarrow}$.*
- *For each field x of M in g with high-level value v_h, there must be a corresponding memory location l in m with low-level value v_l.*
- *For each local variable y with high-level value v_h in each activation record $(n, s) \in \overline{s}$ for which $n = \text{name}(M)$, there must be a corresponding memory location l in m with low-level value v_l.*
- *For each pair (v_h, v_l) of high- and low-level values described above, the following properties must hold.*
  - *If v_h is Unit-typed, then $v_l = 0$.*
  - *If v_h is Int-typed, then $v_h = v_l$.*
  - *If v_h has function reference type $\langle T \to T'\rangle$, then for every other pair (v'_h, v'_l) with the* same type*, we have $v_h = v'_h \iff v_l = v'_l$.*

By definition of the compiler, the initial high-level state of a module M will always be equivalent to the initial low-level state of its translation $M{\downarrow}$, as expressed by the following lemma.

Lemma 3.6.1 (Initial state equivalence)**.** *For any C_l, C_h and M, the initial state of M in the execution of $C_h|M$ is equivalent to the initial state of $M{\downarrow}$ in the execution of $C_l|M{\downarrow}$.*

Proof. This follows directly from the definition of the compiler. There are no activation records of M yet, hence we only need to take field values into account. *Unit*-typed values are translated to 0, integers are trivially translated to their own value, and each occurrence of a particular function reference $c.f$ is translated to the same memory location p in the linking phase. □

The following lemma says that executing statements of M together with the corresponding instructions in $M{\downarrow}$ does not break state equivalence.

Lemma 3.6.2 (State equivalence preservation)**.** *If, for some C_h and C_l, at some point in the execution of $C_h|M$, the next statement to be executed is part of M, and the current execution state is equivalent to the execution state of $M{\downarrow}$ at some point in the execution of $C_l|M{\downarrow}$, then the state of M will still be equivalent to the state of $M{\downarrow}$ when that statement and the corresponding low-level instructions have been executed.*

Proof sketch. Since the high- and low-level states are initially equivalent and the behavior of the next low-level instructions to be executed corresponds exactly to the behavior of the next high-level statement to be executed, the states will still be equivalent after the execution of the high-level statement and the corresponding low-level instructions. More specifically, if we consider all state-changing operations that are possible in the high-level language, we can see that the compiler ensures that the low-level state changes correspondingly. One potential state-changing operation that deserves special attention is an equality test between two function references. The high-level language allows function references of the same type to be compared to each other in the condition of an if-statement. Hence, to ensure that high- and low-level executions remain in sync, it is important that our definition of state equivalence requires any two equal high-level function reference variables of the same type, to be equal at the low-level as well, and vice versa. □

We can now prove the correctness of our algorithm, using the lemmas given above.

Theorem 3.6.3 (Algorithm correctness)**.** *Given two high-level modules M_1 and M_2, and the non-equal revised low-level traces $\overline{a_1}$ and $\overline{a_2}$ corresponding to the execution of $C_l|M_1{\downarrow}$ and $C_l|M_2{\downarrow}$ respectively, where C_l is a low-level module that distinguishes $M_1{\downarrow}$ from $M_2{\downarrow}$, the algorithm described in Section 3.6.2 generates a high-level C that such that the execution of $C|M_1$ and $C|M_2$ generate different traces.*

Proof. By Lemma 3.6.1, there is an initial state equivalence between M_1 and $M_1{\downarrow}$, and between M_2 and $M_2{\downarrow}$. We will argue by induction on the interactions between C and the modules, that for $x \in \{1, 2\}$, we have that (1) this equivalence is maintained throughout the synced execution of $C|M_x$ and $C_l|M_x{\downarrow}$, and that (2) the high-level actions performed by M_x are similar to the revised low-level actions performed by $M_x{\downarrow}$, in the sense that M_1 and M_2 will perform different actions in the same step that $M_1{\downarrow}$ and $M_2{\downarrow}$ do. By synced execution, we mean that when we execute a statement of $C|M_x$, we execute the corresponding instructions in $C_l|M_x{\downarrow}$.

Assume that, after interaction i, the state of M_1 is equivalent to the state of $M_1{\downarrow}$ and the state of M_2 is equivalent to the state of $M_2{\downarrow}$, where i is odd. We will show that the states are still equivalent after interaction $i + 2$ and that $l_1^{(i+2)} \neq l_2^{(i+2)} \implies h_1^{(i+2)} \neq h_2^{(i+2)}$, where $\overline{l_1}$ and $\overline{l_2}$ are the revised low-level execution traces of $C_l|M_1{\downarrow}$ and $C_l|M_2{\downarrow}$ respectively, and $\overline{h_1}$ and $\overline{h_2}$ are the high-level execution traces of $C|M_1$ and $C|M_2$ respectively.

Since i is odd, $i + 1$ is even and hence $h_1^{(i+1)} = h_2^{(i+1)}$ are ?-type actions. These actions are thus performed by the generated context C and the statements causing these actions have been generated in the construction phase of the algorithm. From the definition of the algorithm, we know that if $h_1^{(i+1)} = h_2^{(i+1)}$ are calls **call** $m.f(v_h)$? then the corresponding low-level actions $l_1^{(i)} = l_2^{(i)}$ are calls **call** $p(v_l)$?. Similarly, if they are return-backs **ret** v_h?, we know that the corresponding low-level actions $l_1^{(i)} = l_2^{(i)}$ are return-backs **ret** v_l?. In either case, the value v_h passed during these interactions is the lifted version of v_l. As can be seen from the definition of the lifting function, the lifted value is always equivalent to the original low-level value. Furthermore, the algorithm ensures that the next high-level statement to be executed in M_1 and M_2 (i.e., the call target locations or return locations) corresponds to the next instructions to be executed in $M_1{\downarrow}$ and $M_2{\downarrow}$ respectively. That is, the next instructions to be executed at the low level are the result of the translation (by the secure compiler) of the next statements to be executed at the high level. Hence, because (1) the original high- and low-level states were equivalent, (2) the newly passed-in values are equivalent, and (3) the next low-level instructions to be executed correspond to the next high-level statements to be executed, we can see that the

high- and low-level states are still equivalent after execution has entered M_1 and M_2 as the result of action $i+1$. We can then repeatedly apply Lemma 3.6.2 to see that the high- and low-level states will remain equivalent until right before action $i+2$. Now, because the states are still equivalent at this point, the high- and low-level actions will be similar as well. More precisely, the high- and low-level actions will be of the same kind and the high- and low-level values v_h and v_l passed as part of these interactions will also be equivalent, in the following sense.

- If v_h is of type *Unit*, $v_l = 0$.
- If v_h is of type *Int*, $v_l = v_h$.
- If v_h has function reference type $\langle T \to T' \rangle$, then $v_h = c.f$ for some function f that was *uniquely* associated with the pair $(v_l, \langle T \to T' \rangle)$ through the algorithm's function table τ.

Therefore, when $l_1^{(i+2)} \neq l_2^{(i+2)}$, we will also have $h_1^{(i+2)} \neq h_2^{(i+2)}$. □

Our full abstraction theorem now follows.

Theorem 3.6.4 (Full abstraction)**.** *For any two high-level modules M_1 and M_2, we have:*

$$M_1 \simeq M_2 \iff M_1{\downarrow} \simeq M_2{\downarrow}$$

Proof. As mentioned at the start of this section, we assume the soundness of our compiler and hence only prove completeness here. Thus, we need to prove the statement $M_1{\downarrow} \not\simeq M_2{\downarrow} \implies M_1 \not\simeq M_2$. From Lemma 3.5.4, we know that $M_1{\downarrow} \not\simeq M_2{\downarrow}$ implies $M_1{\downarrow} \not\simeq_{RT} M_2{\downarrow}$. Thus, $\mathrm{Tr}^{\mathrm{R}}(M_1{\downarrow}) \neq \mathrm{Tr}^{\mathrm{R}}(M_2{\downarrow})$, which means there is some low-level C_l such that the trace $\overline{l_1}$ generated by the execution of $C_l|M_1{\downarrow}$ is different from the trace $\overline{l_2}$ generated by the execution of $C_l|M_2{\downarrow}$. We can feed M_1, M_2 and these two revised low-level traces $\overline{l_1}$ and $\overline{l_2}$ into our algorithm, which will generate a high-level C that, according to Theorem 3.6.3, generates two different traces $\overline{h_1}$ and $\overline{h_2}$ in the executions $C|M_1$ and $C|M_2$ respectively. This means $M_1 \not\simeq_{HT} M_2$ and thus we can use Lemma 3.5.1 to see that $M_1 \not\simeq M_2$. □

3.7 Summary

We started this chapter by giving a formal definition of contextual equivalence. We then moved on to defining the high- and low-level languages we would be

working with. Our secure compiler was formalized in the form of an OCaml implementation that takes as input a high-level module M and returns a low-level protected module $M{\downarrow}$. To prove that this compiler is fully abstract, we first defined high- and low-level execution traces. We then argued that securely compiled modules perform only a restricted set of low-level interactions and that therefore we can use a simpler, revised form of low-level execution traces. We then used these traces to define an algorithm that, given two high-level modules M_1 and M_2 and corresponding non-equal revised low-level traces $\overline{t_1}$ and $\overline{t_2}$ generated by the interaction of $M_1{\downarrow}$ and $M_2{\downarrow}$ with a distinguishing low-level context C_l, produces a high-level context C_h that can distinguish M_1 from M_2. Finally, the existence of this algorithm was used to prove that our compiler implements a fully abstract translation from a high-level imperative programming language towards a processor featuring a fine-grained, program counter-based memory access control scheme.

Chapter 4

Formal software verification

Preamble

In this chapter we present background information on Hoare logic-based formal software verification techniques. Readers that are familiar with Hoare logic, separation logic and symbolic execution might prefer to skip this chapter. The chapter is a literature study and hence its contributions lie not in the originality of the technical matter, but in the selection and synopsis of the works discussed.

4.1 Introduction

The goal of software development is to write code that works, i.e., does what it is intended to do, is sufficiently performant, is maintainable and readable for other developers and does not introduce any security vulnerabilities. Unfortunately, history has shown that real-world software development often falls short of these objectives. Good quality software is estimated to have about one defect per 2,000 lines of code, while the accepted industry standard is even twice that number of defects [33]. The consequences of such defects include system malfunction, security vulnerabilities, and in the case of safety-critical systems even human injury [123]. The economic impact is significant, with annual productivity and turnover losses estimated at €1.6 billion in the Netherlands [118] and up to $59.5 billion in the US [82].

Over the years, researchers and practitioners have come up with various technical approaches for reducing the number of programming defects. The most well-

known and straightforward way of doing this is by *testing*, in which (a part of) the program is executed and checked to run correctly on a specific set of input values. A good testing approach involves choosing the set of input values such that all code paths and boundary conditions are covered. Unfortunately, for complex systems the number of possible code paths is so large that full test coverage is impossible to achieve. Since testing says nothing about the behavior of the system under test outside of the tested scenarios, its provided correctness guarantees are limited.

A different approach is to consider *formal verification* techniques, which can provide rigorous guarantees about a system's behavior for *all possible input values*. In general, formal software verification denotes techniques for checking the validity of certain properties about a rigorously specified mathematical model of a software system. Interesting properties to verify include memory safety, absence of race conditions and deadlocks, adherence to communication protocols, and full functional correctness. The model on which these properties are checked can range from the system's source code itself, over a compiled binary of the system, to a model defined separately in a different language, or a combination of these approaches. In any case, it is vital that both the model and the formal semantics for reasoning about the model truthfully reflect the actual system's behavior, for otherwise the verified properties will hold for the model but not for the system itself. This requisite is not always easy to achieve, since even when the model consists of an executable description of the system (such as its source code or a compiled binary), its runtime behavior will depend on the specific compiler and platform that will be used for its execution.

While formal verification methods have been used in the hardware industry since the 1990's [44], their adoption in the software industry is proceeding much more slowly. One of the main reasons for this, is the fact that the hard correctness guarantees offered by formal verification algorithms come at a price. Creating the mathematical model and formal description of the desired properties for a complex software system, is a difficult task, requiring substantial human effort. Experience with practical software verification tools has shown that the time spent on verifying a piece of software can easily exceed the time spent writing the original source code [100]. Hence, only when the impact of potential software defects outweighs the extra development cost of applying formal verification methods, does the approach become economically interesting. Nevertheless, there are a number of large, real-world software systems that have been the subject of formal verification in the past years, demonstrating its maturity and real-world feasibility. Three high-profile examples are the verification of Microsoft's Hyper-V hypervisor using the VCC verifier [35], the verification of seL4 [66], a micro-kernel of the L4 family, and CompCert [68], a formally verified C compiler.

In the rest of this chapter, we will focus on a specific set of formal verification methods, namely those based on Hoare logic. In Section 4.2 we discuss Hoare logic itself, in Section 4.3 we discuss separation logic and in Section 4.4 we discuss symbolic execution techniques based on separation logic. Finally, in Section 4.5 we summarize the chapter.

4.2 Hoare logic

Hoare logic provides a formal framework for reasoning about imperative software programs. Its foundations date back to the work of Floyd [46] and Hoare [53] in the 1960's. The key construct of Hoare logic is the Hoare triple, written $\{P\}\ C\ \{Q\}$. It denotes the statement "if the *precondition* P holds before executing the program C, then either C will end up in an infinite loop, or Q will hold when C has been executed". The assertions P and Q are formulas from a mathematical logic, typically taken to be first-order logic.

As an example, consider the Hoare triple below. Its precondition $\top$ is by definition always satisfied and hence the triple states that the program variable `x` will be equal to 5 if the program finishes.

$$\{\top\}\ \textbf{int}\ x := 0;\ \textbf{while}\ (x < 5)\ \textbf{do}\ x := x + 1\ \{x = 5\}$$

4.2.1 Formal semantics

In this section, we will formally define the semantics of Hoare logic. We start by defining the programming language we will reason about and the assertion language out of which we will draw the assertions we want to prove. We will then start from a model-theoretic viewpoint and define what it means for a Hoare triple to be valid, before considering the proof-theoretic approach in which we describe a proof calculus for proving Hoare triples.

Programming language

The programming language we will use in this section is the so-called *while programming language*. It is the programming language for which Hoare logic was originally introduced [53] and it provides only simple variable assignments, if-then-else statements and while statements as its basic constructs. Figure 4.1 formally defines the language's syntax and semantics. In this figure, the transition relation $\longrightarrow$ relates configurations, which are state-command pairs

$$E ::= n \mid x \mid E + E \mid E - E$$

$$B ::= \texttt{true} \mid \texttt{false} \mid E = E \mid E < E \mid \neg B \mid B \wedge B \mid B \vee B$$

$$C ::= \textbf{skip} \mid x := E \mid \textbf{if } B \textbf{ then } C \textbf{ else } C \mid \textbf{while } B \textbf{ do } C \mid C; C$$

$$n \in \mathbb{N} \qquad x \in \mathit{Vars}$$

$$\frac{[\![B]\!]_s = \texttt{true}}{\langle s, \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2\rangle \longrightarrow \langle s, C_1\rangle} \qquad \frac{[\![B]\!]_s = \texttt{false}}{\langle s, \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2\rangle \longrightarrow \langle s, C_2\rangle}$$

$$\frac{[\![B]\!]_s = \texttt{true}}{\langle s, \textbf{while } B \textbf{ do } C\rangle \longrightarrow \langle s, C; \textbf{while } B \textbf{ do } C\rangle} \qquad \frac{[\![B]\!]_s = \texttt{false}}{\langle s, \textbf{while } B \textbf{ do } C\rangle \longrightarrow \langle s, \textbf{skip}\rangle}$$

$$\frac{[\![E]\!]_s = n}{\langle s, x := E\rangle \longrightarrow \langle s[x \to n], \textbf{skip}\rangle} \qquad \frac{}{\langle s, \textbf{skip}; C\rangle \longrightarrow \langle s, C\rangle} \qquad \frac{\langle s, C_1\rangle \longrightarrow \langle s', C_1'\rangle}{\langle s, C_1; C_2\rangle \longrightarrow \langle s', C_1'; C_2\rangle}$$

Figure 4.1: Syntax and small step operational semantics for the *while programming language*. The notation $[\![E]\!]_s$ represents the evaluation of expression E under the store s (which we assume has a mapping for each of the free variables of E), using the natural interpretation of the available arithmetic and logical operators.

$\langle s, C\rangle$. The state consists of only a store s, which is a partial function mapping variable names to integers. We write $\langle s, C\rangle \longrightarrow^* \langle s', C'\rangle$ to indicate there is a finite sequence of transitions from $\langle s, C\rangle$ to $\langle s, C'\rangle$ and we write $\langle s, C\rangle \uparrow$ to indicate that $\langle s, C\rangle$ diverges. Each inference rule is deterministic, and hence any configuration either diverges or results in exactly one end state. In this section and the rest of this chapter, we always assume programs are *well-formed*, meaning that expressions never reference undefined variables.

Assertion semantics

We will use first-order logic as the language from which we can draw assertions for our Hoare triples, as is common in Hoare logic literature. Figure 4.2 defines the syntax and semantics of this logic, using a relation $s \vDash P$ which asserts that the assertion P holds under state s.

Given this logic, we can define what it means for a Hoare triple to be *valid* or not. Informally, we say a triple $\{P\}\ C\ \{Q\}$ is valid if, starting from a state where the precondition P holds, the postcondition Q holds after C has been executed. We write this statement as $\vDash \{P\}\ C\ \{Q\}$ and we can formally define its meaning as follows.

$$P, Q, R ::= \; B \mid P \Rightarrow Q \mid P \wedge Q \mid \exists x.\, P$$

$s \vDash E = E$	iff $[\![E]\!]_s = [\![E]\!]_s$	$s \vDash E < E$	iff $[\![E]\!]_s < [\![E]\!]_s$
$s \vDash P \Rightarrow Q$	iff if $s \vDash P$ then $s \vDash Q$	$s \vDash \texttt{false}$	never
$s \vDash \exists x.\, P$	iff $\exists v \in \mathbb{N}.\, s[x \to v] \vDash P$	$s \vDash P \wedge Q$	iff $s \vDash P$ and $s \vDash Q$

Figure 4.2: First-order logic syntax and semantics. Other logical connectives can be defined in terms of these primitives: $\neg P \mathrel{\hat{=}} P \Rightarrow \texttt{false}$; $\texttt{true} \mathrel{\hat{=}} \neg\texttt{false}$; $P \vee Q \mathrel{\hat{=}} (\neg P) \Rightarrow Q$; $\forall x.P \mathrel{\hat{=}} \neg\exists x.\, \neg P$.

Definition 4.2.1 (Validity of a Hoare triple for partial correctness)**.**

$$\vDash \{P\}\, C\, \{Q\} \;\; \textit{iff}$$

$$\forall s, s'.\, (s \vDash P \wedge \langle s, C\rangle \longrightarrow^* \langle s', \textbf{\textit{skip}}\rangle) \implies s' \vDash Q$$

Notice that this definition gives no guarantees if C diverges. That is, if C does not terminate then $\{P\}\, C\, \{Q\}$ is valid. For this reason, we say $\{P\}\, C\, \{Q\}$ are *partial* correctness triples and we write *total* correctness triples, which require C to terminate, as $[P]\, C\, [Q]$. Validity of total correctness triples is defined as follows.

Definition 4.2.2 (Validity of a Hoare triple for total correctness)**.**

$$\vDash [P]\, C\, [Q] \;\; \textit{iff}$$

$$\forall s.\, s \vDash P \implies (\exists s'.\, \langle s, C\rangle \longrightarrow^* \langle s', \textbf{\textit{skip}}\rangle \wedge s' \vDash Q)$$

In the rest of this chapter we will focus primarily on partial correctness triples.

Notice that in both definitions the stores s and s' can contain variables that are not referenced anywhere in the program C. These are called *auxiliary variables* and can be used to relate a triple's pre- and postcondition. The following triple, for instance, is valid and uses an auxiliary variable y.

$$\{x = y\}\; x := x + 1\; \{x = y + 1\}$$

Proof calculus

Given a program C and pre- and postconditions P and Q that do *not* hold for C, we can directly show a negative result $\nvDash \{P\}\, C\, \{Q\}$ by finding a counterexample,

$$\frac{}{\{P\}\,\mathbf{skip}\,\{P\}}\ \text{H0} \qquad \frac{}{\{P[E/x]\}\,x := E\,\{P\}}\ \text{H1} \qquad \frac{\{P\}\,C_1\,\{Q\} \qquad \{Q\}\,C_2\,\{R\}}{\{P\}\,C_1; C_2\,\{R\}}\ \text{H2}$$

$$\frac{\{B \wedge P\}\,C\,\{P\}}{\{P\}\,\mathbf{while}\ B\ \mathbf{do}\ C\,\{\neg B \wedge P\}}\ \text{H3} \qquad \frac{\{B \wedge P\}\,C_1\,\{Q\} \qquad \{\neg B \wedge P\}\,C_2\,\{Q\}}{\{P\}\,\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\,\{Q\}}\ \text{H4}$$

$$\frac{\{P\}\,C\,\{Q\} \qquad R \Rightarrow P}{\{R\}\,C\,\{Q\}}\ \text{H5-PRE} \qquad \frac{\{P\}\,C\,\{Q\} \qquad Q \Rightarrow S}{\{P\}\,C\,\{S\}}\ \text{H5-POST}$$

Figure 4.3: Hoare logic syntax, axioms and inference rules for proving partial correctness properties for *while programs*. The term $P[y/x]$ denotes the assertion P with all occurrences of x replaced by y.

Proof.

$\{\top\}$	PRECONDITION
$\{0 = 0\}$	H5-PRE
`int x := 0;`	
$\{x = 0\}$	H1
$\{x <= 5\}$	H5-POST
`while(x < 5) do {`	
$\quad\{x < 5 \wedge x <= 5\}$	GUARD $\wedge$ INVARIANT
$\quad\{x < 5\}$	H5-PRE
$\quad\{x + 1 <= 5\}$	H5-PRE
`  x := x + 1;`	
$\quad\{x <= 5\}$	H1
`}`	
$\{x >= 5 \wedge x <= 5\}$	H3
$\{x = 5\}$	H5-POST

□

Figure 4.4: This *proof tableaux* proves the example Hoare triple $\{\top\}$ **int** $x :=$ 0; **while**$(x < 5)$ **do** $x := x + 1\,\{x = 5\}$. The proof starts from the precondition $\top$ and ends with the postcondition $x = 5$, justifying each step in between by referring to the Hoare rules listed in Figure 4.3.

i.e., a state such that the implication in Definition 4.2.1 does not hold. However, since our ultimate goal is to use Hoare logic for program verification, we are also interested in positive results $\vDash \{P\}\, C\, \{Q\}$, for some interesting program property expressed by P and Q. Since this is a statement about *all possible* states, of which there are an infinite amount, it is often difficult to show such results using the model-based approach above. Instead, Hoare logic provides a calculus of axioms and inference rules for *proving* such triples. Figure 4.3 shows the standard set of Hoare rules for the while programming language, as they were defined in [53]. We write $\vdash \{P\}\, C\, \{Q\}$ to express that a triple is derivable from these rules. Figure 4.4 shows how these rules can be used to prove the example triple shown at the start of this section.

The Hoare logic rules are designed to capture the meanings of the individual constructs out of which the while programming language is constructed. Notice that they are designed for *backward* reasoning, since each of the rules can be applied for an arbitrary postcondition. In particular, the assignment axiom H1 shows how to derive a precondition $P[y/x]$ from an arbitrary postcondition P. The equivalent forward reasoning axiom was given by Floyd [46] and is as follows.

$$\frac{}{\{P\}\, x := E\, \{\exists x'.\, x = E[x'/x] \wedge P[x'/x]\}}\ \text{H1-FWD}$$

The basic rules of Figure 4.3 are often extended with the three structural rules shown below, which can potentially simplify proofs. In each these rules, $\mathsf{FV}(C)$ is the set of free variables in C and $\mathsf{mod}(C)$ is the set of variables assigned in C. The first rule is the auxiliary variable elimination rule, which allows auxiliary variables to be replaced by existentially quantified logic variables.

$$\frac{\{P\}\, C\, \{Q\}}{\{\exists x.\, P\}\, C\, \{\exists x.\, Q\}}\ \text{H6}\ \ (\text{where } x \notin \mathsf{FV}(C))$$

The second rule is the variable substitution rule, which allows free variables in P, C and Q to be renamed or replaced by an expression.

$$\frac{\{P\}\, C\, \{Q\}}{(\{P\}\, C\, \{Q\})[E_1/x_1, \ldots, E_k/x_k]}\ \text{H7}$$

(where $\{x_1, \ldots, x_k\} \supseteq \mathsf{FV}(P, C, Q)$, and $x_i \in \mathsf{mod}(C)$ implies that E_i is a variable not occurring free in any other E_j)

The third rule is the rule of constancy, also known as the *simple frame rule*.

$$\frac{\{P\}\, C\, \{Q\}}{\{P \wedge R\}\, C\, \{Q \wedge R\}}\ \text{H8}\qquad (\text{where } \mathsf{FV}(R) \cap \mathsf{mod}(C) = \emptyset)$$

The simple frame rule allows extending local specifications of C to global specifications including assertions about variables that are not modified by C.

The rule is sound because the values of variables not modified by C remain constant when executing C. It was soon recognized that this rule is vital for the scalability of proofs [105, Section 3.3.5], since it enables modular reasoning: a proof of a small program fragment C can now easily be incorporated into the proof of a larger program that contains C.

Total correctness As for proving *total* correctness triples, the only program construct that can cause non-termination is the while loop. Hence, the only inference rule that must be adapted to reason about total correctness is H3. Manna and Pnueli [71] were the first to present such a rule, based on a technique described earlier by Floyd [46]. The technique consists of finding a numeric expression that (1) is larger than some lower bound while executing the loop and (2) strictly decreases with every loop iteration. Eventually the expression must reach the lower bound and hence the loop must terminate. The total correctness version of H3 is shown below.

$$\frac{[B \wedge P \wedge (E = n)]\, C\, [P \wedge (E < n)] \qquad P \wedge B \implies E \geq 0}{[P]\ \textbf{while}\ B\ \textbf{do}\ C\ [\neg B \wedge P]}\ \text{H3-Total}$$

4.2.2 Soundness and completeness

The relation between validity and provability is determined by a logic's soundness and completeness properties. For a sound logic, we have that $\vdash \{P\}\, C\, \{Q\}$ implies $\vDash \{P\}\, C\, \{Q\}$ and the opposite implication holds for a complete logic.

Soundness

Since the goal of program verification is to prove program correctness properties, we need our logic to be sound, for otherwise our verifier could assert properties that do not actually hold. Given a formal semantics for a programming language, we can prove the soundness of a set of Hoare logic rules with respect to these semantics. For instance, below we prove the soundness of Hoare rules H2 and H3 (the other rules are trivial) with respect to the language defined in Figure 4.1.

Soundness proof of Hoare rule H2. There are two cases to consider:

- C_1 contains an infinite loop and hence never evaluates to **skip**. In this case, $C_1; C_2$ also contains an infinite loop and since we are only concerned with partial correctness, H2 holds.

- C_1 evaluates to **skip** with an updated store s', after n evaluation steps (n can be 0 and s' can be equal to s). From the first premise of $H2$, we can see that Q will hold in s'. Since $\langle s', \textbf{skip}; C_2\rangle \longrightarrow \langle s', C_2\rangle$, we now have to prove $\{Q\}\, C_2\, \{R\}$, which is trivial because it is the second premise of H2.

□

Soundness proof of Hoare rule H3. Since we are only concerned with partial correctness, the rule holds by definition if the loop does not terminate. If the loop ends after n iterations, we can prove the rule by induction on n.

- For $n = 0$, we have $[\![B]\!]_s = \texttt{false}$ and consequently $\langle s, \textbf{while}\ B\ \textbf{do}\ C\rangle \longrightarrow \langle s, \textbf{skip}\rangle$. We now need to prove $\{P\}\,\textbf{skip}\,\{\neg B \wedge P\}$, which follows directly from H0 and our assumption of $[\![B]\!]_s = \texttt{false}$.
- Assuming the rule applies for $n = i$, we can prove that it also holds for $n = i + 1$ as follows. We know $[\![B]\!]_s = \texttt{true}$, and consequently $\langle s, \textbf{while}\ B\ \textbf{do}\ C\rangle \longrightarrow \langle s, C;\ \textbf{while}\ B\ \textbf{do}\ C\rangle$, hence our goal reduces to proving $\{B \wedge P\}\, C;\ \textbf{while}\ B\ \textbf{do}\ C\, \{\neg B \wedge P\}$. According to H2, we can do so by proving $\{B \wedge P\}\, C\, \{Q\}$ and $\{Q\}\ \textbf{while}\ B\ \textbf{do}\ C\, \{\neg B \wedge P\}$ for some Q. We can choose $Q = P$ so that the first triple follows directly from the premise of H3 and the second triple follows from the induction hypothesis.

□

Completeness

Soundness says nothing about triples that can *not* be proved. For instance, the empty set of inference rules can in principle be considered sound. For practical software verification purposes, we are also interested in some form of *completeness*. A complete set of inference rules allows proving any valid triple. Unfortunately, Gödel's first incompleteness theorem says that any enumerable theory capable of expressing elementary arithmetic cannot be both sound and complete, and hence we cannot expect Hoare logic with first-order logic as the underlying assertion language to be complete. This certainly does not imply that Hoare logic and practical verification algorithms are useless: many interesting properties can be proved using an incomplete logic.

An interesting question to ask is whether Hoare logic's incompleteness stems from the complexity of the programming language we are dealing with, or from

the incompleteness of the underlying logic in which we write our assertions. If the underlying logic is incomplete, there are valid implications that cannot be proved in the logic, and since the premises of rules H5-Pre and H5-Post contain implications, there are also valid Hoare triples that cannot be proved. Cook [32] showed in 1978 that Hoare logic for while programs is *relatively complete*: if the underlying logic is complete and sufficiently expressive to formulate the necessary assertions of our Hoare proofs, then Hoare logic itself is also complete. While Cook's original proof holds for partial correctness properties of while programs extended with non-recursive procedures, others have extended his work to include total correctness properties [111] and recursive procedures [16, 50].

4.2.3 Hoare logic as a language definition

We have so far considered Hoare logic purely as a system for reasoning about programs written in a separately-defined imperative programming language. We first defined the formal semantics of a programming language and base logic, and we then defined the set of Hoare inference rules followed by a proof that these rules are sound with regards to the language semantics.

There is, however, an alternative way to look at a set of Hoare-style inference rules: we can consider them to be the definitive *specification* of a programming language's semantics. This interpretation was in fact proposed in Hoare's original paper [53] as a way of capturing the essentials of a programming language, while leaving undefined certain implementation-level details, such as for instance the range of integer values and overflow handling. The advantage of this approach as opposed to, for instance, specifying operational semantics, is that it allows implementations of the language to be verified against its formal specification, while still giving implementers enough freedom to create an efficient implementation for a particular target platform. For our further discussion of formal software verification, it does not matter which interpretation we employ, as long as our Hoare logic is sound with respect to our programming language semantics.

4.2.4 Extensions and limitations

The while programming language we have been working with so far has only the most basic constructs expected from an imperative programming language. Many extensions to Hoare logic have been proposed over the years, to move from a toy programming language towards a language similar to Algol-60. For instance, researchers have described how to add support for recursive

procedures [54, 109, 90] with various parameter passing mechanisms [11], non-determinism [84, 12] and higher-order functions [27, 11]. These extensions make Hoare logic compatible with most, but not all, features of Algol-60. In fact, Clarke [27] has shown that there are certain sets of programming language constructs for which it is impossible to obtain a sound and relatively complete Hoare-style logic. For instance, it is impossible to obtain a sound and relatively complete Hoare logic for a language featuring simultaneously (1) higher-order functions, (2) recursion, (3) static scoping, (4) global variables and (5) nested functions. This shows that full Algol-60 cannot have an adequate Hoare-style logic. However, Clarke also showed that a sound and complete Hoare logic *can* be obtained by modifying any one of these features, while not adding any additional features.

Pointers

One particular extension that has received a lot of attention in literature is support for pointers to shared mutable data structures [24, 78, 21]. Such pointers appear explicitly in commonly used system programming languages such as C and C++, and implicitly in modern-day general-purpose programming languages such as Java and C#.

The most significant difficulty of reasoning about pointer programs has to do with *aliasing*: the fact that two different pointers can refer to a single shared field. Take for instance the following Hoare triple, where q and r are assumed to be pointer variables and $[\cdot]$ is the heap access operator.

$$\{\top\}\ q := r;\ [q] := 5\ \{q = r \wedge [q] = 5 \wedge [r] = 5\}$$

Intuitively, we can see that this triple is valid, but standard Hoare logic does not allow us to prove it. More concretely, reasoning backwards from the postcondition using the standard Hoare logic rules would leave us with the too strong precondition of $\{[r] = 5\}$.

It is possible to generalize the Hoare assignment axiom to express that the value of *any alias* of the pointee being assigned also changes. Unfortunately, this now gives the logic a *global* character instead of a *local* one. Instead of being able to handle assignment by a simple syntactic substitution of the variable being assigned, we now have to take into account all variables that can potentially be aliases of the variable being assigned. This complicates our Hoare logic proofs, hampers modular reasoning and, more fundamentally, is at odds with the fact that pointer assignment is operationally local. That is, operationally a pointer assignment corresponds to a single, local memory write, but it can now affect an arbitrary number of variables in our Hoare logic assertions. Hence

there appears to be a mismatch between the axiomatic treatment of pointers as proposed, and the intuitions behind them [88].

This mismatch led researchers to pursue the principle of *spatial separation* as a way of regaining local reasoning for pointer programs. The ideas behind this principle were already present in the work of Burstall [24], but were made explicit in the later work of O'Hearn, Reynolds and Yang [88, 104]. The central idea behind spatial separation is that the heap can be split into *disjoint* components for which different assertions hold. When a heap cell assignment modifies a certain component, it affects only the assertions related to that component and leaves the other assertions untouched. Specifications can then focus only on the heap cells modified by a certain program fragment (i.e., the so-called *footprint* of that fragment). This is the intuition behind *separation logic*, which is discussed in detail in the next section.

4.3 Separation logic

Separation logic is an extension of Hoare logic for reasoning about imperative programs with pointers to shared mutable data structures. It was developed by Reynolds, O'Hearn, Ishtiaq and Yang [104, 88, 57] around the year 2000, but is rooted in the earlier work of Burstall [24] published in 1972.

Pointers appear implicitly in high-level general-purpose programming languages such as C# and Java to implement object references. For simplicity, these kinds of programming languages shield programmers from working with pointers directly. System programming languages such as C and C++, on the other hand, provide pointers as an explicit construct and hence allow them to be manipulated directly by programmers. For instance, programmers can apply arithmetic operations to pointer variables and can dereference them at will. Such manipulations are very powerful, but are also notoriously difficult to get right. For instance, dereferencing a pointer to unallocated memory can cause a program to crash and can even lead to memory safety bugs that eventually allow execution of attacker-injected code [122]. Detecting such bugs is beyond the scope of conventional type systems, and more advanced type systems for low-level programs suffer from reduced performance due to additional type or bounds information that must be carried along with pointers and checked at runtime [64, 79, 30].

Because of their excellent performance and low-level characteristics, system programming languages have been very popular over the last decades and will likely remain popular for time to come. At the same time, the risks of working with pointers combined with the fact that these languages are commonly used

for implementing embedded safety-critical systems, make them an interesting target for formal verification. Unfortunately, as argued in Section 4.2.4, the standard Hoare logic rule for assignment (i.e., rule H1 in Figure 4.3) is unsound for reasoning about pointer programs. The problem is that assignment to a memory cell through a pointer can affect the value of an arbitrary number of seemingly unrelated expressions in Hoare logic assertions.

Separation logic relies upon the principle of *spatial separation* to mitigate this problem. The key idea is to introduce a new logical operation $P * P'$, called the *separating conjunction*, which asserts that P and P' hold for *disjoint* portions of the heap [104]. In addition, separation logic introduces a new type of assertion $E_1 \mapsto E_2$, called the *points-to assertion*, which expresses that the heap is a singleton with address E_1 containing value E_2 (both expressions are to be evaluated under a specific store). Arbitrary heaps can be described by combining these two assertions. For instance, the assertion $(x \mapsto 5) * (y \mapsto 8)$ describes a heap containing the values 5 and 8 (and no other values), at the addresses contained in the program variables x and y respectively. The combination of $*$ and $\mapsto$ leads to a very natural global heap mutation axiom:

$$\overline{\{(E \mapsto -) * P\}\,[E] := E'\,\{(E \mapsto E') * P\}}$$

Here, the wildcard pattern $E \mapsto -$ means that E points to some value on the heap, but that value is unspecified. A heap mutation now has a local effect on assertions: only the points-to assertion of E is affected by an assignment to $[E]$. It is the *disjointness* expressed by the separating conjunction that enables this kind of local reasoning.

4.3.1 Formal semantics

In this section, we will define the formal semantics of separation logic. We follow the same approach as in the Hoare logic section, by first defining the programming language to reason about and the assertion language out of which we will draw the assertions we want to prove. We then define what it means for a separation logic triple to be valid, and finally give a proof calculus for proving such triples.

Programming language

The programming language to reason about in this section is the *pointer programming language*. It is an extension of the while programming language of the previous section, with new commands for heap allocation, deallocation,

$$
\begin{array}{lll}
C ::= & \ldots & \textit{command extension} \\
& \mid x := \textbf{alloc} \mid x := [E] & \\
& \mid [E] := E \mid \textbf{dealloc}(E) &
\end{array}
$$

$$\frac{\ell \in \mathbb{N}_0^+ \qquad \ell \mod 2 = 0 \qquad \{\ell, \ell+1\} \cap \operatorname{dom} h = \emptyset \qquad v_1, v_2 \in \mathbb{N}}{\langle (s,h),\, x := \textbf{alloc} \rangle \longrightarrow \langle (s[x \to \ell], h[\ell \to v_1, \ell+1 \to v_2]),\, \textbf{skip} \rangle}\ \textsc{Allocation}$$

$$\frac{[\![E]\!]_s \in \operatorname{dom} h}{\langle (s,h),\, x := [E] \rangle \longrightarrow \langle (s[x \to h([\![E]\!]_s)], h),\, \textbf{skip} \rangle}\ \textsc{Lookup-ok}$$

$$\frac{[\![E]\!]_s \in \operatorname{dom} h}{\langle (s,h),\, [E] := E' \rangle \longrightarrow \langle (s, h[[\![E]\!]_s \to [\![E']\!]_s]),\, \textbf{skip} \rangle}\ \textsc{Mutate-ok}$$

$$\frac{[\![E]\!]_s \in \operatorname{dom} h \qquad [\![E]\!]_s \mod 2 = 0}{\langle (s,h),\, \textbf{dealloc}(E) \rangle \longrightarrow \langle (s, h \setminus \{[\![E]\!]_s, [\![E]\!]_s + 1\}),\, \textbf{skip} \rangle}\ \textsc{Dealloc-ok}$$

$$\frac{[\![E]\!]_s \notin \operatorname{dom} h}{\langle (s,h),\, [E] := E' \rangle \longrightarrow \textbf{fail}}\ \textsc{Mutate-fail} \qquad \frac{[\![E]\!]_s \notin \operatorname{dom} h}{\langle (s,h),\, x := [E] \rangle \longrightarrow \textbf{fail}}\ \textsc{Lookup-fail}$$

$$\frac{[\![E]\!]_s \notin \operatorname{dom} h}{\langle (s,h),\, \textbf{dealloc}(E) \rangle \longrightarrow \textbf{fail}}\ \textsc{Dealloc-F} \qquad \frac{[\![E]\!]_s \mod 2 \neq 0}{\langle (s,h),\, \textbf{dealloc}(E) \rangle \longrightarrow \textbf{fail}}\ \textsc{Dealloc-F}$$

$$\frac{\langle (s,h),\, C_1 \rangle \longrightarrow \langle (s',h'),\, C_1' \rangle}{\langle (s,h),\, C_1; C_2 \rangle \longrightarrow \langle (s',h'),\, C_1'; C_2 \rangle}\ \textsc{Seq} \qquad \frac{\langle (s,h),\, C_1 \rangle \longrightarrow \textbf{fail}}{\langle (s,h),\, C_1; C_2 \rangle \longrightarrow \textbf{fail}}\ \textsc{Seq-fail}$$

Figure 4.5: Syntax and small step operational semantics extensions for the *pointer programming language*, which adds a heap to the while programming language defined in Figure 4.1. Configurations are still state-command pairs, but states now contain a heap h in addition to a store s. All transition rules of the while programming language, except the sequencing rule, still apply, and none of them have any effect on the heap.

lookup and mutation. Figure 4.5 defines the language's syntax and semantics. To avoid redundancy, we only define the semantics for the new commands and refer to Figure 4.1 for the commands inherited from the while programming language.

The first important difference to notice between the pointer programming language and the while programming language, is that the program state now contains a heap h in addition to a store s. The heap is a partial function from positive integers to integers. Another difference is that the language now contains commands that can fail. More specifically, reading, writing and deallocating an unallocated heap address are considered illegal operations that will cause a transition to the **fail** configuration. Notice also that the language is now non-deterministic, since both the start address ℓ of an allocated memory block and its initial values v_1 and v_2 are unspecified in the allocation rule. Hence a configuration $\langle (s, h), C \rangle$ can now diverge, fail, or transition in a finite number of steps into an arbitrary number of final states. Finally, notice that $[E]$ is not an expression and hence cannot appear as a sub-expression of a larger expression; it can only appear directly on the left or right side of the assignment operator. Hence, expressions never depend on the heap and can still be evaluated against a store only.

Assertion semantics

As mentioned above, separation logic introduces a number of new types of assertions that make statements about the heap. Figure 4.6 formally defines the syntax and semantics of these new assertions. Since the validity of these assertions depends on the heap, they can only be evaluated against states consisting of both a heap and a store.

Intuitively, we can associate the following meanings with each of the new assertions: **emp** asserts that the heap is empty; the points-to assertion $E \mapsto E'$ asserts that the heap contains exactly one value E', at address E; the separating conjunction $P * Q$ asserts that the heap can be split into two *disjoint* parts on which P and Q hold individually; and the separating implication (also called the *magic wand*) $P -\!\!* Q$ asserts that if we extend the heap with a disjoint fragment where P holds, then Q will hold on the extended heap. Note that we will sometimes write $E \mapsto E', E''$ as syntactic sugar for $(E \mapsto E') * (E + 1 \mapsto E'')$.

Based on these definitions, we can define what it means for a separation logic triple to be valid under partial and total correctness semantics.

$$
\begin{array}{rl}
P, Q, R ::= & \ldots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{assertion extension} \\
& \mid \mathbf{emp} \mid E \mapsto E \mid P * P \mid P \mathrel{-\!\!*} P
\end{array}
$$

$$
\begin{array}{rll}
(s,h) \vDash \mathbf{emp} & \text{iff} & \operatorname{dom} h = \emptyset \\
(s,h) \vDash E \mapsto E' & \text{iff} & \operatorname{dom} h = \{[\![E]\!]_s\} \text{ and } h([\![E]\!]_s) = [\![E']\!]_s \\
(s,h) \vDash P * Q & \text{iff} & \exists h_0, h_1.\, h_0 \perp h_1 \text{ and } h_0 \cup h_1 = h \text{ and} \\
 & & (s,h_0) \vDash P \text{ and } (s,h_1) \vDash Q \\
(s,h) \vDash P \mathrel{-\!\!*} Q & \text{iff} & \forall h'.\, (h' \perp h \text{ and } (s,h') \vDash P) \text{ implies } (s, h \cup h') \vDash Q
\end{array}
$$

Figure 4.6: Separation logic assertions syntax and semantics. These assertions extend the definitions of Figure 4.2, which all still apply to separation logic, assuming we extend the state s before the entails symbol to include a heap h. The notation $h \perp h'$ means the domains of h and h' are disjoint.

Definition 4.3.1 (Validity of a separation logic triple for partial correctness)**.**

$\vDash \{P\}\, C\, \{Q\}$ *iff*

$$
\begin{array}{l}
\forall s, h.\ (s,h) \vDash P \implies \\
\quad [\langle (s,h),\, C\rangle \not\longrightarrow {}^{*}\boldsymbol{fail} \wedge \\
\quad (\forall s', h'.\, \langle (s,h),\, C\rangle \longrightarrow^{*} \langle (s',h'),\, \boldsymbol{skip}\rangle \implies (s',h') \vDash Q)]
\end{array}
$$

Definition 4.3.2 (Validity of a separation logic triple for total correctness)**.**

$\vDash [P]\, C\, [Q]$ *iff*

$$
\begin{array}{l}
\forall s, h.\ (s,h) \vDash P \implies \\
\quad [\langle (s,h),\, C\rangle \not\longrightarrow {}^{*}\boldsymbol{fail} \wedge \neg(\langle (s,h),\, C\rangle \uparrow) \wedge \\
\quad (\forall s', h'.\, \langle (s,h),\, C\rangle \longrightarrow^{*} \langle (s',h'),\, \boldsymbol{skip}\rangle \implies (s',h') \vDash Q)]
\end{array}
$$

Notice that partial nor total correctness triples are valid for executions that fail. Thus, programs that have been verified to meet a separation logic specification will not perform illegal memory reads, writes or deallocations.

$$\frac{}{\{\mathbf{emp}\}\ x := \mathbf{alloc}\ \{(x \mapsto -) * (x+1 \mapsto -)\}}\ \textsc{Allocate}$$

$$\frac{}{\{E \mapsto n \wedge x = m\}\ x := [E]\ \{x = n \wedge E[m/x] \mapsto n\}}\ \textsc{Lookup}$$

$$\frac{}{\{E \mapsto -\}\ [E] := E'\ \{E \mapsto E'\}}\ \textsc{Mutate}$$

$$\frac{}{\{(E \mapsto -) * (E+1 \mapsto -)\}\ \mathbf{dealloc}(E)\ \{\mathbf{emp}\}}\ \textsc{Deallocate}$$

Figure 4.7: Small separation logic axioms for the heap access commands introduced in Figure 4.5.

Within the language of separation logic assertions, we can identify a subclass of *precise* assertions [89], defined as follows.

Definition 4.3.3 (Precise assertions)**.** *An assertion P is precise iff for all states (s, h) there is at most one sub-heap $h' \subseteq h$ where $(s, h') \vDash P$*

A precise assertion unambiguously identifies a region of memory, namely the sub-heap h' in the above definition. For instance, the assertion `true` is not precise, while the assertion **emp** is.

Proof calculus

We can now define the proof calculus that will allow us to reason about pointer programs. Since expressions do not depend on the heap, all the Hoare logic rules and axioms defined in Figure 4.3 remain valid. Figure 4.7 shows the so-called *small axioms* for reasoning about the heap access commands introduced in the pointer programming language. Similar to the basic Hoare rules H0-H4, these small axioms succinctly express the semantics of the new commands. For reasoning about actual pointer programs, they are not very convenient though, since they are only directly applicable to programs that use at most a single heap cell. To enable reasoning about larger programs, separation logic includes the four *structural* rules shown in Figure 4.8. The first three of these rules are identical to Hoare rules H5, H6 and H7 respectively, while the third rule, the *frame rule*, is similar to Hoare logic's rule of constancy. By combining the small axioms and these structural rules, more practical global versions of the small axioms can be derived. For instance, the global version of the mutation axiom is as follows.

$$\frac{}{\{(E \mapsto -) * R\}\ [E] := E'\ \{(E \mapsto E') * R\}}\ \textsc{Mutation-global}$$

Consequence

$$\frac{P' \Rightarrow P \qquad \{P\}\, C\, \{Q\} \qquad Q \Rightarrow Q'}{\{P'\}\, C\, \{Q'\}}$$

Auxiliary variable elimination

$$\frac{\{P\}\, C\, \{Q\}}{\{\exists x.\, P\}\, C\, \{\exists x.\, Q\}} \qquad \text{(where } x \notin \mathsf{FV}(C)\text{)}$$

Variable substitution

$$\frac{\{P\}\, C\, \{Q\}}{(\{P\}\, C\, \{Q\})[E_1/x_1, \ldots, E_k/x_k]}$$

(where $\{x_1, \ldots, x_k\} \supseteq \mathsf{FV}(P, C, Q)$, and $x_i \in \mathsf{mod}(C)$ implies that E_i is a variable not occurring free in any other E_j)

Frame rule

$$\frac{\{P\}\, C\, \{Q\}}{\{P * R\}\, C\, \{Q * R\}} \qquad \text{(where } \mathsf{FV}(R) \cap \mathsf{mod}(C) = \emptyset\text{)}$$

Figure 4.8: Separation logic structural inference rules.

It is also possible to derive axioms that can be applied for any postcondition and hence are suitable for backward reasoning. The backward reasoning version of the mutation axiom is as follows.

$$\frac{}{\{(E \mapsto -) * ((E \mapsto E') \mathrel{-\!\!*} P)\}\, [E] := E'\, \{P\}}\ \textsc{Mutation-bwd}$$

Global and backward reasoning versions of the other axioms can be found in [104].

The frame rule is of particular importance in separation logic. It expresses exactly the property that was lost when trying to apply Hoare logic to pointer programs: the principle of local reasoning. That is, the frame rule codifies the idea that a program that executes successfully in a small store and heap will behave the same when executed in a bigger store and heap and will have no influence on the additional parts of the state. The reason this rule is sound, is because the precondition in a valid separation logic triple $\{P\}\, C\, \{Q\}$ represents not only a sufficient logical property for C to execute, but also a sufficient area of heap storage. Any heap location read, written or deallocated by C (i.e., the *footprint* of C) must be specified by a points-to assertion in P. Hence, any extension R of the precondition that describes a *disjoint* heap and does not mention any variables modified by C, will continue to hold after executing C.

4.3.2 Soundness and completeness

As described in Section 4.2.2, the basic Hoare rules H0-H4, combined with the structural consequence rule H5, constitute a sound and relatively complete proof calculus for the while programming language. The other structural rules H6-H8 are superfluous for relative completeness.

For pointer programs however, the small axioms defined in Figure 4.7 together with the consequence rule are not enough to obtain relative completeness. This can easily be seen from the following example triple.

$$\{\mathbf{emp}\}\ x := \mathbf{alloc}; y := \mathbf{alloc}\ \{(x \mapsto -,-) * (y \mapsto -,-)\}$$

This triple is valid, but cannot be derived from the small axioms alone, because the allocation rule can only be applied starting from states with an empty heap, and the heap will not be empty after executing the first command. This shows that we need at least the frame rule in order to reason about arbitrary heap structures. The full set of separation logic rules, i.e., the small axioms together with the basic Hoare rules as defined in Figure 4.3 and the structural separation logic rules of Figure 4.8, has been shown to be sound and relatively complete by Tatsuta et al. [116] in 2009[1].

4.3.3 Extensions

In this subsection, we briefly discuss how separation logic can be extended to support abstract inductive predicates and concurrent programs.

Abstract inductive predicates

Pointers are commonly used to implement recursive data structures such as lists and trees. The assertion language used in this section so far has no means of representing such arbitrary-length data structures. The logic could be extended ad-hoc with new predicates for representing specific data structures, but this would be a rather inflexible solution, since users would be limited to using only the built-in predicates. Instead, separation logic can be extended to support *user-defined inductive predicates*, which allow users to model their own data structures in the assertion logic. For instance, the following inductive definition

[1] Actually, Tatsuta showed that the set of *global* separation logic axioms is sound and relatively complete, but these global rules can be derived from the small axioms and the structural rules.

represents a list of n integers, starting at a given heap location l.

$$\begin{aligned} &\mathit{list}(l, n) \mathrel{\hat{=}} \\ &\quad [(l = 0) \wedge (n = 0) \wedge \mathbf{emp}] \vee \\ &\quad [\neg(l = 0) \wedge ((l \mapsto -) \mathbin{*} (\exists l'.\, (l + 1 \mapsto l') \mathbin{*} \mathit{list}(l', n - 1)))] \end{aligned}$$

The use of the first $*$ in this formula ensures there can be no cycles in the list.

While the formal study of such recursive definitions dates back to Morris [76], Parkinson and Bierman [93] introduced the notion of *abstract* predicates into separation logic. An abstract predicate has a name, a definition, and a scope. The predicate's name has global visibility, while its definition is limited to the predicate's scope. That is, within its scope, a predicate can be replaced by its definition, but outside the scope the predicate can only be handled atomically, by its name. The advantages of this approach lie in the area of abstract data types and information hiding. Suppose that we have a module that provides a number of functions for performing operations on an abstract data type. The essence of an abstract data type, is that users of this module need not know how the data type is actually implemented. Abstract predicates apply the same principle to function specifications. That is, they allow the pre- and postcondition of a function to be considered a contract that specifies in abstract terms how that function behaves, without exposing the implementation details of that function. For instance, suppose we have a function for appending an integer to a list. The signature and contract of this function could be as follows.

$$\mathbf{routine}\ \mathit{append}(i, l) = \exists n.\, \mathbf{pre}\ \mathit{list}(l, n)\ \mathbf{post}\ \mathit{list}(l, n + 1)$$

Users of this function need not know the definition of the *list* predicate, since the (partial) behavior of *append* is specified in terms of the abstract predicate *list*.

Concurrency

The programming languages we have considered so far are all single-threaded. That is, each command of a program is processed in sequence and no two commands can ever be processed at the same time. Many practical programs, however, are *multi-threaded* in nature. Problems occur in these programs when multiple threads access the same resource at the same time. For instance, if two threads simultaneously write to the same heap cell, the result is undefined. Hence, one of the key programming challenges when writing concurrent programs, is to ensure mutually exclusive resource usage between threads. Since the core ingredient of separation logic is its ability to split up the heap into disjoint chunks, it is well-suited for reasoning about shared-variable concurrency, where

different portions of the heap are accessed by different threads at the same time. The ideas behind axiomatic methods for reasoning about concurrent programs date back to an early paper of Hoare [55] and were further developed by Owicki and Gries [92] and later O'Hearn [87].

Of course, concurrent programs of which the threads are completely disjoint are not very interesting; there is usually some shared state between the threads. For instance, in a producer-consumer style program, one thread (the producer) will calculate some result and subsequently place that result into a buffer, while the other thread (the consumer) in parallel retrieves those results from the same buffer and subsequently performs some calculations with them. This program will work fine as long as the producer does not add a value to the buffer at the same time that the consumer is retrieving one. Essentially, we want to dynamically transfer the *permission* to access the buffer between the two threads, and we can model this permission using the separation logic points-to assertion.

In order to perform such parallel executions with mutual exclusive access to resources, a number of new constructs must be added to our programming language. The concurrent Hoare and separation logic literature [92, 87] uses the following construct for parallel routines, based on Algol 60.

$$
\begin{array}{l}
\mathit{init};\\
\textbf{resource}\ r_1(\overline{x}_1), \ldots, r_m(\overline{x}_m)\\
C_1 \ ||\ \ldots\ ||\ C_n
\end{array}
$$

The idea is that the *init* statement is a sequentially executed sequence of commands that initializes some resources r_i, before forking into the parallel execution of $C_1, \ldots, C_n$. Each resource r_i is a set of variables $\overline{x}_i$ that are shared between the parallel threads, in a controlled way. The command for accessing such variables is the *conditional critical region*.

$$\textbf{with}\ r\ \textbf{when}\ B\ \textbf{do}\ C$$

Here, r is a resource, B is a (heap-independent) boolean expression and C is a command that can use the variables of r. It models common programming idioms for controlling access to shared resources, such as binary semaphores, mutexes and monitors. When a thread tries to execute a critical region, it will wait until the condition B is true and the resource r is not being used by another thread. When the thread acquires r and B is true, it executes C and then releases r again when C has been executed.

The following syntactic restrictions ensure that shared variables accessed in parallel threads are protected by critical regions:

- each variable must belong to *at most one* resource;
- variables that belong to a resource r cannot appear in a parallel thread *unless* they are in a critical region for r; and
- variables that *do not* belong to a resource and are modified in some thread C_i cannot appear in any other thread C_j (where $i \neq j$).

Although these rules are sufficient to prevent concurrent variable access for while programs, they are not enough to avoid races in pointer programs, due to aliasing. For instance, the following parallel program adheres to the above rules, but contains a race condition.

$$x := \mathbf{alloc};\ y := x;\ ([x] := 5 \mid\mid [y] := 6)$$

We can use separation logic to reason about such concurrent programs, by extending it with the following proof rules. Of course, the idea is that programs with race conditions, such as the one above, cannot be proven to be valid. The first extension is the *complete program rule*, as shown below.

Complete program rule

$$\frac{\{P\}\ \mathit{init}\ \{RI_{r_1} * \ldots * RI_{r_m} * P'\} \qquad \{P'\}\ C_1 \mid\mid \ldots \mid\mid C_n\ \{Q\}}{\{P\}\ \mathit{init};\ \mathbf{resource}\ r_1(\overline{x}_1), \ldots, r_m(\overline{x}_m)\ C_1 \mid\mid \ldots \mid\mid C_n\ \{RI_{r_1} * \ldots * RI_{r_m} * Q\}}$$

This rule should be interpreted as follows. The *init* statement is expected to establish a disjoint *resource invariant* RI_{r_i} for each declared resource. Each free variable in a resource invariant RI_{r_i} must belong to resource r_i, and each resource invariant must also be *precise* (as defined in Section 4.3.1). After the initialization, the parallel composition $C_1 \mid\mid \ldots \mid\mid C_n$ will execute, but none of the parallel threads can assume the resource invariants just yet. It is only when a thread enters a critical region for a resource that it receives access to the invariant of that resource, as indicated by the next rule.

Critical region rule

$$\frac{\{(P * RI_r) \wedge B\}\ C\ \{Q * RI_r\}}{\{P\}\ \mathbf{with}\ r\ \mathbf{when}\ B\ \mathbf{do}\ C\ \{Q\}}\ \begin{array}{l}(\text{no } x \in (\mathsf{FV}(P) \cup \mathsf{FV}(Q)) \text{ is modified}\\ \text{by another thread})\end{array}$$

Finally, the *parallel composition rule* allows each thread of a parallel composition to obtain a *disjoint* piece of memory and to combine the postconditions of each thread at the end of the parallel execution.

Parallel composition rule

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \qquad \ldots \qquad \{P_n\}\ C_n\ \{Q_n\}}{\{P_1 * \ldots * P_n\}\ C_1 \mid\mid \ldots \mid\mid C_n\ \{Q_1 * \ldots * Q_n\}}\ \begin{array}{l}(\text{where } (\mathsf{FV}(P_i) \cup \mathsf{FV}(Q_i)) \cap\\ \mathsf{mod}(C_j) = \emptyset \text{ when } i \neq j)\end{array}$$

To see how these rules prevent racy programs, consider again the example program shown above. We can consider the first two commands to be the initialization, and, since there are no resource declarations, there are no resource invariants. Hence we will end up having to prove

$$\{(x \mapsto -, -) \wedge y = x\}\, [x] := 5 \;||\; [y] := 6\, \{Q\}$$

for some postcondition Q. But, the only proof rule that can be considered is the parallel composition rule and there is no way to split the precondition into disjoint parts P_1 and P_2 such that $\vdash \{P_1\}\,[x] := 5\,\{Q_1\}$ and $\vdash \{P_2\}\,[y] := 6\,\{Q_2\}$.

To illustrate how these proof rules allow proper resource sharing using critical regions, consider the following example, taken from [87].

```
full = 0;
resource r(c, full)
x := alloc;                  ||    with r when (full = 1) do
with r when (full = 0) do    ||      y := c;
  c := x;                    ||      full := 0;
  full := 1                  ||    dealloc(y)
```

Below is a full proof of this program, starting from the initial precondition **emp** and ending with the same postcondition. The resource invariant RI_r has been selected to be $(\mathit{full} = 1 \wedge c \mapsto -, -) \vee (\mathit{full} = 0 \wedge \mathbf{emp})$.

```
{emp}
full = 0;
resource r(c, full)
{emp ∗ emp}
{emp}                                ||   {emp}
x := alloc;                          ||   with r when (full = 1) do
{x ↦ −, −}                           ||     {(RI_r ∗ emp) ∧ full = 1}
with r when (full = 0) do            ||     {(c ↦ −, −) ∧ full = 1}
  {(RI_r ∗ (x ↦ −, −)) ∧ full = 0}   ||     y := c; full := 0;
  {(x ↦ −, −) ∧ full = 0}            ||     {(y ↦ −, −) ∧ full = 0}
  c := x; full := 1                  ||     {(emp ∧ full = 0) ∗ (y ↦ −, −)}
  {(c ↦ −, −) ∧ full = 1}            ||     {RI_r ∗ (y ↦ −, −)}
  {RI_r}                             ||   {y ↦ −, −}
  {RI_r ∗ emp}                       ||   dealloc(y)
{emp}                                ||   {emp}
{emp ∗ emp}
{emp}
```

In the rest of this chapter we will return our attention to sequential programs.

4.4 Symbolic execution

The previous two sections defined formalisms for proving properties of imperative programs with and without pointers. While these formalisms can be used immediately for sound manual reasoning about programs, there is still a gap that needs to be filled in order to use them for *automatic* program verification. In particular, we need a way to encode the separation logic proof rules into an algorithm that (semi-)automatically determines whether a program annotated with separation logic assertions upholds its specification under all possible inputs. This clearly cannot be done by simply executing the program and checking whether the concrete store and heap satisfy the specifications at all points, because in general the set of possible input values is infinite. Furthermore the non-deterministic nature of memory allocation (as defined in Figure 4.5) implies that there is an infinite set of possible executions for each user input.

Berdine et al. [20] were the first to publish an algorithm for the *symbolic execution* of imperative pointer programs based on separation logic. In symbolic execution, variables and heap cells are bound to *symbols*, which are placeholders for real program values. Flow-dependent constraints are placed on these symbols by executing successive program statements. The idea is that, since symbols can represent an infinite number of concrete values, a single symbolic execution of a program can represent all possible concrete executions of that program.

To illustrate the power of symbolic execution, consider the following simple example program.

$$x := \mathbf{alloc}; \mathbf{if}\ (0 < [x])\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2$$

The **alloc** command allocates two heap cells at some arbitrary location with some arbitrary initial value and lets x point to those cells. Hence, there are an infinite number of concrete executions of this program. We can however represent all of these executions by using symbols to represent the value of x and the value in the allocated heap cell. Initially these symbols are unconstrained and hence represent any possible integer value. Symbolically executing the if-statement will cause the execution to fork into two execution paths. The first path executes C_1, in which the symbol corresponding to the value of x will be constrained to values larger than 0. The other path executes C_2, in which the symbol is constrained to values smaller than or equal to 0. These constraints will influence the further symbolic execution. Suppose for instance that $C_1 = \mathbf{if}\ [x] < 5\ \mathbf{then\ dealloc(0)}$. The symbolic execution algorithm will deduce that there are some concrete executions where the (illegal) deallocation command will be executed and hence it must reject the program. However if C_1 would be, for instance, $\mathbf{if}\ [x] < 0\ \mathbf{then\ dealloc(0)}$, the symbolic execution

algorithm would deduce that the path of the deallocation command is infeasible and thus will not (yet) reject the program.

In the rest of this section, we will first define the programming and assertion languages we will work with. We will then describe the symbolic execution algorithm published by Jacobs et al. [60, 119], which is the core of the VeriFast program verifier [61]. This algorithm is based on the original symbolic execution algorithm published by Berdine et al. [20], which underlies the Smallfoot program verifier [19]. Both algorithms are designed to verify partial correctness, but extensions for total correctness do exist [31, 22].

4.4.1 Programming and assertion languages

The programming language we will be working with, is a variant of the pointer programming language, to which we have added support for *routines*, so that we can illustrate the modularity of the approach. This programming language is formally defined in Figure 4.9. As can be seen from this figure, the state now consists of a *stack* and a heap, i.e., the store has been replaced by a stack of stores. The store at the top of the stack is the *current* store, i.e., the store accessed by the currently executing routine. Separation logic triples for this language make statements only about the current store. Routine definitions have user-specified preconditions (**req**uires) and postconditions (**ens**ures), and while-loops are fitted with a user-specified **inv**ariant, corresponding to the assertion P of Hoare rule H3 in Figure 4.3.

The assertion language we will use is a restricted version of the separation logic assertion language, extended with user-defined abstract predicates. The language is formally defined in Figure 4.10. As this figure shows, the satisfaction relation $(s, h) \vDash P \rightsquigarrow s'$ now contains three components: the satisfying state (s, h), the satisfied assertion P and an updated store s'. This updated store was added to support the pattern variables $?x$ that appear in points-to and user-defined predicate assertions. That is, the updated store s' is an extended version of the original satisfying store s, that contains a binding for each pattern variable $?x$ in the satisfied assertion P. Pattern variables provide an alternative to auxiliary variables, by enabling us to introduce new existentially bound variables within assertions themselves. Their scope is the rest of the assertion they reside in.

The restriction on assertions in this section compared to the full separation logic assertion language, is that assertions are now limited to being $*$-separated boolean expressions, conditionals, points-to predicates or user-defined predicates. Hence, the language does not support non-separating conjunction, disjunction, negation or separating implication. Although these restrictions may appear

$$C ::= \textbf{skip} \mid x := E \mid \textbf{if } B \textbf{ then } C \textbf{ else } C \mid \textbf{while } B \textbf{ inv } P \textbf{ do } C \mid x := r(\overline{E}) \mid$$
$$x := \textbf{alloc} \mid x := [E] \mid [E] := E \mid \textbf{dealloc}(E) \mid C; C$$

$$\mathit{rdef} ::= \textbf{routine } r(\overline{x}) \ = \textbf{req } P \textbf{ ens } P \textbf{ do } C \qquad\qquad r \in \mathit{Routines}$$

$$\frac{[\![B]\!]_s = \texttt{true}}{\langle (s{::}\overline{tl}, h), \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2\rangle \longrightarrow \langle (s{::}\overline{tl}, h), C_1\rangle}\ \textsc{If-T} \qquad \frac{[\![B]\!]_s = \texttt{false}}{\langle (s{::}\overline{tl}, h), \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2\rangle \longrightarrow \langle (s{::}\overline{tl}, h), C_2\rangle}\ \textsc{If-F}$$

$$\frac{[\![B]\!]_s = \texttt{true}}{\langle (s{::}\overline{tl}, h), \textbf{while } B \textbf{ inv } P \textbf{ do } C\rangle \longrightarrow \langle (s{::}\overline{tl}, h), C; \textbf{while } B \textbf{ inv } P \textbf{ do } C\rangle}\ \textsc{While-T} \qquad \frac{[\![B]\!]_s = \texttt{false}}{\langle (s{::}\overline{tl}, h), \textbf{while } B \textbf{ inv } P \textbf{ do } C\rangle \longrightarrow \langle (s{::}\overline{tl}, h), \textbf{skip}\rangle}\ \textsc{While-F}$$

$$\frac{\langle (\overline{s}, h), C_1\rangle \longrightarrow \langle (\overline{s}', h'), C_1'\rangle}{\langle (\overline{s}, h), C_1; C_2\rangle \longrightarrow \langle (\overline{s}', h'), C_1'; C_2\rangle}\ \textsc{Seq} \qquad \frac{}{\langle (\overline{s}, h), \textbf{skip}; C\rangle \longrightarrow \langle (\overline{s}, h), C\rangle}\ \textsc{Seq-skip}$$

$$\frac{\langle (\overline{s}, h), C_1\rangle \longrightarrow \textbf{fail}}{\langle (\overline{s}, h), C_1; C_2\rangle \longrightarrow \textbf{fail}}\ \textsc{Seq-fail} \qquad \frac{[\![E]\!]_s = n}{\langle (s{::}\overline{tl}, h), x := E\rangle \longrightarrow \langle (s[x \to n]{::}\overline{tl}, h), \textbf{skip}\rangle}\ \textsc{Assign}$$

$$\frac{[\![E]\!]_s \in \operatorname{dom} h}{\langle (s{::}\overline{tl}, h), x := [E]\rangle \longrightarrow \langle (s[x \to h([\![E]\!]_s)]{::}\overline{tl}, h), \textbf{skip}\rangle}\ \textsc{Lookup-ok} \qquad \frac{[\![E]\!]_s \notin \operatorname{dom} h}{\langle (s{::}\overline{tl}, h), x := [E]\rangle \longrightarrow \textbf{fail}}\ \textsc{Lookup-fail}$$

$$\frac{[\![E]\!]_s \in \operatorname{dom} h}{\langle (s{::}\overline{tl}, h), [E] := E'\rangle \longrightarrow \langle (s{::}\overline{tl}, h[[\![E]\!]_s \to [\![E']\!]_s]), \textbf{skip}\rangle}\ \textsc{Mutate-ok} \qquad \frac{[\![E]\!]_s \notin \operatorname{dom} h}{\langle (s{::}\overline{tl}, h), [E] := E'\rangle \longrightarrow \textbf{fail}}\ \textsc{Mutate-fail}$$

$$\frac{\textbf{routine } r(\overline{y}) \ = \textbf{req } P \textbf{ ens } Q \textbf{ do } C}{\langle (s{::}\overline{tl}, h), x := r(\overline{E})\rangle \longrightarrow \langle (\{\overline{y} \to [\![\overline{E}]\!]_s\}{::}s{::}\overline{tl}, h), C; \textbf{ret}_x\rangle}\ \textsc{Call} \qquad \frac{}{\langle (s{::}s'{::}\overline{tl}, h), \textbf{ret}_x\rangle \longrightarrow \langle (s'[x \to [\![\texttt{result}]\!]_s]{::}\overline{tl}, h), \textbf{skip}\rangle}\ \textsc{Return}$$

$$\frac{\ell \in \mathbb{N}_0^+ \qquad \{\ell, \ell+1\} \cap \operatorname{dom} h = \emptyset \qquad v_1, v_2 \in \mathbb{N}}{\langle (s{::}\overline{tl}, h), x := \textbf{alloc}\rangle \longrightarrow \langle (s[x \to \ell]{::}\overline{tl}, h[\ell \to v_1, \ell+1 \to v_2]), \textbf{skip}\rangle}\ \textsc{Alloc}$$

$$\frac{[\![E]\!]_s \in \operatorname{dom} h \qquad [\![E]\!]_s \mod 2 = 0}{\langle (s{::}\overline{tl}, h), \textbf{dealloc}(E)\rangle \longrightarrow \langle (s{::}\overline{tl}, h \setminus \{[\![E]\!]_s, [\![E]\!]_s + 1\}), \textbf{skip}\rangle}\ \textsc{Dealloc-ok}$$

$$\frac{([\![E]\!]_s \notin \operatorname{dom} h) \vee ([\![E]\!]_s \mod 2 \neq 0)}{\langle (s{::}\overline{tl}, h), \textbf{dealloc}(E)\rangle \longrightarrow \textbf{fail}}\ \textsc{Dealloc-fail}$$

Figure 4.9: Syntax and small-step operational semantics for the symbolic execution programming language, which is an extension of the pointer programming language defined in Figure 4.5.

$$pdef ::= \textbf{pred}\ p(x, y) = P \qquad\qquad p \in Preds$$

$$P ::= B \mid B\,?\,P\ :\ P \mid E \mapsto ?x \mid p(E, ?x) \mid P * P$$

$(s,h) \vDash B \rightsquigarrow s$	iff $[\![B]\!]_s = \texttt{true}$
$(s,h) \vDash B\,?\,P\ :\ P' \rightsquigarrow s$	iff if $(s,h) \vDash B$ then $(s,h) \vDash P$ else $(s,h) \vDash P'$
$(s,h) \vDash E \mapsto ?x \rightsquigarrow s'$	iff $\mathrm{dom}\, h = [\![E]\!]_s$ and $s' = s[x \to h([\![E]\!]_s)]$
$(s,h) \vDash p(E, ?z) \rightsquigarrow s'$	iff $\textbf{pred}\ p(x, y) = P$ and $\exists v.\, (\{x \to [\![E]\!]_s, y \to v\}, h) \vDash P$ and $s' = s[z \to v]$
$(s,h) \vDash P * P' \rightsquigarrow s''$	iff $\exists h_0, h_1.\, h_0 \perp h_1$ and $h_0 \cup h_1 = h$ and $(s, h_0) \vDash P \rightsquigarrow s'$ and $(s', h_1) \vDash P' \rightsquigarrow s''$
$(s,h) \vDash P$	iff $\exists s'.\, (s,h) \vDash P \rightsquigarrow s'$

Figure 4.10: Symbolic execution assertion language syntax and semantics.

to limit the expressivity of the assertion language, real-world experience with verification tools that use this language [19, 61, 100, 38, 25] has proven that it suffices to express all desired verification properties in practice.

Assertions from this assertion language can be divided into two categories: *spatial* assertions and *pure* assertions. Spatial assertions describe the structure of the heap, while pure assertions only place constraints on logical symbols without making any claims about the heap. Points-to assertions and user-defined predicate assertions are spatial, while boolean expression assertions are pure.

4.4.2 Symbolic execution algorithm

The symbolic execution algorithm will keep track of a *symbolic state* $\gamma = (\Delta, \Pi, \Sigma)$, consisting of a symbolic store Δ, a path condition Π, and a symbolic heap Σ. We will explain the role of each of these components, based on the symbolic execution of the following example routine.

```
routine r(x,y) =
  req (x > -5) ✳ (x < 5) ✳ p(x,y)
  ens true
do
  z = alloc;
  if x < 0 then
    result = -1
  else
    result = 1
```

The symbolic store is a mapping from local variable names to symbolic expressions that represent the variables' values. For instance, at the highlighted execution point in the above routine, the symbolic store is simply the trivial mapping $\{x := \dot{x}, y := \dot{y}, z := \dot{z}\}$, where the non-dotted names represent variables and the dotted names are symbols.

The path condition is a first-order logic formula that constrains the values of the logical symbols that appear in the symbolic store or heap, based on the routine's precondition and the path taken through the routine's code. For instance, at the highlighted execution point in the example routine, the path condition will be $(\dot{x} > -5 \wedge \dot{x} < 5) \wedge (\dot{x} < 0)$, where the outer left conjunct comes from the precondition and the right conjunct comes from the if-statement.

The symbolic heap is a multiset of *heap chunks*, where each heap chunk is of the form $p(E_1, E_2)$ with p the name of a predicate (i.e., either a user-defined name or $\mapsto$) and E_1 and E_2 are the arguments of the predicate. Each chunk on the symbolic heap potentially represents a piece of heap storage that is exclusively accessible by the current routine and is disjoint from the storage represented by all other chunks on the symbolic heap. At the highlighted execution point in the example routine, the symbolic heap will consist of a user-defined predicate chunk $p(\dot{x}, \dot{y})$ originating from the precondition and two points-to predicate chunks $\dot{z} \mapsto \dot{v}$ and $\dot{z} + 1 \mapsto \dot{v}'$, originating from the $z :=$ **alloc** command.

Transition relations

As in [60], we will define the semantics of the symbolic execution algorithm by means of transitions $\gamma \rightsquigarrow o$ from an initial symbolic state γ to an *outcome* o. An outcome is either a symbolic state, or the special failure outcome **fail**.

At some points during the execution of a program, there will be a choice of the path to follow next. For instance, when executing an if-then-else command, the concrete execution will choose between the then-path and the else-path, based on the value of the condition. Since the symbolic execution represents

all possible concrete executions, it must consider *both* paths and both must succeed for the overall symbolic execution to succeed (unless a path is deemed infeasible, in which case it immediately succeeds). Following [119], we call such choices *demonic*. Thus, when encountering an if-then-else command, the symbolic execution must fork, which has the effect that a single initial state can now lead to multiple different outcomes. For instance, the if-then-else statement in the example program leads to the following two result transitions:

- $(\Delta_0, \emptyset, \emptyset) \rightsquigarrow (\Delta, (\dot{x} > -5 \wedge \dot{x} < 5 \wedge \dot{x} < 0 \wedge \dot{r} = -1)), \{\dot{z} \mapsto -\})$, and
- $(\Delta_0, \emptyset, \emptyset) \rightsquigarrow (\Delta, (\dot{x} > -5 \wedge \dot{x} < 5 \wedge \dot{x} > 0 \wedge \dot{r} = 1)), \{\dot{z} \mapsto -\})$,

where Δ_0 is the initial symbolic store $\{x := \dot{x}, y := \dot{y}\}$ and Δ is the final symbolic store $\Delta_0 \cup \{z := \dot{z}, \mathit{result} := \dot{r}\}$.

Hence, as a first approximation, the result of the symbolic execution of a routine starting from some initial state γ can be modeled as a transition relation $R = \{\gamma \rightsquigarrow o', \gamma \rightsquigarrow o'', \dots\}$, consisting of pairs of related symbolic states and outcomes. The overall symbolic execution is successful if none of the transitions in R starting from the initial state ends with the failure outcome.

However, as will be made explicit in the next section, there are also points in the symbolic execution of a routine where there is a choice of how to proceed, such that the overall execution is successful if *at least one* of the choices leads to success. These are so-called *angelic* choices [119]. To model this, the result of the symbolic verification of a routine from an initial state γ will actually be a *set of* transition relations. That is, the result will have the form $W = \{R_1, \dots, R_n\}$ where each R_i is a transition relation $\{\gamma \rightsquigarrow \gamma', \gamma \rightsquigarrow \gamma'', \dots\}$. Each R_i corresponds to a different combination of angelic choices made throughout the algorithm and each pair $\gamma \rightsquigarrow \gamma'$ *within* an R_i corresponds to a different execution path (i.e., a different set of demonic choices), given those angelic choices. Verification is successful if there is *at least one* R_i where *none* of the transition relations starting from the initial state end with the failure outcome.

Assumption, production and consumption

The symbolic execution algorithm is based upon three core operations: *assumption*, *production* and *consumption* [59]. In order to formally define these operations, we first define the conjunction and sequential composition of transition relations.

$$W \wedge W' = \{R \cup R' \mid R \in W \wedge R' \in W'\}$$

$$(\bigwedge i \in I.\, W(i)) = \{\bigcup i \in I.\, \psi(i) \mid \forall i \in I.\, \psi(i) \in W(i)\}$$

The conjunction $W \wedge W'$ denotes the pairwise union of relations from W and W'. It can be considered a demonic operation, since at least one relation in W *and* at least one in W' must succeed for $W \wedge W'$ to succeed.

$$W; W' = \{R; R' \mid R \in W \wedge R' \in W'\}$$

The sequential composition $W; W'$ of sets of transition relations denotes the pairwise sequential composition of relations of W and W', where the sequential composition of transition relations is defined as follows.

$$R; R' = \{\gamma \rightsquigarrow \mathbf{fail} \mid \gamma \rightsquigarrow \mathbf{fail} \in R\} \cup \{\gamma \rightsquigarrow o \mid \gamma \rightsquigarrow \gamma' \in R \wedge \gamma \rightsquigarrow o \in R'\}$$

We can now define the three core operations of the verification algorithm. The assumption operation takes as input a boolean expression and returns a singleton set of transition relations in which this expression has been added to the path condition, unless this would lead to a contradiction in the path condition.

$$\mathbf{assume}(B) = \ \{\{(\Delta, \Pi, \Sigma) \rightsquigarrow (\Delta, \Pi \cup \{[\![B]\!]_\Delta\}, \Sigma) \mid \Pi \not\vdash_{\mathrm{SMT}} \neg[\![B]\!]_\Delta\}\}$$

As suggested by the above notation, detecting a contradicting path condition can be done using an SMT solver [36], or alternatively it can be done using a separate proof engine that is part of the verifier.

The production and consumption operations both take as input an assertion and return a set of transition relations. Informally, production corresponds to adding chunks to the symbolic heap and adding constraints to the path condition, while consumption corresponds to removing chunks from the symbolic heap and checking that a given symbolic expression follows from the path condition. We will now formally define these operations for each possible kind of assertion.

$$\mathbf{produce}(p(E, E')) = \{\{(\Delta, \Pi, \Sigma) \rightsquigarrow (\Delta, \Pi, \Sigma \uplus \{p([\![E]\!]_\Delta, [\![E']\!]_\Delta)\})\}\}$$

$$\mathbf{produce}(p(E, ?x)) = \{\{(\Delta, \Pi, \Sigma) \rightsquigarrow (\Delta[x := \dot{v}], \Pi', \Sigma \uplus \{p([\![E]\!]_\Delta, \dot{v})\}) \mid (\dot{v}, \Pi') = \mathbf{nextFresh}(\Pi)\}\}$$

$$\mathbf{produce}(E \mapsto E)) = \mathbf{produce}(\mapsto (E, E))$$

$$\mathbf{produce}(E \mapsto ?x)) = \mathbf{produce}(\mapsto (E, ?x))$$

Producing a *spatial* assertion (i.e., a points-to or user-defined predicate assertion) means adding a corresponding chunk to the symbolic heap, thereby potentially

binding a new symbol to a variable if a pattern variable is specified. The function

$$\mathbf{nextFresh}(\Pi) = (\dot{v}, \Pi \cup \{\dot{v} = \dot{v}\})$$

returns a fresh symbol $\dot{v}$ that does not appear in Π, and an updated path condition in which the new symbol does appear. We sometimes use the same function to allocate multiple fresh variables at once. Apart from the infix notation, producing a points-to predicate is identical to producing a user-defined predicate. We use the wildcard notation $E \mapsto -$ as syntactic sugar for $E \mapsto ?x$ when we do not want to specify an expression for the value at location E, and we also have no need to bind a symbol to the value.

$$\mathbf{choice}(W) = \{\{\psi(R) \mid R \in W\} \mid \forall R \in W.\, \psi(R) \in R\}$$

$$\begin{aligned}
&\mathbf{consume}(p(E, E')) = \\
&\quad \text{let } \mathbf{matches}(\Delta, \Pi, \Sigma) = \\
&\qquad \{(\Delta, \Pi, \Sigma) \rightsquigarrow (\Delta, \Pi, \Sigma') \mid \Sigma = \Sigma' \uplus \{p(\dot{v}, \dot{v}')\} \wedge \Pi \vdash_{\text{SMT}} [\![(E, E')]\!]_\Delta = (\dot{v}, \dot{v}')\} \text{ in} \\
&\qquad \mathbf{choice}(\{\mathbf{matches}(\Delta, \Pi, \Sigma) \mid \mathbf{matches}(\Delta, \Pi, \Sigma) \neq \emptyset\}) \\
&\qquad \wedge \{\{(\Delta, \Pi, \Sigma) \rightsquigarrow \mathbf{fail} \mid \mathbf{matches}(\Delta, \Pi, \Sigma) = \emptyset\}\}
\end{aligned}$$

$$\begin{aligned}
&\mathbf{consume}(p(E, ?x)) = \\
&\quad \text{let } \mathbf{matches}(\Delta, \Pi, \Sigma) = \\
&\qquad \{(\Delta, \Pi, \Sigma) \rightsquigarrow (\Delta[x := \dot{v}'], \Pi, \Sigma') \mid \Sigma = \Sigma' \uplus \{p(\dot{v}, \dot{v}')\} \wedge \Pi \vdash_{\text{SMT}} [\![E]\!]_\Delta = \dot{v}\} \text{ in} \\
&\qquad \mathbf{choice}(\{\mathbf{matches}(\Delta, \Pi, \Sigma) \mid \mathbf{matches}(\Delta, \Pi, \Sigma) \neq \emptyset\}) \\
&\qquad \wedge \{\{(\Delta, \Pi, \Sigma) \rightsquigarrow \mathbf{fail} \mid \mathbf{matches}(\Delta, \Pi, \Sigma) = \emptyset\}\}
\end{aligned}$$

$$\mathbf{consume}(E \mapsto E) = \mathbf{consume}(\mapsto (E, E))$$

$$\mathbf{consume}(E \mapsto ?x) = \mathbf{consume}(\mapsto (E, ?x))$$

Consuming a spatial assertion means removing a matching chunk from the symbolic heap, again thereby potentially binding a symbolic value to a variable if a pattern variable is specified. Consumption fails if no matching chunk is found, since this means the program could be trying to access a memory location that it is not allowed to access (e.g. because the location is unallocated). However, since the symbolic heap is a *multi*set, it is also possible that there are *multiple* matching chunks when consuming a predicate assertion. Hence there could be a choice of chunks for the algorithm to consume in order to proceed. This choice is of the *angelic* kind, since the overall execution will be successful if *at least one* of the choices leads to success. In practice, we can either (1) abort the algorithm when an ambiguous match must be made, in order to force the user to structure the program such that ambiguous matches cannot occur, or (2) simply pick an arbitrary matching chunk to proceed with. In the latter case, if

the wrong chunk is chosen, the algorithm will eventually fail, at which point it can backtrack and choose another chunk.

$$\mathbf{produce}(B) = \mathbf{assume}(B)$$

$$\mathbf{consume}(B) = \\ \{\{(\Delta, \Pi, \Sigma) \rightsquigarrow (\Delta, \Pi, \Sigma) \mid \Pi \vdash_{\text{SMT}} [\![B]\!]_\Delta\} \cup \{(\Delta, \Pi, \Sigma) \rightsquigarrow \mathbf{fail} \mid \Pi \nvdash_{\text{SMT}} [\![B]\!]_\Delta\}\}$$

Producing a *pure* assertion (i.e., a boolean expression), means adding it to the path condition. Consuming a pure assertion means asserting that it follows from the current path condition.

$$\mathbf{produce}(B \,?\, P \,:\, P') = \\ \quad \mathbf{assume}(B); \mathbf{produce}(P) \wedge \mathbf{assume}(\neg B); \mathbf{produce}(P')$$

$$\mathbf{consume}(B \,?\, P \,:\, P') = \\ \quad \mathbf{assume}(B); \mathbf{consume}(P) \wedge \mathbf{assume}(\neg B); \mathbf{consume}(P')$$

Producing or consuming a conditional assertion $B \,?\, P \,:\, P'$ causes the verification algorithm to fork in a demonic way. On one execution path it will assume B (i.e., add it to the path condition) and subsequently produce, respectively consume P. On the other execution path it will assume $\neg B$ and subsequently produce, respectively consume P'. The algorithm fails if either of these two execution paths fail.

$$\mathbf{produce}(P * P') = \mathbf{produce}(P); \mathbf{produce}(P')$$

$$\mathbf{consume}(P * P') = \mathbf{consume}(P); \mathbf{consume}(P')$$

Finally, producing or consuming a separating conjunction $P * P'$ simply means first producing, respectively consuming P and then subsequently producing, respectively consuming P'.

Routine verification

We can now use the production and consumption operations given above, to define what it means to verify a routine. The symbolic execution of a routine will result in a set of transition relations, and the routine will be deemed *valid* if there is at least one transition relation in this set where the initial state does not lead to failure.

$$
\begin{aligned}
&\textbf{valid}(\textbf{routine}\ r(\overline{x})\ =\ \textbf{req}\ P\ \textbf{ens}\ Q\ \textbf{do}\ C) = \\
&\quad \exists R \in W.\, ((\{\overline{x} := \dot{\overline{v}}\}, \Pi_0, \emptyset) \rightsquigarrow \textbf{fail}) \notin R \\
&\quad \text{where } (\dot{\overline{v}}, \Pi_0) = \textbf{nextFresh}(\emptyset) \\
&\quad \text{and } W = \bigwedge \Delta.\, \{\{(\Delta, \Pi, \Sigma) \rightsquigarrow (\Delta, \Pi, \Sigma)\}\}; \textbf{produce}(P); \\
&\qquad \bigwedge \Delta'.\, \{\{(\Delta', \Pi, \Sigma) \rightsquigarrow (\Delta, \Pi, \Sigma)\}\}; \textbf{verify}(C); \\
&\qquad \bigwedge \Delta''.\, \{\{(\Delta'', \Pi, \Sigma) \rightsquigarrow (\Delta'[\text{result} := \Delta''(\text{result})], \Pi, \Sigma)\}\}; \textbf{consume}(Q); \\
&\qquad \textbf{leakCheck}
\end{aligned}
$$

The symbolic execution starts from the initial state, where the routine's parameters are bound to fresh symbols, and the path condition and symbolic heap are empty. We first "save" the initial symbolic store Δ, using the $\bigwedge$ operator as a quantifier over all possible symbolic stores. Next, the precondition P is produced, the resulting symbolic store Δ' is saved, and the original symbolic store Δ is restored. The body of the routine is then verified using the **verify** operation, which is defined below. The post-production symbolic store Δ' is then restored, with the addition of a *result* variable that is copied over from the routine body verification step. This store is then used to consume the postcondition Q. Finally a so-called *leak check* is performed, leading all states with a non-empty symbolic heap to failure.

$$\textbf{leakCheck} = \{\{(\Delta, \Pi, \emptyset) \rightsquigarrow (\Delta, \Pi, \emptyset)\}\} \wedge \{\{(\Delta, \Pi, \Sigma) \rightsquigarrow \textbf{fail} \mid \Sigma \neq \emptyset\}\}$$

If the symbolic heap still contains heap chunks at the end of the routine validation process, there could potentially be a memory leak in the verified routine. This is because the remaining chunks can be points-to chunks or abstract predicate chunks that represent the exclusive permission for the routine to access a piece of memory. If these chunks are not removed by consuming the routine's postcondition, then the permission to access the corresponding memory locations is not transferred to the routine's caller when the routine returns (see the definition of $\textbf{verify}(x := r(\overline{E}))$ below). Hence the permissions are lost and routine will no longer be able to access the corresponding memory locations.

We will now describe how to verify each of the commands that make up the body of a routine. We start with three simple structural commands.

$$\textbf{verify}(\textbf{skip}) = \{\{(\Delta, \Pi, \Sigma) \rightsquigarrow (\Delta, \Pi, \Sigma)\}\}$$

$$\textbf{verify}(\textbf{if}\ B\ \textbf{then}\ C_1\ \textbf{else}\ C_2) = \textbf{assume}(B); \textbf{verify}(C_1) \wedge \textbf{assume}(\neg B); \textbf{verify}(C_2)$$

$$\textbf{verify}(C; C') = \textbf{verify}(C); \textbf{verify}(C')$$

Verifying a **skip** command does not change the symbolic state at all. Verifying an if-then-else command is similar to the production or consumption of a conditional assertion: the symbolic execution forks into two execution paths that must both succeed for the overall execution to succeed. The first execution path assumes B and executes the then-case, while the other execution path assumes $\neg B$ and executes the else-case. Verifying a command sequence $C; C'$ simply means first verifying C and subsequently verifying C'.

$$\mathbf{verify}(x := E) = \{\{(\Delta, \Pi, \Sigma) \rightsquigarrow (\Delta[x := [\![E]\!]_\Delta], \Pi, \Sigma)\}\}$$

Verifying a variable assignment means updating the symbolic store to reflect the variable's new value.

$$\begin{array}{l}\mathbf{verify}(x := \mathbf{alloc}) = \\ \quad \mathbf{produce}(\mathrm{mb}(x, 2) * (x \mapsto -) * (x + 1 \mapsto -))\end{array}$$

$$\begin{array}{l}\mathbf{verify}(\mathbf{dealloc}(E)) = \\ \quad \mathbf{consume}(\mathrm{mb}(E, 2) * (E \mapsto -) * (E + 1 \mapsto -))\end{array}$$

Verifying a memory allocation of a pair of heap cells means producing a *memory block* chunk $\mathrm{mb}(x, 2)$ and two points-to chunks, where x is the name of the variable that will point to the allocated memory region. Verifying a memory deallocation performs the opposite operation: consume the memory block chunk $\mathrm{mb}(E, 2)$ and the two corresponding points-to chunks. The $\mathrm{mb}(x, 2)$ chunk serves to remember the fact that x points to the first of the pair of allocated memory cells, so that an attempt to deallocate $x + 1$ would not verify.

$$\begin{array}{l}\mathbf{verify}(x := [E]) = \\ \quad \text{let } \mathbf{matches}(\Delta, \Pi, \Sigma) = \\ \qquad \{(\Delta, \Pi, \Sigma) \rightsquigarrow (\Delta[x := \dot{v}'], \Pi, \Sigma) \mid\mapsto (\dot{v}, \dot{v}') \in \Sigma \wedge \Pi \vdash_{\mathrm{SMT}} [\![(x, E)]\!]_\Delta = (\dot{v}, \dot{v}')\} \text{ in} \\ \qquad \mathbf{choice}(\{\mathbf{matches}(\Delta, \Pi, \Sigma) \mid \mathbf{matches}(\Delta, \Pi, \Sigma) \neq \emptyset\}) \\ \qquad \wedge \{\{(\Delta, \Pi, \Sigma) \rightsquigarrow \mathbf{fail} \mid \mathbf{matches}(\Delta, \Pi, \Sigma) = \emptyset\}\}\end{array}$$

Verifying a memory lookup means asserting that there is a points-to chunk for the corresponding memory location and assigning the symbolic value v' of that chunk to the assigned variable x in the symbolic store. Although in practice there will be at most one matching points-to chunk in the symbolic heap, pathological situations can occur in which there are multiple matching chunks. In that case the symbolic execution proceeds as it did when consuming a predicate assertion. That is, a different transition relation is constructed for each possible match and the overall verification succeeds if there is at least one relation that does not lead to failure from the initial state.

$$\mathbf{verify}([E] := E') = \mathbf{consume}(E \mapsto -); \mathbf{produce}(E \mapsto E')$$

Verifying a memory mutation simply means consuming a points-to predicate of the memory location being modified and subsequently producing a points-to chunk of the same memory location pointing to the updated value.

$$
\begin{aligned}
&\mathbf{verify}(\mathbf{while}\ B\ \mathbf{inv}\ P\ \mathbf{do}\ C) = \\
&\quad \mathbf{consume}(P); \\
&\quad \bigwedge \Sigma.\, \{\{(\Delta, \Pi, \Sigma) \rightsquigarrow (\Delta[\mathit{mod}(C) := \dot{\overline{v}}], \Pi', \emptyset) \mid (\dot{\overline{v}}, \Pi') = \mathbf{nextFresh}(\Pi)\}\}; \\
&\quad \mathbf{produce}(P); \\
&\quad (\mathbf{assume}(B); \mathbf{verify}(C); \mathbf{consume}(P); \mathbf{leakCheck}; \mathbf{assume}(\mathit{false}) \wedge \\
&\qquad \mathbf{assume}(\neg B); \{\{(\Delta, \Pi, \Sigma') \rightsquigarrow (\Delta, \Pi, \Sigma)\}\}) \\
&\text{where } \mathit{mod}(C) \text{ is the set of local variables modified by } C
\end{aligned}
$$

Verifying a while loop consists of first consuming the loop invariant P. Then, all remaining chunks are removed from the symbolic heap, but the original heap Σ is remembered. Notice that this means that only the heap chunks mentioned in the loop invariant can be accessed by the loop body. The same step also assigns a fresh symbol to each variable that is modified in the body of the loop. Next, the loop invariant P is produced and the symbolic execution forks into two execution paths. The first execution path assumes the loop condition B is true and then verifies the loop body C, followed by the consumption of the loop invariant P and a check that the symbolic heap is now empty. At this point, the execution path ends, which is modeled by the assumption of *false*. The other execution path assumes the loop condition B is false and restores the symbolic heap Σ for the execution of commands following the while loop. Since all variables that are modified by the loop body have been assigned a fresh symbol, the conjunction of $\neg B$ and the loop invariant should express any conditions on those variables that are to be assumed after the loop has been executed.

$$
\begin{aligned}
&\mathbf{verify}(x := r(\overline{E})) = \\
&\quad \bigwedge \Delta.\, \{\{(\Delta, \Pi, \Sigma) \rightsquigarrow (\{\overline{x} := [\![\overline{E}]\!]_\Delta\}, \Pi, \Sigma)\}\}; \mathbf{consume}(P); \\
&\quad \bigwedge \dot{r}.\, \{\{(\Delta', \Pi, \Sigma) \rightsquigarrow (\Delta'[\mathit{result} := \dot{r}], \Pi', \Sigma) \mid (\dot{r}, \Pi') = \mathbf{nextFresh}(\Pi)\}\}; \\
&\qquad \mathbf{produce}(Q); \{\{(\Delta'', \Pi, \Sigma) \rightsquigarrow (\Delta[x := \dot{r}], \Pi, \Sigma)\}\} \\
&\text{where } \mathbf{routine}\ r(\overline{x}) \ = \ \mathbf{req}\ P\ \mathbf{ens}\ Q\ \mathbf{do}\ C
\end{aligned}
$$

Finally, verifying a routine call consists of first creating a new symbolic store in which the routine's parameters have been bound to call's arguments. The callee's precondition P is then consumed under this new symbolic store, but the original store Δ is remembered. Next, a fresh symbol $\dot{r}$ is bound to the *result* variable and the postcondition Q is produced. Finally, the original store Δ is restored, with variable x bound to the result symbol $\dot{r}$.

Ghost statements

The algorithm presented up until this point provides a symbolic execution step for each of the basic constructs of the pointer programming language extended with user-defined predicates. There is however no way for users to actually use the predicates they define. Suppose, for instance, that the user has defined the following predicate, representing a linked list of length n.

$$\begin{array}{l}\mathrm{list}(x, ?n) \mathrel{\hat{=}} \\ \quad (x = 0)\,? \quad (n = 0) \quad : \\ \qquad ((x \mapsto -) * (x + 1 \mapsto ?next) * \mathrm{list}(next, ?n0) * (n = n0 + 1))\end{array}$$

Now suppose the symbolic heap, at some point during the symbolic execution, contains the following chunks: $(\dot{x} \mapsto 5) * (\dot{x} + 1 \mapsto 0)$. This corresponds to a linked list of length one, but there is no way to transform these chunks into a list chunk. Hence, at this point calling a routine that requires such a chunk would fail.

One way to solve this problem is to modify the verification algorithm to automatically search for chunks that can be transformed into the required precondition on routine calls. While such automatic transformations can be convenient for the user, it is a complex search operation that can potentially significantly increase the overall verification time if many transformations need to be considered.

Another approach is to leave the transformation of heap chunks to the user. This is the approach taken by VeriFast, which offers two so-called *ghost commands* to open and close predicates. A ghost command is a command that is executed as part of the symbolic execution process, but that does nothing during concrete execution (i.e., it there behaves as the **skip** command). Given that **pred** $p(x, y) = P$, the open and close ghost commands behave as follows.

$$\begin{array}{l}\mathbf{verify}(\mathbf{open}\ p(E, E')) = \\ \quad \mathbf{consume}(p(E, E')); \bigwedge \Delta.\, \{\{(\Delta, \Pi, \Sigma) \rightsquigarrow (\{x := [\![E]\!]_\Delta, y := [\![E']\!]_\Delta\}, \Pi, \Sigma)\}\}; \\ \quad \mathbf{produce}(P); \{\{(\Delta', \Pi, \Sigma) \rightsquigarrow (\Delta, \Pi, \Sigma)\}\} \\ \\ \mathbf{verify}(\mathbf{close}\ p(E, E')) = \\ \quad \bigwedge \Delta.\, \{\{(\Delta, \Pi, \Sigma) \rightsquigarrow (\{x := [\![E]\!]_\Delta, y := [\![E']\!]_\Delta\}, \Pi, \Sigma)\}\}; \\ \quad \mathbf{consume}(P); \{\{(\Delta', \Pi, \Sigma) \rightsquigarrow (\Delta, \Pi, \Sigma)\}\}; \mathbf{produce}(p(E, E'));\end{array}$$

Opening a predicate means consuming the corresponding predicate chunk from the symbolic heap, and subsequently producing the predicate's body

under a symbolic store where the predicate's parameters have been bound to its arguments. Closing a predicate is the reverse operation: consuming the predicate's body under a symbolic store where the predicate's parameters have been bound to its arguments, and subsequently producing the predicate assertion under the original store.

Another ghost command that can be convenient for users is the **assert** command. Many programming languages provide a runtime assert command that checks that some boolean expression evaluates to true and ends execution if it does not. However, we can also define a *static* assert command that is used during *symbolic* execution to check that some boolean expression follows from the current path condition. For boolean expressions, this ghost command simply corresponds to consuming the expression.

$$\textbf{verify}(\textbf{assert}\ B) = \textbf{consume}(B)$$

Like all ghost commands, the command has no effect during concrete execution.

Soundness

Proving the soundness of the symbolic execution algorithm is out of scope for this text, but for most commands it is easy to see the resemblance between the symbolic execution step and the corresponding Hoare or separation logic rule. A soundness proof sketch is given in [60] and a more elaborate proof is given in [119].

4.5 Summary

We started this chapter by proposing formal software verification as a solution for reducing the number of programming defects, for applications where the impact of such defects outweighs the extra development costs associated with the approach. We then zoomed in on two specific logics that provide an axiomatic approach to proving program correctness, namely Hoare logic and separation logic. Hoare logic provides a sound formal basis for reasoning about imperative programs, but is limited in its ability to reason about pointer programs, due to the effects of aliasing. Separation logic does away with this limitation, and can thus reason about languages with pointers to shared mutable data structures, such as C, C++ and Java. For this it relies on the principle of spatial separation, which allows us to express that certain sub-assertions hold for *disjoint* parts of the memory heap, such that local heap mutations will only have a local effect on a specific sub-assertion. Finally, we looked at the symbolic execution algorithm that underlies the VeriFast program verifier, which reveals how separation logic

proof rules can be used for semi-automatic program verification. The idea behind symbolic execution is that an infinite number of concrete executions can be represented by a single symbolic execution, by using variables that hold symbols rather than concrete values. Although the algorithm deals with a restricted version of the separation logic assertion language, the supported assertions still allow a natural way of expressing correctness properties for imperative programs.

Chapter 5

Sound Verification in an Unverified Context

Publication data

P. Agten, B. Jacobs, F. Piessens. *Sound modular verification of C code executing in an unverified context: extended version.* CW Reports CW676. Department of Computer Science, KU Leuven, Nov. 2014

P. Agten, B. Jacobs, F. Piessens. "Sound Modular Verification of C Code Executing in an Unverified Context". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL '15. Mumbai, India: ACM, 2015, pp. 581–594

Pieter Agten is the main contributor to these works, which were conducted under supervision of Frank Piessens and Bart Jacobs.

5.1 Introduction

The construction of reliable software in unsafe languages like C or C++ is known to be challenging. Yet, because of the excellent performance of these languages, and because they can give the programmer access to low-level details of the machine on which the program is executing, they are often the languages

of choice for infrastructural software such as hypervisors, operating systems and servers, and for embedded software.

As discussed in Chapter 4, one way of significantly increasing assurance in the reliability of software, is the use of program verification. For large systems, it is essential that the verification technique used is *modular*. Each module (i.e., for instance, each function) is verified to comply with its specification, relying only on the *specification* of the other modules that the verified module is interacting with. Several sound modular verification tools for C have been proposed [29, 61]. However, the soundness properties of these verifiers have an important limitation. To the best of our knowledge, all soundness results for modular C verifiers have the form: under the condition that *all* modules of a program have been verified, any execution of that program will comply with the specification. In other words, as soon as there is one unverified module, all bets are off. The implementations of modules that are not verified are part of the *trusted computing base*; it is *assumed* that they comply with the specifications for these modules that were used to verify the verified part of the program. Such assumptions are particularly troublesome for memory-unsafe languages such as C, as a single memory-safety error (such as a buffer overflow) in one unverified module can in principle mess up the runtime state of *all* modules. This has several undesirable consequences:

- While testing a partially verified program, failures may still occur in the verified part of the program, and the root cause for such failures may be hard to track down. This includes both memory safety failures (e.g. dereferencing invalid memory addresses) as well as failures of assertions that were statically verified to hold.
- Security properties verified to hold in a module are not guaranteed to hold when that module is used as part of a larger, unverified program. Hence, the benefits of partial verification for security purposes are limited. In particular, in a security setting, one must consider that memory safety errors may be *exploited* by means of code injection attacks [41]. Maintaining the integrity of a verified module in such a setting is very challenging.

What is needed, is a technique for ensuring that failures cannot occur in the verified part of the program. Any runtime error should either (1) lead to a failure in the unverified part, or (2) be detected on entry to the verified part. This will entail that the state of the verified module is always valid, and that no properties that were verified to hold for this module state will ever be violated.

The approach we suggest is based on performing *runtime checks* at the *boundary* between the verified and unverified part of the program. While sound approaches

for such dynamic contract checking exist for safe languages [43, 45], to the best of our knowledge there is no system that achieves such sound contract checking for unsafe languages such as C. Furthermore, existing approaches instrument *each* memory access in the unverified part [83] or verified part [108], entailing a large performance cost, while in our approach checks are only performed when crossing the verified-unverified boundary.

The main contribution of this chapter is the development of a program transformation that, given a C program partitioned into an unverified module and a module verified by means of separation logic, transforms the verified module into a *hardened* module that includes sound and complete runtime checks at the boundaries of the module. A key problem that needs to be solved is how to make sure that memory errors (or alternatively, malicious code) in the unverified module cannot corrupt the state of the verified module, while still only performing explicit checks at the boundary. We solve this problem in two steps. First, the boundary checks perform integrity checks on the *heap footprint* of the verified module. This footprint is the part of the heap currently "owned" by the verified module, i.e., memory that the verified module is allowed to access at a certain point in the program execution, according to its separation logic contracts. On re-entry to the verified module, we check that these memory locations have not been changed by the unverified part of the program. This ensures detection of any bad heap write performed by the unverified module that could influence the further execution of the verified module. Second, we need a mechanism for protecting the integrity of local variables and control flow metadata of the verified module. For this, we rely on the secure compilation scheme to protected module architectures (PMAs), presented in Chapter 2. Early PMA prototypes [73, 113] are hypervisor-based [117], while the most recent research prototypes [85] implement this protection in hardware, thereby reducing the trusted computing base to just the processor itself. Intel recently announced hardware support for PMAs under the name Intel Software Guard Extensions (SGX) [56], hence this type of protection mechanisms will likely be broadly available in the near future.

The combination of the module boundary checks and the secure compilation protection of local variables and control flow gives us a very strong modular soundness guarantee: *no verified assertion in the verified module will ever fail at runtime, even if the module runs as part of a vulnerable application that is subject to code injection attacks.*

The remainder of this chapter is structured as follows. First, we elaborate on the problem we solve and on our proposed solution in Section 5.2 and Section 5.3. In Section 5.4, we give a precise but informal explanation of our program transformations, and in Section 5.5 we illustrate these transformations using an example program. In Section 5.6 we discuss our prototype implementation and

the results of our benchmarks. Finally, we discuss related work in Section 5.7 and summarize the chapter in Section 5.8. The next chapter will provide a formal correctness proof of the runtime checks presented here.

5.2 Problem Statement

We assume as given a separation logic-based program logic for C [104], and a sound modular static verifier that checks compliance of C functions with contracts expressed in the program logic. For concreteness, we work in this paper with the VeriFast verifier [61], but our results are not specific to VeriFast. The assertion language we use for our examples is based on the symbolic execution assertion language defined in Section 4.4.1.

For programs in which each function is statically verified and where the `main` function has an empty precondition, verification ensures that no function ever performs an illegal memory operation and that each function upholds its contract. However, verifying the entire code base of a program is often infeasible, for instance because it is too costly in terms of programmer effort. Trying to prove full program correctness properties for partially verified programs would clearly be overambitious, since there are no guarantees about the behavior of the unverified parts. However, as this section will point out, even statements concerned only with the verified parts of the program cannot be proven in general for partially verified programs written in memory unsafe languages.

We consider single-threaded C programs consisting of two parts: an unverified *context* and a statically verified module. Each function of the verified module and each function prototype used by it, specifies a separation logic contract, consisting of a precondition (**req**uires) and a postcondition (**ens**ures). Static verification ensures that the verified functions are memory safe and comply with their specifications, but only under the assumption that the precondition holds on function entry and that any function called from those functions complies to its own specification.

Consider the example program shown below. On the left is the context, consisting of a `main` function, a `srt` function and a prototype for a function `med`. On the right is the verified module, which contains the function `med` and a prototype for `srt`. The `med` function takes as input a non-empty list and claims that, after execution, the list still contains the original values and the return value will be the median of the input list. This function relies on functions `len`, `nth` and `srt` to perform its task. As described in Section 4.4, the verifier relies on the contracts of those functions, in addition to the proof statements in `med` itself, to prove that the function will uphold its contract. The implementation

of `len` and `nth` is not shown in the example, but they are assumed to be part of the verified module, hence the verifier can verify those functions as well. On the other hand, for the `srt` function the verified module only contains a prototype. Hence the verifier can only *assume* that its implementation will uphold the specified contract.

```
// Prototypes
int med(struct lst *l);


// Unverified functions
int main()
{
  struct lst *l = read_list();
  print(med(l));
}

void srt(struct lst *l)
{
  <unverified sort
   implementation>
}
```

```
// Prototypes
void srt(struct lst *l);
 req list(l, ?v0);
 ens list(l, ?v1) ✳
     val_eq(v0, v1) ✳
     sorted(v1);

// Verified functions
int med(struct lst *l)
 req list(l, ?v0) ✳
     0 < length(v0);
 ens list(l, v0) ✳
     result = median(v0);
{
  int s = len(l);
  struct lst *l0 = copy(l);
  srt(l0);
  <proof statements>
  return nth(l0, s/2);
}
```

Linking the two parts of the example program together and executing them may still lead to violations of the verified module specifications, if one of the functions has a *bug*. We say that a function has a bug if it does not comply with its contract. That is, there exists an execution of the function that satisfies its precondition, but exhibits an invalid memory access, violates the postcondition, or performs another function call and violates the precondition of the called function. We assume that the static verifier is *sound*, i.e., it rejects any function with a bug. Hence there are only bugs in the unverified context, not in the verified module.

For instance, if `read_list` (called from `main`) has a bug and returns an invalid (e.g., unallocated) memory address, the `len` function or some other verified function could perform an illegal memory operation. Likewise, if `srt` has a bug and violates the contract specified in its function prototype, then the verified function `med` might not uphold its contract either. Furthermore, because C is a memory unsafe language, `srt` can write to arbitrary memory locations, thereby modifying data belonging to the `med` function. For instance, `srt` could write to

the original list `l`, instead of the copy `l0` that it was given. Hence, any properties verified to hold by the verifier about `l` while verifying `med` might be violated at runtime after a call to the unverified function `srt`. Note that `srt` can also *read* memory that it is not allowed to by its contract. This is also a bug, but it will not violate any property of the verified module assumed by the verifier, hence our runtime checks will allow this.

How bad the effects of bugs in the unverified part of the program can be, depends on how the program is executed. A *safe* execution performs complete runtime checking and will detect bugs as soon as they appear. Nguyen et al. [83] propose a way to perform such executions. Every memory access is checked and contracts are checked on each function entry and exit. Hence, safe executions are expensive. It is sound to remove the runtime checks from the verified module, as the soundness of verification implies that these checks will never fail in the safe execution. But as long as there is a significant unverified part, the performance cost will be high.

Because of this performance cost, executions of C programs are usually *not* safe. Hence, executions can enter an *error state* and continue executing. We say an execution is in an error state if it has performed memory accesses resulting in undefined behavior according to C semantics or if the execution has violated some of the separation logic specifications. An execution can only *enter* an error state in a function with a bug. That function is the *root cause* of the error state. An execution *fails* at the point where it detects the error state and terminates. Safe executions fail immediately at the root cause of a bug, but other executions may continue after entering an error state. Typically what happens then is implementation-dependent: the program behavior depends on details of the compiler and the machine on which the compiled code is executed. Most C compilers will generate code that may detect some error states such as the dereference of a memory address that the operating system has not even allocated to the program. But in general, failure of the execution may happen long after going into an error state. As a consequence, executions may enter an error state in the unverified context, but then fail in the verified module. The verified module may also be operating while in an error state, yet *not* fail, and possibly further mess up the runtime state and worsen the error state of the execution. This is exactly why partial verification is less useful than it could be, as discussed in the introduction.

We have developed efficient runtime checks to be inserted at the boundary between verified and unverified code, that make sure that no failures can occur in the verified module. Executions can enter an error state while executing in the unverified context and the execution may then continue in an error state, but we have the guarantee that *any error state that might impact the verified module will be detected before control flow enters the verified module.* As a

consequence, we have that the execution never fails while control is inside the verified module.

5.3 Overview of our solution

To guarantee that error states never impact verified modules, we need to model the execution of programs with memory safety errors. We describe two such models below.

5.3.1 Control-flow safe execution

The *control-flow safe* execution models programs as commands that operate on a heap. This is a standard model of unsafe imperative programs, which we already used in previous chapters. Under this model, memory safety errors may impact any part of the heap, but they cannot modify local variables or the control flow. In other words, code-injection attacks or stack smashing are not modeled.

For the control-flow safe execution, it is sufficient to perform runtime checks at the boundary between the verified module and the unverified context. Roughly speaking, the checks that need to be performed at the boundary are the following. Each function of the verified module should (1) check that its precondition holds on entry from an unverified function, (2) check that the callee's postcondition holds after an *outcall* (i.e., a call from the verified module to an unverified function), and (3) ensure that unverified functions do not modify any heap locations that could affect the verified function's correct execution. In our approach, the first two checks are based on a translation of separation logic pre- and postconditions into equivalent C code that will check the validity of those conditions at runtime. For the third check, our approach keeps track of the *footprint* of the verified module, i.e., the memory locations that the module can read or write, and it uses an integrity check to ensure that unverified functions do not modify the contents of those locations (except for the locations explicitly allowed to be modified, i.e., those specified in the precondition of the called unverified function). Right before performing a call from a verified function to a function of the unverified context, a cryptographic checksum is calculated over the contents of the verified module's footprint, which is recalculated and compared against the original on re-entry of the verified module.

In Section 5.4 we describe a program transformation on the verified module that injects the necessary runtime checks, and in Chapter 6 we prove these checks correct for the control-flow safe execution of programs.

5.3.2 Unsafe execution

Of course, for most realistic C compilers, the control-flow safe execution model is too abstract. Control flow information and local variables (i.e., the runtime stack) are stored in the same memory space as the heap, and hence memory safety errors can also modify control flow or contents of local variables. This is particularly relevant if we consider the possibility that our program might be under attack, and an adversary provides input that triggers a memory safety error in the unverified part of the program by performing one of the many possible low level attacks against C programs [41, 122].

Hence, we also consider *unsafe* executions, where programs are compiled in the standard way to a Von Neumann style processor architecture. Under such unsafe executions, the boundary checks discussed above are insufficient, as memory safety errors might corrupt the integrity checksum that the boundary checks compute. Also corruption of the control flow or of the verified functions' local variables may lead to failures in the verified module.

To restore the property that no failures occur in the verified module, we build on the fully abstract secure compilation scheme described in Chapter 2. This compilation technique protects modules from a potentially malicious context and ensures that any possible effect that the malicious context can have on the hardened module can be understood at source code level. By composing this secure compilation technique with the program transformations for the control-flow safe execution, we get the desired property that no failures can occur in the verified module, even in the presence of stack-smashing, code injection attacks or other exploitations of memory safety errors in the unverified context.

5.4 Program transformations

This section describes how a verified module can be transformed into a *hardened* module containing runtime boundary checks, and how our prototype implements these checks. At an architectural level, the hardened module can be subdivided into a *functional* part and a *boundary checking* part (see Figure 5.1). The functional part is essentially a copy of the verified module given as input, where

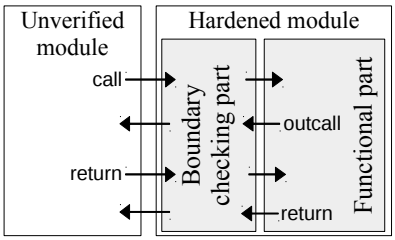


Figure 5.1: At an architectural level, a hardened module can be subdivided into a boundary checking part that performs the necessary runtime checks, and a functional part that contains the code of the original verified module.

the separation logic contracts and proof statements have been removed and the functions have been given a fresh name and marked `static` in order to remove them from the module's public interface. The boundary checking part contains new functions and data structure definitions to actually perform the runtime boundary checks. The hardened module is constructed such that each transition between the context and the functional part passes through the appropriate function of the boundary checking part.

The transformation is based solely on the source code of the verified module and the annotated function *prototypes* of the unverified part. Hence, the transformation does not require access to the source code of the unverified part. The resulting hardened module can be linked with the unverified part as-is: no recompilation of the unverified part is necessary.

The sections below explain concretely how different kinds of separation logic constructs can be translated into C code for checking them. We assume the control-flow safe execution model, since it is the execution model provided to us by the fully abstract compilation scheme of Chapter 2.

5.4.1 Pure assertions

As described in Section 4.4.1, pure assertions, as opposed to spatial assertions, do not make any claims about the heap. Such assertions can be translated straightforwardly into a C expression, as shown in the example below.

```
// Original module
int fac(int x)
 req x >= 0;
 ens result > 0;
{
  if (x == 0) return 1;
  int p = prod(x, fac(x-1));
  return p;
}

int prod(int x, int y);
 req true;
 ens result = (x * y);
```

```
// Hardened module
// (Functional part)
static int _fac(int x) {
  if (x == 0) return 1;
  int p = _prod(x, _fac(x-1));
  return p;
}

// (Boundary checking part)
static
int _prod(int x, int y) {
  int r = prod(x, y);
  if (! (r == x * y)) trap();
  return r;
}

int fac(int x) {
  if (! (x >= 0)) trap();
  return _fac(x);
}
```

All assertions in this example, i.e., the contracts of `fac` and `prod`, are pure. In the hardened module, `fac` has been replaced by an *entry stub* that first checks the precondition, before calling `_fac`, which is a slightly modified version of the original `fac`. The only functional difference is that the modified version calls the `_prod` *outcall stub* instead of the original function `prod` in the context. The outcall stub first calls `prod` in the context and then checks whether the postcondition holds. If any check fails, the `trap` function is called, which ends execution. The functions `_prod` and `_fac` have been marked `static` to indicate they are not in the public interface of the hardened module.

5.4.2 Spatial assertions

Spatial assertions describe (parts of) the heap: they indicate that a certain memory region should contain certain values. The assertions need not necessarily specify exactly what those values are, they can instead existentially quantify over them, by binding a logic variable. The difficulty with spatial assertions is that a function in the context can overwrite these values, even though that function might not be allowed to do so by its contract, thereby possibly violating properties of the verified module verified to hold by the verifier. In separation logic terms, this corresponds to a violation of the *frame rule*. As described in

Section 5.3, we use a cryptographic checksum over the memory footprint of the hardened module to solve this problem.

```
// Original module
struct pair {int a, b;};


void f(struct pair* p)
 req (p->a ↦ ?a) ✳
     (p->b ↦ ?b);
 ens (p->a ↦ −) ✳
     (p->b ↦ −);
{
 <...>
 ct(p);
 <...>
}

void ct(struct pair* p);
 req p->a ↦ ?n;
 ens (p->a ↦ ?m) ✳
     (m = n + 1);
```

```
// Hardened module
struct pair {int a, b;};

static
void _f(struct pair* p) {
 <...>
 _ct(p);
 <...>
}

static
void _ct(struct pair* p) {
 struct hashval h0, h1;
 int n = intp(&(p->a), C);
 fp_hash(&h0);
 ct(p);
 fp_hash(&h1);
 if (!eq(&h0, &h1)) trap();
 int m = intp(&(p->a), P);
 if (m != n+1) trap();
}

void f(struct pair* p) {
 a = intp(&(p->a), P);
 b = intp(&(p->b), P);
 _f(p);
 intp(&(p->a), C);
 intp(&(p->b), C);
}
```

The code above shows our approach. In the hardened module on the right, the verified function `f` has been replaced by an entry stub that first calls `intp` for both integer *points-to* assertions in the precondition of the original `f`, then calls `_f` and finally calls `intp` again for both integer *points-to* assertions in the postcondition.

The `intp` function is a *data type checking function* provided by our runtime checking system. It takes as arguments a pointer `p` to an integer and an enumeration value `C` or `P`, and performs two important functions. The function first checks whether `p` points to a valid integer, which it does by simply reading `*p`. Secondly, the function adds or removes the memory region occupied by the

integer (i.e., the memory region of `sizeof(int)` bytes starting at address `p`) to or from a global data structure describing the hardened module's current footprint. When the enumeration value is `P` (for *produce*), the memory region is added to the footprint and when it is `C` (for *consume*) the region is removed. When adding a region to the footprint, `intp` checks that the region does not overlap with the existing footprint. This corresponds to the semantics of the separating conjunction, which requires that the footprint of each of its conjuncts occupies a *disjoint* part of the heap. Similar to VeriFast and other separation logic-based tools, we do not support non-separating conjunction or negation. However, as argued in Section 4.4.1, this does not pose expressibility problems in practice. If all checks pass, the `intp` function returns the integer value at the given address, or ends execution by calling `trap` otherwise. Functions similar to `intp` are provided for other primitive data types (`char`, `unsigned int`, ...) and pointers, because the memory sizes of those data types can differ.

In the function `_f` of the hardened module, the call to the `ct` function of the context has been replaced by a call to the `_ct` outcall stub. This stub first removes (consumes) the footprint of `ct`'s precondition from the hardened module's footprint, before calculating a hash over the memory regions described by the remaining footprint. This hash is stored in the local variable `h0`, where it is protected from the context by the secure compilation scheme. Next, the stub calls `ct`, handing over control to the context. When the context function returns, the outcall stub verifies that the memory described by the footprint has not been tampered with, by recalculating the hash and comparing it to the original value stored in `h0`. Finally, the stub checks whether the postcondition of `ct` holds by producing its footprint and checking whether the values in the corresponding memory region adhere to the contract. Note that we do not prevent the context from *reading* memory it is not supposed to by its contract, because this can never violate properties of the verified module assumed by the verifier.

For our prototype, we implemented the footprint data structure as a radix trie. This data structure supports $O(k)$ addition, removal and overlap testing, and $O(n)$ traversal, with k the number of bits in a memory address (e.g., 64 for a 64-bit CPU) and n the number of memory regions in the trie. The footprint description must be protected from being overwritten by memory safety errors in the unverified context and hence we store it in the protected data memory section provided by the PMA (see Section 2.2.3).

5.4.3 Predicates

As described in Section 4.3.3, separation logic predicates are named, parametrized assertions, used to provide data encapsulation and to describe recursive data structures. For instance, the following predicate describes a linked list of integers of a certain size.

```
struct list { int value; struct list* next; };

pred list_pred(struct list* l; int size) =
  l = 0 ? (size = 0) : (l->value ↦ _) ∗ (l->next ↦ ?n) ∗
    list_pred(n, ?size0) ∗ (size = size0 + 1);
```

Predicates can have an arbitrary number of parameters and separation logic allows us to existentially quantify over each of them. For instance, a valid precondition could be `list_pred(?l, 5)`, meaning that there must be a linked list of size 5 somewhere on the heap. Translating this quantification into a runtime check would however be problematic, since in general the entire heap would have to be examined in order to bind a value to `l` at runtime. To overcome this problem, we require predicates to be *precise*, which is accomplished by separating predicate parameters into input and output parameters and by allowing only output parameters to be quantified over when using a predicate. In a predicate definition, each output parameter of the predicate must be assigned a value in all execution paths of the predicate's body. The points-to assertion `x` $\mapsto$ `y` is treated as a precise predicate, of which `x` is an input parameter and `y` is an output parameter. In our VeriFast-based assertion syntax, parameters before the semicolon in the parameter list of a user-defined predicate definition are input parameters and the other parameters are output parameters. Hence, for `list_pred` defined above, `l` is an input parameter and `size` is an output parameter. We discuss the further implications of our preciseness requirement in Section 6.3.1.

The code below shows the transformation of the `list_pred` predicate. It is a *predicate checking function* with one more parameter than the original predicate. This extra parameter is an enumeration value, indicating whether the predicate will be used for consumption or production and its value is simply passed on to the data type checking functions (e.g. `intp`) described in Section 5.4.2. Input parameters have the same type in the runtime checking function as in the original predicate and output parameters are *pointers* to the type of the parameter in the original predicate.

```
static void
list_pred(struct list* l, int* size, enum op_type op) {
  if (l == 0) {
    *size = 0;
  } else {
    intp(&(l->value), op);
    struct list* n = ptrp(&(l->next), op);
    int size0;
    list_pred(n, &size0, op);
    *size = size0 + 1;
  }
}
```

The predicate's body is transformed straightforwardly into equivalent C code. When an output parameter is constrained to a value in the predicate body using the equals operator, that value is assigned to the corresponding pointer parameter in the predicate checking function. As exemplified by the recursive call to `list_pred`, a predicate call assertion is transformed into a call to the corresponding predicate checking function. If an assertion uses a constant value or previously bound variable for an output argument instead of binding a new variable (e.g. `list_pred(l, s)` with `s` already bound, instead of `list_pred(l, ?s)`), then this is pre-transformed to first binding a fresh variable to the output parameter and then constraining it with an equality (e.g. `list_pred(l, ?s0) ∗ s0 = s`). This allows the core transformation to assume that output arguments are always existentially quantified.

5.4.4 Inductive data types

While spatial assertions and predicates are in some cases sufficient to prove memory safety, they are insufficient to prove full functional correctness for most programs. For instance, the `list_pred` predicate defined in the previous section specifies the size and memory footprint of a linked list, but does not say anything about its *contents*. VeriFast supports a rich specification language with inductive data types and fixpoint functions (i.e., primitive recursive functions) over them. The example below shows how such constructs can be used for specifying functional correctness properties.

```
induct ints = ints_nil() | ints_cons(int, ints);

pred list_pred(struct list* l; ints values) =
  l = 0 ?
    values = ints_nil()
  :
    (l->value ↦ ?v) ∗ (l->next ↦ ?n) ∗
    list_pred(n, ?vs_tail) ∗ (values = ints_cons(v, vs_tail));

fixpoint int head(ints lst) {
  switch (lst) {
    case ints_nil(): return 0;
    case ints_cons(v, tail): return v;
  }
}

int get_first(struct list* l)
  req list_pred(l, ?values);
  ens list_pred(l, values) ∗ (result = head(values));
{
  <implementation omitted>
}
```

The functional behavior of `get_first` is completely specified by its contract. Our transformation translates such inductive data type specifications into *tagged structures*, a known technique for implementing variant types in C. The code below shows the data structure definitions corresponding to `ints` and its two constructors.

```
struct ints { int tag; };

struct ints_nil {
  int tag;
  // No members
};

struct ints_cons {
  int tag;
  int _1;
  struct ints* _2;
};
```

To prevent the context from tampering with instances of these data structures when the context is in control, the data must either be stored in the protected data memory section provided by the PMA, or be included in the module's footprint such that it's incorporated in the cryptographic checksum described in Section 5.4.2. We chose to store the data in the protected data memory section, which is faster than the checksum-based approach, but does require more protected memory.

Besides these structure declarations, the transformation also generates an equality comparison function for each inductive data type. Finally, fixpoint function definitions are translated straightforwardly into equivalent C functions.

5.4.5 Function pointers

VeriFast allows programs using function pointers to be verified, by letting users associate contracts with *families* of functions. The code below shows how a verified module can use a function pointer to call a function in the unverified part.

```
typedef int int_func(int x);
  req true;
  ens result > 0;
```

```
void f(int_func* g, int x)
  req is_int_func(g);
  ens true;
{
  int y = g(x);
  <...>
}
```

The typedef on the left defines `int_func` as the family of functions that take an integer argument and return a strictly positive integer result. On the right, the function `f` takes a pointer `g` to such a function, applies it to a local variable `x` and stores the result in `y`. The contract for `g` is specified by the `is_int_func(g)` assertion in the precondition of `f`, which refers to the typedef on the left.

Our transformation handles function pointer calls in almost the same way as it handles regular calls. More precisely, an outcall stub is generated for each defined function pointer typedef, and the hardened module calls this outcall stub instead of calling corresponding function pointers directly. Function pointer outcall stubs take as an argument a pointer to the concrete function to call. For instance, for the example code above, the function pointer outcall stub would be as follows.

```
static int _int_func(int_func* f, int x) {
   <calculate footprint hash>
   int result = f(x);
   <verify footprint hash, check postcondition>
   return result;
}
```

Indirect function calls from the context to the hardened module are also supported naturally. Since all functions of the functional part of the hardened module have been made `static`, the only publicly accessible functions of the

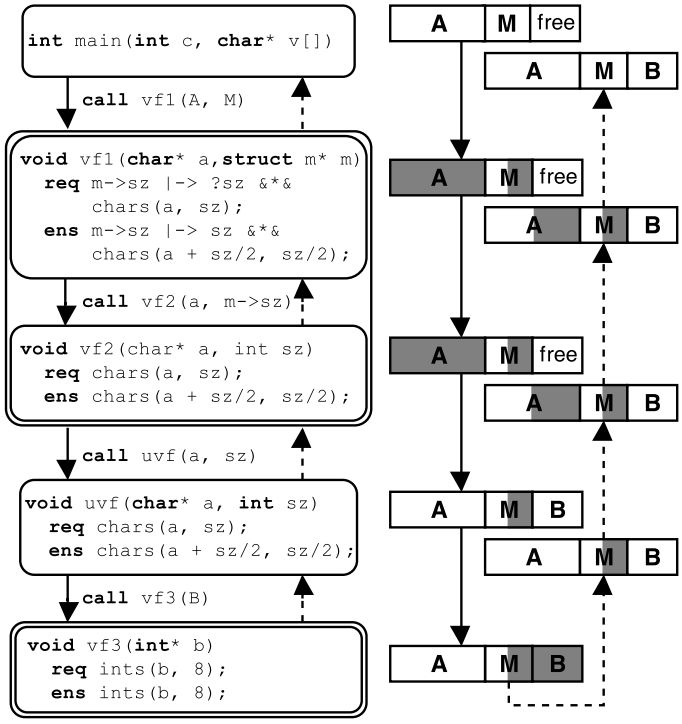


Figure 5.2: When executing the example program on the left, the footprint evolves as shown on the right. The single-bordered boxes on the left are unverified functions while the double-bordered boxes are verified functions. The boxes on the right represent the heap, with the grayed-out parts representing the hardened module's footprint. Solid lines are function calls and dashed lines are returns.

module are those in the boundary checking part. The fully abstract compilation scheme uses the PMA's restriction on module entry points to ensure that only those public functions can be called from the context at runtime. Hence, the necessary runtime checks are performed whenever the unverified part calls the hardened module through a function pointer.

5.5 Example program

This section uses an example program to illustrate how the transformations described above affect the hardened module's footprint description. Figure 5.2

depicts the example program and the footprint at various execution points. Note that the program is an abstract example created to illustrate the footprint evolution under various control flow transitions, and is not intended to model any useful computation. The program consists of the verified functions `vf1`, `vf2` and `vf3`, and, in addition to the standard C library, two unverified functions `main` and `uvf`. The unverified function `uvf` is annotated with a separation logic contract, so it can be called from the verified functions. The function signatures and contracts shown in Figure 5.2 refer to a `struct m` and predicate `chars`, defined as follows.

```
struct m {
  int x; int sz;
};
```

```
pred chars(char *a, int sz) =
  sz <= 0 ? true :
    char(a) ∗ chars(a+1, sz-1);
```

A `chars(a, sz)` instance represents a character array of size `sz`, starting at heap address `a`. The `char` predicate used by `chars` is a VeriFast primitive which asserts that its argument points to a valid character in memory. Its translation is a call to the `charp` data type checking function (similar to `intp` described in Section 5.4.2).

We now discuss how each of the function calls shown in Figure 5.2 affects the hardened module's footprint description. Assume `main` allocates an array `A` of `N` bytes and an instance `M` of `struct m`, using the standard (unverified) `malloc` function, and assigns the value `N` to `M`'s `sz` field. At this point, the heap contains `A` and `M`, and the footprint description is empty. Next, `main` calls the verified function `vf1(A, M)`, and hence this function's entry stub will check whether its precondition holds. The precondition check consists of reading the `sz` field of `M` and verifying that `A` is in fact a character array of size `N`. As part of this check, the memory occupied by `M`'s `sz` field and the entire array `A` will be added to the footprint description. The memory occupied by `M`'s `x` field is not mentioned in `vf1`'s precondition and hence is not added to the footprint description.

Next, `vf1` calls `vf2(a, m->sz)` directly, without passing through a boundary checking function, because both caller and callee are in the hardened module. No runtime checks are performed and hence the footprint description remains the same.

The `vf2` function now makes an outcall `uvf(a, sz)`, which passes through the corresponding outcall stub. The stub first removes the array `A` from the footprint description, because it is referenced in `uvf`'s precondition, and then hashes the memory in the remaining footprint description (consisting of only the `sz` field of `M`), before calling `uvf`.

We assume `uvf` allocates an array `B` of eight integers, again using the standard

`malloc` function, and then calls `vf3(B)`. This function's entry stub will verify that `B` is a valid array of eight integers and will add `B` to the footprint description. The `vf3` function now executes its (unspecified) body and then returns, thereby removing `B` from the footprint description, as specified by its postcondition.

Now `uvf` is back in control and we assume it returns immediately, to its outcall stub in the hardened module. The stub will first check that the memory in the footprint (still consisting of only `M`'s `sz` field) has not been modified, by recalculating the hash and comparing it to the original. Execution is aborted if any change is detected. If the new hash matches the original, the stub checks whether `uvf`'s postcondition holds and adds the second half of `A` back to the footprint description, as specified by `uvf`'s postcondition. The stub then returns control to `vf2`. Note that it is impossible that `vf2` now tries to access the first half of `A`, since this would have been detected by the static verifier when verifying `vf2`.

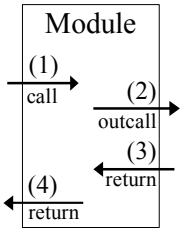
The `vf2` function now returns to `vf1`, without any change in the footprint description, because both functions are part of the hardened module. Finally, `vf1` returns to `main`, removing the second half of `A` and `M`'s `sz` field from the footprint description, as specified by `vf1`'s postcondition. Control is now back at the `main` function and the footprint description is empty.

5.6 Prototype performance

We have implemented the transformations described in Section 5.4 as a source-to-source translator written in OCaml. This translator takes as input a verified C module with VeriFast annotations (including annotated function prototypes for any function of the context called from the verified module), and outputs a hardened version of the module. The translator reuses significant parts of the existing VeriFast codebase, such as its lexer, parser and type checker. Although VeriFast's license prevents us from releasing the source code, a binary version of the translator is available online[1].

In the sections below, we describe the results of measuring the performance impact of the inserted runtime checks versus the verified module without any runtime checks. We ran micro and macro benchmarks on a standard desktop system, without a protected module architecture, in order to quantify the overhead of *just* the runtime checks, and we discuss the additional overhead introduced by a PMA separately in Section 5.6.3. All benchmarks were compiled with GCC 4.8.2, using optimization level 3, and were executed on a system with

[1] https://distrinet.cs.kuleuven.be/software/sound-verification



	Check assertion validity	Add/remove assertion footprint	Hash module footprint
(1)	X (pre)	X	
(2)		X	X
(3)	X (post)	X	X
(4)		X	

Figure 5.3: Actions performed for every type of boundary transition.

a 3.10 GHz Intel Core i5-2400 CPU with 8 GiB of RAM, running Ubuntu 14.04. The hash function used to calculate the hash over the footprint when performing an outcall is BLAKE2b [52].

5.6.1 Micro benchmarks

Since our transformations introduce checks at the *boundary* between the verified and unverified part, there will be a performance overhead when crossing the verified-unverified boundary. During a boundary check, up to three actions are performed: (1) checking whether the assertion (pre- or postcondition) holds, (2) adding or removing the assertion's footprint from the footprint description maintained by the module, and (3) hashing the memory in the module's footprint description Figure 5.3 shows which actions are performed for each kind of boundary transition. We measured the contribution of each of these factors using two micro benchmarks based on simple data structures, similar to those used in [83].

The first micro benchmark is a verified module that takes as input a linked list of integers and sorts it using insertion sort. The second micro benchmark is another verified module that does an in-order traversal of a binary search tree to produce a sorted linked list. Both modules have been verified for memory safety (i.e., not for full functional correctness). The entry point signatures of these two modules are as follows.

```
struct list_node* insertion_sort(struct list_node* l);
  req list_pred(l);
  ens list_pred(result);

struct list_node* bst_to_list(struct bst_node* bst);
  req bst_pred(bst, ?v);
  ens bst_pred(bst, v) ∗ list_pred(result);
```

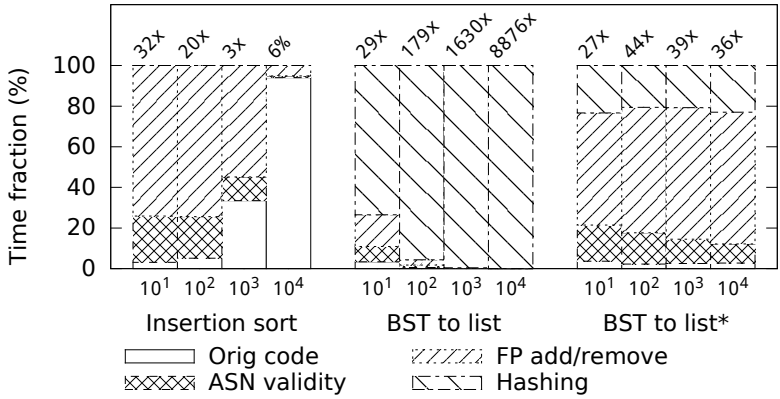


Figure 5.4: The execution time distribution over the different runtime checking actions for our micro benchmarks, for different input lengths. The numbers above the bars indicate the total execution time overhead in comparison to the unhardened code.

Figure 5.4 shows the distribution of execution time over the different actions performed by the runtime checks for these benchmarks, for input lengths of 10^1, 10^2, 10^3 and 10^4 elements. The number above each bar indicates the total overhead in comparison to the unhardened code.

The left bar chart shows that, for small input sizes, the insertion sort module spends significant time modifying the footprint description and checking assertion validity. As the input size increases however, the relative overhead due to these actions drops to the point where it becomes insignificant. This is because modifying the footprint and checking assertion validity are $\mathcal{O}(n)$ operations that are only performed when entering or exiting the module, and insertion sort is a $\mathcal{O}(n^2)$ algorithm. Hence, the time spent doing useful calculations inside the module increases faster than the time spent performing runtime checks. No time is spent on hashing, because the benchmark does not make any outcalls.

The middle bar chart shows that the BST module spends almost all of its time hashing its footprint, resulting in a huge performance overhead that increases with increasing input size. This is because the memory for the output list is allocated piece-by-piece while traversing the input, and hence the benchmark performs an outcall to `malloc` for each node of the input BST that it visits. Because hashing the module footprint is an $\mathcal{O}(n)$ operation and it is performed n times, we have an $\mathcal{O}(n^2)$ hashing overhead. Since the BST to list algorithm itself is only $\mathcal{O}(n)$, the hashing overhead quickly dominates the execution time. It is however possible to reduce the hashing overhead to $\mathcal{O}(n)$ by using a slightly

modified algorithm that first calculates the size of input BST and then allocates memory for the entire output list with a single `malloc` call. This will cause the module footprint to be hashed only once, instead of n times. The performance overhead of this algorithm is shown in the right bar chart of Figure 5.4. While the relative overhead is still significant, it now remains constant with increasing input size. This demonstrates that the choice of boundary between verified and unverified code, and the number of times this boundary is crossed, can have a large impact on performance.

5.6.2 Macro benchmarks

While the micro benchmarks from Section 5.6.1 show how the execution time overhead is distributed over the different actions performed during a runtime check, they do not show one of the major advantages of our approach: the fact that there is no performance impact on code running completely in the verified or in the unverified part but not transitioning between the two. To show this effect and to assess the real-world feasibility of our approach, we have constructed three realistic macro benchmarks in which we verify and harden a small, security-critical part of an application, but leave the bulk of the application unverified. We measured both the execution time and memory overhead for these macro benchmarks.

Apache httpd modules

The first two macro benchmarks are modified Apache httpd authentication modules, which are used by the web server for verifying user credentials (for instance as part of HTTP Basic Authentication). The first Apache benchmark is based on the standard `mod_authn_anon` module, and uses a single pair of valid username/password credentials hardcoded in memory. The other benchmark is based on the standard `mod_authn_file` module, which reads the list of valid credentials from a file on disk. Both modules provide a single entry point function that takes client credentials (sent to the web server by the client) as input and returns an integer indicating whether or not they are valid. The signatures of these functions are shown below.

```
int check_password_mem(char *u, char *p);
  req string(u, ?user) ✳ string(p, ?pass);
  ens string(u, user) ✳ string(p, pass) ✳
      (result = 1) ?
        (user = "username") ✳ (pass = "secret")
      :
        (result = 0) ? true : (result = 2);
```

Table 5.1: The execution time overhead of the program transformations is below 4% for the macro benchmarks.

	Execution time (s)		
	unhardened	**hardened**	**overhead**
mod_authn_anon	33.164	33.388	0.224 (0.68)%
mod_authn_file	33.554	34.809	1.255 (3.74)%
ftpd	23.193	23.242	0.049 (0.21)%

Table 5.2: The peak memory overhead of the program transformations is below 7% for the macro benchmarks.

	Peak resident set size (KiB)		
	unhardened	**hardened**	**overhead**
mod_authn_anon	33,356	33,384	28 (0.08%)
mod_authn_file	33,324	35,524	2,200 (6.60%)
ftpd	952	976	24 (2.52%)

```
int check_password_file(char *u, char *p);
  req string(u, ?user) ✳ string(p, ?pass);
  ens string(u, user) ✳ string(p, pass);
```

The modules' code consists mainly of outcalls to various I/O and string processing functions of the standard library. In particular, the memory-based module performs 2 such outcalls per HTTP request, while the file-based module performs 34. As can be seen from the signatures above, the memory-based module has been verified for full functional correctness, while the file-based module has only been verified for memory safety, but this makes no difference at runtime. The path of the valid credentials file has been hardcoded in the source of the file-based module.

We set up the pre-forked version of Apache httpd 2.4.7 to serve the default WordPress 3.9.1 sample website with a MySQL 5.5.37 database back-end. We used the Apache HTTP server benchmarking tool `ab` to measure the time required to perform 5,000 HTTP requests using 10 concurrent client threads. The client and server were executed on the same host to eliminate any network bottlenecks, and we made sure the web server did not use any form of credential caching. The memory overhead was measured by comparing the peak resident set size of the modules.

The first two rows of Table 5.1 and Table 5.2 show the results for the Apache benchmarks. The execution time overhead is low, averaging at 0.68% and 3.74% over three benchmarking runs for the memory-based and file-based module respectively. The memory overhead is also low, averaging at 0.08% and 6.60%. The difference in overhead between the two modules is due to the different number of outcalls they perform and because the file-based module needs a relatively large buffer for reading lines from the password file. This buffer is part of the module's footprint and hence needs to be described by the footprint description and hashed when making outcalls.

NetKit FTP daemon

The NetKit FTP daemon is an FTP server shipped with many current Linux distributions. It contains a `checkuser` function that is used to determine whether the names of users trying to log in appear in the `/etc/ftpusers` file of blocked users. We have verified this function and have moved it into a separate module, which we then hardened with our prototype translator. The signature of this function is shown below.

```
int checkuser(char *fname, char *name);
  req string(fname, ?fn) ✳ string(name, ?n);
  ens string(fname, fn) ✳ string(name, n);
```

The implementation of this function is quite similar to the `mod_authn_file` Apache module, performing 30 outcalls to various I/O and string processing functions per FTP session. The benchmark consists of performing 500 FTP sessions using 10 concurrent client threads, where each session consists of a user logging in, downloading a 1 KiB file and then disconnecting again.

The third row of Table 5.1 and Table 5.2 shows the results obtained by taking the average of three benchmarking runs. Both the execution time and memory overhead are again low, confirming our claim that real-world applications consisting mainly of unverified code plus a small hardened module, incur only a small performance overhead.

5.6.3 PMA overhead

As explained in Section 5.3, our runtime checks assume a control-flow safe execution model, which we achieve using the fully abstract compilation process from the hardened source code to a PMA, as described in Chapter 2. Since the micro and macro benchmarks described above were performed on a standard desktop system without a PMA, their results do not yet represent the overhead

of our full end-to-end approach. Although recent developments [56] indicate low-overhead hardware-based PMA platforms will be available for commodity desktop systems in the near future, the currently available PMA prototypes are still in an experimental state, prohibiting us from running meaningful end-to-end macro benchmarks on top of them. The micro benchmarks described in Section 2.3.3 indicate there is a 85× performance overhead for calling a protected module function compared to calling an operating system driver function, but this figure represents only the execution time difference for performing a function call. Most of this overhead is due to making the context switch from user-mode to the protected module. When there is some actual computation involved and when the PMA is implemented in hardware instead of using a hypervisor, the overall overhead is likely to be much smaller.

In order to get a more realistic figure for the end-to-end overhead of our approach, we developed a benchmark for Sancus [85], which is a fully-functional hardware-based PMA for low-end networked microcontrollers. The Sancus prototype consists of a fully abstract compiler towards a small PMA-enabled 16-bit microcontroller (based on the TI MSP430) featuring 48 KiB of ROM and 10 KiB of RAM. Although this platform is too resource-constrained to run our macro benchmarks, we developed a micro benchmark that is similar to the code example given in Section 5.2. The benchmark consists of a hardened module that provides a function for calculating the median of a linked list of integers. The function's precondition asserts that the list is a valid non-empty linked list and its body performs three outcalls: one to copy the list, one to sort the copy and one to free the copy before returning. The hash function used for this benchmark was SHA-256. The results indicate the total overhead of Sancus (both the secure compiler and the platform) is below 1%.

5.6.4 Reducing hashing overhead

The micro benchmarks show that considerable time is spent hashing the module's footprint. One way of reducing this overhead is to do away with hashing and instead copy the entire module footprint contents to a secure location in memory (e.g., the PMA's private memory region) when making an outcall and to check the footprint against this copy on return. Our experiments show that this gives a performance benefit of between 0% and 20% in comparison with hashing, but it obviously requires much more protected data memory.

Another potential performance issue is that, as the verified codebase of an application grows, the size of the hardened module's footprint grows as well, which means more data must be hashed on each boundary transition. However, as the verified codebase grows, the part of the data that is used *exclusively* by

the verified part of the application is likely to grow as well. Hence, this data could be placed in protected data memory, where it can be accessed only by the hardened module and hence need not be hashed on boundary transitions.

An interesting way to solve both issues would be by taking advantage of hardware page protection support to reduce the amount of data needing to be hashed on boundary transitions. If an entire memory page is part of the module's footprint, it can be marked read-only in hardware before making an outcall and be reverted to read-write access on return. However, memory pages are typically at least 4 KiB in size, making this approach too coarse-grained to be used directly. A hybrid approach where pages that are completely in the footprint description are set read-only and the rest of the footprint is hashed or copied, is viable, but we consider this to be future work.

5.6.5 Summary

Our micro benchmarks indicate that the performance overhead of the runtime checks can be significant if there is little computation in- or outside the verified module, compared to the computation required for the boundary checks. Most of this overhead is due to hashing the module's footprint and adding/removing memory regions to/from the footprint description. Nevertheless, the macro benchmarks show that this overhead becomes negligible once more computation is performed in the unverified context. Hence, when developing modules to be verified and hardened, it is critical that the boundary between verified and unverified code is chosen wisely. That is, developers should try to minimize the number of verified/unverified boundary crosses in order to minimize the performance overhead. Although we could not run our full set of benchmarks on a PMA-enabled system, a separate benchmark performed on Sancus indicates the platform overhead is negligible.

5.7 Related work

Separation logic-based formal verification ensures memory safety, which can be considered one of its main advantages for memory unsafe languages such as C. There are however many other notable solutions for making C memory safe, such as Safe-C [13], CCured [80] and Cyclone [64]. These systems rely on a combination of type system extensions, static analyses and runtime checks to ensure memory safety, but make no attempt at providing correctness guarantees beyond that. Furthermore, these solutions protect against input-providing attackers, while we protect against more powerful in-code attackers,

i.e., attackers that have already gained the ability to execute code in the unverified part.

The idea that software modules should specify contracts in the form of pre- and post-conditions was popularized by Meyer [75] in the programming language Eiffel. Such contracts can then be checked statically or dynamically, and there is a huge amount of literature both on static and on dynamic checking of contracts. Some notable examples include the Java Modeling Language (JML) based tools [23], and .NET Contracts [18].

Our approach combines modular static verification with runtime checking, to achieve a non-trivial soundness property in the context of an unsafe programming language. The approach is based on separation logic [104] so that there is a clear notion of memory ownership and we can compute the footprint (i.e., the owned region of memory) of a module and take a snapshot of that region's contents. For our implementation and experiments we have used the VeriFast [61, 58] separation logic-based assertion language and static program verification tool for C and Java. Other separation logic-based program verifiers include Smallfoot [19], JStar [38], HIP [26], and Space Invader/Infer [25]. Another notable modular static verification tool for C programs is VCC [29]. However, instead of separation logic, it uses a verification logic that is heavily based on ghost variables, so it is not clear how one would generate runtime checks for module specifications written in VCC's annotation language.

Runtime checking of separation logic assertions is known to be challenging because of the frame rule. A related approach is that of Nguyen et al. [83]. Although some of the techniques used in their approach are similar to ours (e.g., tracking footprints and splitting predicate parameters into input and output parameters), their objective is different from ours. Their runtime checker aims to stay as close to the standard separation logic semantics as possible, while our approach only aims to ensure that no failures can occur in verified code. We can hence allow unverified code to read arbitrary memory, which is not allowed under standard separation logic semantics. Nguyen et al. use a heap coloring technique and runtime checks at every method invocation and field access in unverified code to check framing. This introduces a large performance overhead (on the order of $10,000\times$ if all necessary checks are done) that increases as the size of the unverified code grows. As shown in Section 5.6, the relative performance impact of our approach *decreases* with a larger unverified codebase. Also, since the implementation of Nguyen et al. needs to instrument unverified code, the entire codebase must be recompiled, whereas we only need access to the verified module. Finally, the implementation of Nguyen et al. is for Java, so they do not address the complications related to the lack of memory safety of C.

Another related approach is that of Yarra [108], in which runtime checks are used to protect C programs from non-control data attacks. Developers must annotate critical data structures with special type declarations, from which point on those data structures should only be accessed using those special types. In its *whole program protection mode*, runtime checks are inserted throughout the entire codebase to detect illegal accesses to the critical data structures, causing a large performance overhead. In its *library protection mode* however, only the memory accesses of a small core of the application (loosely corresponding to the verified module of our approach) are instrumented. Critical memory writes in the core are modified to maintain a shadow copy of critical objects on separate memory pages, which are made read-only using hardware page protections before calling untrusted code. Critical memory reads in the core are instrumented to check consistency of both copies, thereby detecting unauthorized writes to critical objects from untrusted code. The library protection mode is similar to how we enforce the separation logic frame rule, in the sense that critical regions of memory are integrity protected when calling untrusted code. Our solution provides stronger guarantees than Yarra however, since we ensure validity of arbitrary separation logic assertions, instead of only data structure integrity. Also, Yarra does not prevent untrusted code from disabling the shadow page protections, making it vulnerable to code-injection attacks in the unprotected part. Finally, although the performance cost of Yarra's library protection mode is low, it grows with both the number of boundary crossings *and* the number of reads and writes to critical data in the core part of the application.

Kosmatov et al. [67] described the runtime checking of E-ACSL annotations for C programs, in the context of the Frama-C platform. E-ACSL is an executable subset of ACSL, a behavioral specification language for C programs. Both function contracts and in-body annotations can be specified and can be translated into runtime checks by the E-ACSL2C translator. In order to perform such runtime checks, each memory allocation, deallocation and variable assignment is instrumented to record information about the modified region of memory into a dedicated *data store*. This store hence contains a copy of the program's data and some meta data about it. The runtime pre, post and in-body annotation checks query the store in order to determine the annotations' validity. Although the approach mentions the use of static analyses to statically discard some of the runtime checks, there is no notion of a verified and an unverified part. Hence, the entire program must be instrumented for the checks to be sound and complete. This results in a high overall performance cost, ranging from 13× to 800×.

The problem of checking contracts at the boundary between statically checked modules and unchecked modules has also been studied extensively in higher-order programming languages. Findler and Felleisen pioneered this line of work

and proposed higher-order contracts [43], which have been implemented in the Racket programming language [45]. The main challenge addressed is that of function values passed over the boundary. Compliance of such function values with their specified contract is generally undecidable. But it can be handled by wrapping the function with a wrapper that will check the contract of the function value at the point where the function is called. This is similar to how we handle function pointers: the corresponding contract is checked when the function is called. One concern that has received extensive attention is the proper assignment of *blame* once a contract violation is detected [51, 37]. While this line of research shares our goal of safely composing a statically checked module with an unchecked module, the issues of higher order contracts and blame assignment are largely orthogonal to the problems we addressed in this chapter.

5.8 Summary

Separation logic-based verification of C code is a powerful technique for guaranteeing the absence of code failures. However, verifying large programs is difficult and requires significant expertise and developer effort. Modular verification tools support partial verification, where only the most critical modules are verified, and where over time more and more modules get verified. Unfortunately, this kind of partial verification gives only limited guarantees at runtime. Bugs in the unverified part of the program can also impact the state of the *verified* part, and hence might trigger failures in verified modules.

We have proposed a way to transform and compile partially verified programs such that the runtime guarantees are significantly better. After our code transformations, no failures can ever occur in the verified module; if a bug is triggered in the unverified part of the program, this is detected before it can impact the state or control flow of the verified module. This is useful for testing, as it detects bugs faster, and for security as it can guarantee verified properties of modules even in the presence of code injection attacks against the unverified part of the program. We have tested the performance of the transformed code, and have found that the overhead is low if the boundary between verified and unverified code is chosen wisely, demonstrating the real-world feasibility of our approach.

Chapter 6

Sound Verification in an Unverified Context: Formalization

Publication data

P. Agten, B. Jacobs, F. Piessens. *Sound modular verification of C code executing in an unverified context: extended version.* CW Reports CW676. Department of Computer Science, KU Leuven, Nov. 2014

P. Agten, B. Jacobs, F. Piessens. "Sound Modular Verification of C Code Executing in an Unverified Context". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL '15. Mumbai, India: ACM, 2015, pp. 581–594

Pieter Agten is the main contributor to these works, which were conducted under supervision of Frank Piessens and Bart Jacobs.

6.1 Introduction

This chapter formalizes the program transformations described in Chapter 5 and proves them to be *safe* and *precise*. Safety means that the hardened module does not fail, even when it is interacting with a context that does not uphold

$$E ::= n \mid x \mid E + E \mid E - E$$

$$B ::= \texttt{true} \mid \texttt{false} \mid E = E \mid E < E \mid \neg B$$

$$C ::= \textbf{skip} \mid x := E \mid \textbf{if } B \textbf{ then } C \textbf{ else } C \mid x := r(\overline{x}) \mid x := \textbf{alloc} \mid$$
$$x := [x] \mid [x] := x \mid \textbf{dealloc}(x) \mid \textbf{assert}(B) \mid \textbf{alloced}(x) \mid C; C$$

$$l, n \in \mathbb{N}^+ \quad x, y, z \in \mathit{Vars}$$

$$\mathit{rdef} ::= \textbf{routine } r(\overline{x}) \ = C$$

Figure 6.1: Syntax definition of our imperative language.

its contracts. Precision means that the hardened module behaves exactly like the original verified module when interacting with a context that *does* uphold its contracts.

As in Chapter 3, we have tried to divide our efforts in the most efficient way between developing a thorough implementation of our translator (see Section 5.6) and a rigorous formalization. We have again focused on proving the theorems and lemmas that contribute most to convincing the reader of the correctness of our results, marking the less rigorously conducted proofs as *proof sketches*. Further focusing on the essentials of our transformation, we do not formalize inductive data types nor function pointers.

The further outline of this chapter is as follows. We first define a simple imperative programming language that models C in Section 6.2 and define our separation logic assertion language in Section 6.3. Then, in Section 6.4, we define a function prod that translates assertions to executable program statements that check whether or not those assertions hold. This function would be sufficient to generate complete runtime checks for a module that does not perform any outcalls. After proving this function's correctness, we use it as a building block in Section 6.5 for a more complex transformation function that safely supports outcalls. In Section 6.6 we prove that programs generated by this transformation function are safe and precise. Finally, in Section 6.7 we summarize the chapter.

$$\textsc{VarAssign}\quad \frac{}{\Sigma \vdash \langle (\overline{s}, h),\ x := E\rangle \longrightarrow \Sigma \vdash \langle (s[x \to [\![E]\!]_s] :: \overline{s}, h),\ \mathbf{skip}\rangle}$$

$$\textsc{Skip}\quad \frac{}{\Sigma \vdash \langle (\overline{s}, h),\ \mathbf{skip}; C\rangle \longrightarrow \Sigma \vdash \langle (\overline{s}, h),\ C\rangle}$$

$$\textsc{IfTrue}\quad \frac{[\![E]\!]_s = \mathtt{true}}{\Sigma \vdash \langle (s :: \overline{s}, h),\ \mathbf{if}\ E\ \mathbf{then}\ C\ \mathbf{else}\ C'\rangle \longrightarrow \Sigma \vdash \langle (s :: \overline{s}, h),\ C\rangle}$$

$$\textsc{IfFalse}\quad \frac{[\![E]\!]_s = \mathtt{false}}{\Sigma \vdash \langle (s :: \overline{s}, h),\ \mathbf{if}\ E\ \mathbf{then}\ C\ \mathbf{else}\ C'\rangle \longrightarrow \Sigma \vdash \langle (s :: \overline{s}, h),\ C'\rangle}$$

$$\textsc{HeapRead}\quad \frac{(s(x') \to n) \in h}{\Sigma \vdash \langle (s :: \overline{s}, h),\ x := [x']\rangle \longrightarrow \Sigma \vdash \langle (s[x \to n] :: \overline{s}, h),\ \mathbf{skip}\rangle}$$

$$\textsc{HeapWrite}\quad \frac{(s(x) \to n') \in h}{\Sigma \vdash \langle (s :: \overline{s}, h),\ [x] := x'\rangle \longrightarrow \Sigma \vdash \langle (s :: \overline{s}, h[s(x) \to s(x')]),\ \mathbf{skip}\rangle}$$

$$\textsc{HeapRead-Fail}\quad \frac{(s(x') \to n) \notin h}{\Sigma \vdash \langle (s :: \overline{s}, h),\ x := [x']\rangle \longrightarrow \mathbf{fail}}$$

$$\textsc{HeapWrite-Fail}\quad \frac{(s(x) \to n') \notin h}{\Sigma \vdash \langle (s :: \overline{s}, h),\ [x] := x'\rangle \longrightarrow \mathbf{fail}}$$

$$\textsc{Call}\quad \frac{\Sigma(r) = (\mathbf{routine}\ r(\overline{x}) = C)}{\Sigma \vdash \langle (s :: \overline{s}, h),\ z := r(\overline{y})\rangle \longrightarrow \Sigma \vdash \langle (\{\overline{x} \to s(\overline{y})\} :: s :: \overline{s}, h),\ C;\ z := \mathbf{ret}\rangle}$$

$$\textsc{Return}\quad \frac{}{\Sigma \vdash \langle (s :: s' :: \overline{s}, h),\ x := \mathbf{ret}\rangle \longrightarrow \Sigma \vdash \langle (s'[x \to s(\mathtt{res})] :: \overline{s}, h),\ \mathbf{skip}\rangle}$$

$$\textsc{Alloc}\quad \frac{l \notin \operatorname{dom} h}{\Sigma \vdash \langle (s :: \overline{s}, h),\ x := \mathbf{alloc}\rangle \longrightarrow \Sigma \vdash \langle (s[x \to l] :: \overline{s}, h[l \to n]),\ \mathbf{skip}\rangle}$$

$$\textsc{Dealloc}\quad \frac{(s(x) \to n) \in h}{\Sigma \vdash \langle (s :: \overline{s}, h),\ \mathbf{dealloc}(x)\rangle \longrightarrow \Sigma \vdash \langle (s :: \overline{s}, h \setminus \{s(x) \to n\}),\ \mathbf{skip}\rangle}$$

$$\textsc{Dealloc-Fail}\quad \frac{(s(x) \to n) \notin h}{\Sigma \vdash \langle (s :: \overline{s}, h),\ \mathbf{dealloc}(x)\rangle \longrightarrow \mathbf{fail}}$$

$$\textsc{Return-End}\quad \frac{}{\Sigma \vdash \langle (s, h),\ \mathbf{skip}\rangle \longrightarrow s(\mathtt{res})}$$

$$\textsc{AllocedTrue}\quad \frac{s(x) \in \operatorname{dom} h}{\Sigma \vdash \langle (s :: \overline{s}, h),\ \mathbf{alloced}(x)\rangle \longrightarrow \Sigma \vdash \langle (s :: \overline{s}, h),\ \mathbf{skip}\rangle}$$

$$\textsc{AllocedFalse}\quad \frac{s(x) \notin \operatorname{dom} h}{\Sigma \vdash \langle (s :: \overline{s}, h),\ \mathbf{alloced}(x)\rangle \longrightarrow \Sigma \vdash \langle (s :: \overline{s}, h),\ \mathbf{trap}\rangle}$$

$$\textsc{AssertTrue}\quad \frac{[\![B]\!]_s = \mathtt{true}}{\Sigma \vdash \langle (s :: \overline{s}, h),\ \mathbf{assert}(B)\rangle \longrightarrow \Sigma \vdash \langle (s :: \overline{s}, h),\ \mathbf{skip}\rangle}$$

$$\textsc{AssertFalse}\quad \frac{[\![B]\!]_s = \mathtt{false}}{\Sigma \vdash \langle (s :: \overline{s}, h),\ \mathbf{assert}(B)\rangle \longrightarrow \Sigma \vdash \langle (s :: \overline{s}, h),\ \mathbf{trap}\rangle}$$

Figure 6.2: Small-step operational semantics of our imperative language.

6.2 Programming language

The syntax of our formal programming language is defined in Figure 6.1 and its operational semantics are described in Figure 6.2. We write $[\![E]\!]_s$ to indicate the value of an expression E evaluated under a store s, assuming standard non-negative integer expression evaluation. Both the syntax and the semantics of the programming language are very similar to those of the language defined in Section 4.4.

In addition to standard imperative language constructs such as heap lookup, mutation, allocation and deallocation, the language provides two assertion commands. The first is **assert**(b), which asserts that the boolean expression b holds and the other is **alloced**(l), which asserts that the memory address l is allocated. Both commands evaluate to **skip** if the assertion succeeds and to **trap** otherwise. Trapping (as opposed to failing) is a clean way of indicating an abnormality: our runtime checks trap whenever they discover a bug in the context.

The evaluation operator $\longrightarrow$ relates successive program states $\Sigma \vdash \langle(\overline{s}, h),\ C\rangle$, where Σ is a map from routine names to routine definitions, $\overline{s}$ is the stack, h is the heap and C is the program under execution. The stack is a list of stores s, each of which is a partial function from *Vars* to $\mathbb{N}^+$. The heap is a partial function from memory locations in $\mathbb{N}^+$ to values in $\mathbb{N}^+$. Execution starts from an empty store and heap, and results in an outcome o, which is either an integer n, the failure outcome **fail**, the trapped outcome **trap**, or $\lightning$ if the program diverges.

We assume our programs are well-formed. That is, they never try to use a variable that was not defined earlier in the code, and they do not refer to undefined routines. From Figure 6.2 we can see that programs only fail when they try to read, write or deallocate an unallocated memory location. Absence of failure according to this definition implies absence of failure of verified assertions, because we can always modify a program to perform an illegal memory access when an assertion that was statically verified to hold, does not hold at runtime.

6.3 Separation logic assertions

We define separation logic triples of the form $\Gamma \vdash \{P\}\ C\ \{Q\}$ and $\Delta \vDash \{P\}\ C\ \{Q\}$. Triples of the first form mean that the *static verifier* asserts, given the partial function Γ mapping routine names to contracts, that if P holds then C will not fail and Q will hold after executing C. The Γ corresponds to the *prototypes* of

$s, h \vDash \top \rightsquigarrow s$	always
$s, h \vDash B \rightsquigarrow s$	iff $[\![B]\!]_s = \texttt{true}$
$s, h \vDash E \mapsto ?x \rightsquigarrow s'$	iff $h = \{[\![E]\!]_s \to n\}$ and $s' = s[x \to n]$
$s, h \vDash E \mapsto x \rightsquigarrow s$	iff $s, h \vDash E \mapsto ?y \rightsquigarrow s'$ and $s'(y) = s(x)$
$s, h \vDash p(E, ?z) \rightsquigarrow s'$	iff **pred** $p(x, y) = P$ and $\exists v. \{x \to [\![E]\!]_s\}, h \vDash P \rightsquigarrow s''$ and $s' = s[z \to s'(y)]$
$s, h \vDash p(E, x) \rightsquigarrow s$	iff $s, h \vDash p(E, ?y) \rightsquigarrow s'$ and $s'(y) = s(x)$
$s, h \vDash B \ ? \ P : P' \rightsquigarrow s'$	iff if $s, h \vDash B$ then $s, h \vDash P \rightsquigarrow s'$ else $s, h \vDash P' \rightsquigarrow s'$
$s, h \vDash y := E \rightsquigarrow s'$	iff $s' = s[y \to [\![E]\!]_s]$
$s, h \vDash P * P' \rightsquigarrow s''$	iff $\exists h_0, h_1.\ h_0 \perp h_1$ and $h_0 \cup h_1 = h$ and $s, h_0 \vDash P \rightsquigarrow s'$ and $s', h_1 \vDash P' \rightsquigarrow s''$
$s, h \vDash P \vee P' \rightsquigarrow s$	iff $s, h \vDash P$ or $s, h \vDash P'$
$s, h \vDash P \wedge P' \rightsquigarrow s''$	iff $s, h \vDash P \rightsquigarrow s'$ and $s', h \vDash P' \rightsquigarrow s''$
$s, h \vDash \neg P \rightsquigarrow s$	iff $\neg\exists s'.\ s, h \vDash P \rightsquigarrow s'$
$s, h \vDash P$	iff $\exists s'.\ s, h \vDash P \rightsquigarrow s'$

Figure 6.3: Formal semantics of our assertion language.

the functions of the context (including their contracts). Triples of the latter form mean that if P holds in some state (stack and heap), then *executing* C *under the context* Δ won't fail and Q will hold in the resulting state. The Δ corresponds to a concrete context, in the form of a map from routine names to routine definitions. We are only concerned with partial correctness, so C is allowed to diverge.

Because the verifier is sound, $\Gamma \vdash \{P\}\ C\ \{Q\}$ implies $\Delta \vDash \{P\}\ C\ \{Q\}$, under the critical condition that *the routines of* Δ *uphold the contracts defined in* Γ. The essence of our formalization is to show that our program transformation will allow us to discard this critical condition.

$\mathit{fixed}_x(x = E)$	always
$\mathit{fixed}_x(B \ ? \ A : A')$	iff $\mathit{fixed}_x(A)$ and $\mathit{fixed}_x(A')$
$\mathit{fixed}_x(A * A')$	iff $\mathit{fixed}_x(A)$ xor $\mathit{fixed}_x(A')$
$\mathit{precise}(\mathbf{pred}\ p(x, y) = A)$	iff $\mathit{fixed}_y(A)$

Figure 6.4: Definition of precise predicates.

Assertions P and Q are defined as follows.

$$P, Q ::= B \mid B \ ? \ P : Q \mid E \mapsto ?x \mid p(E, ?x) \mid y := E \mid P * Q \mid \\ P \wedge Q \mid P \vee Q \mid \neg P \mid \top$$

Boolean expressions B and integer expressions E are defined as in Figure 6.1, p refers to a user-defined predicate, and $?x$ introduces a logic variable x. The first parameter of a user-defined predicate is an input parameter and the second is an output parameter. The assignment assertion $y := E$ is used for binding a value to the output parameter of a predicate within a predicate definition. Having a separate syntactic construct for this purpose simplifies the definition of the program transformations.

The formal semantics of the assertion language are defined in Figure 6.3. In this figure, a judgment $s, h \vDash P \rightsquigarrow s'$ means that the assertion P holds under store s and heap h and binds new logic variables (using the $E \mapsto ?x$ and $p(e, ?x)$ constructs) to create the updated store s'.

We always assume assertions are well-formed. That is, (1) all logic variables are distinct from program variables, (2) assertions never refer to undefined program or logic variables and (3) assertions never re-assign logic variables. These properties can always be achieved by a renaming of logic variables.

6.3.1 Contract assertion language

Although assertions in our meta-theory range over the full language defined above, routine and predicate contracts come from a more restricted language of *precise* assertions.

$$A ::= B \mid B \ ? \ A : A \mid E \mapsto ?x \mid p(E, ?x) \mid y := E \mid A * A$$

In particular, routine and predicate assertions do not include standard conjunction, disjunction, negation nor top. Furthermore, we require user-defined predicates to constrain their output parameter to a single value (using the $y := E$ construct), exactly once on each possible execution path of their body, as defined in Figure 6.4. These requirements make existential quantification *constructive*: assertions indicate how each variable can be assigned a value, thereby avoiding an exhaustive search which would entail an enormous runtime performance cost.

This contract assertion language corresponds exactly to the assertion language defined in Section 4.4. As we already argued there, excluding disjunction, negation and non-separating conjunction between spatial predicates might seem to limit the expressiveness of the contract assertion language, but this language subset corresponds exactly to the assertion languages supported by VeriFast [61], Smallfoot [19] and other separation logic-based program verifiers [38, 25]. Extensive experience with these tools has shown that this subset is sufficiently expressive for all practical purposes [100].

In the rest of the text, we will consistently use symbols P and Q for meta-level assertions and symbol A for contract assertions.

6.4 Assertion production

We now define a function $\mathsf{prod}(A)$ from assertions to commands that models the runtime *production* of A (see Section 5.4.2). This function by itself would be sufficient for generating safe and precise checks for verified modules that do not perform any outcalls. We will use this function as a building block for the definition of a transformation with support for outcalls, in Section 6.5.

The code generated by $\mathsf{prod}(A)$ assumes there is a program variable `fp` containing a set of memory locations that represents the hardened module's current footprint. Since memory locations in our programming language are non-negative integers, a Gödel encoding could be used to store this footprint. The generated code will (1) trap if A does not hold or if its footprint would overlap with the footprint in `fp`, (2) create a program variable x for each logic variable $?x$ in A and (3) add the assertion's footprint to `fp`. The function is defined as follows.

$$\mathsf{prod}(y := E) = (y := E)$$

$$\mathsf{prod}(B) = \mathbf{assert}(B)$$

$$\mathsf{prod}(B \ ?\ A_1 : A_2) = \mathbf{if}\ B\ \mathbf{then}\ \mathsf{prod}(A_1)\ \mathbf{else}\ \mathsf{prod}(A_2)$$

$$\mathsf{prod}(E \mapsto ?x) = \begin{cases} x := E;\ x := \mathtt{in}(x, \mathtt{fp}); \\ \mathbf{assert}(x = 0);\ x := E;\ \mathbf{alloced}(x); \\ \mathtt{fp} := \mathsf{add}(x, \mathtt{fp});\ x := [x] \end{cases}$$

where $\mathtt{fp}$ is the program variable that stores the footprint, $\mathtt{in}(x, y)$ returns 1 if x is in the list represented by y and 0 otherwise, and $\mathsf{add}(x, y)$ adds x to the list represented by y.

$$\mathsf{prod}(p(E, ?x)) = (x := E;\ \{\mathtt{fp}, x\} := \mathit{prod}_p(\mathtt{fp}, x))$$

where prod_p implements the production part of the *predicate checking routine* for p, defined as **routine** $\mathit{prod}_p(\mathtt{fp}, x) = \mathsf{prod}(A)$; $\mathtt{res} := \{\mathtt{fp}, y\}$ with A the body of **pred** $p(x, y)$. The $\{\mathtt{fp}, x\}$ is syntactic sugar for a tuple consisting of $\mathtt{fp}$ and x.

$$\mathsf{prod}(A_1 * A_2) = \mathsf{prod}(A_1);\ \mathsf{prod}(A_2)$$

Before proving the correctness of prod, we first define a number of auxiliary definitions and lemmas.

Definition 6.4.1 (Partial function subset)**.** *We say a partial function $f : X \to Y$ is a subset of a (partial or total) function $f' : X \to Y$, written $f \sqsubseteq f'$ iff $\forall x \in \mathit{dom}\, f : f(x) = f'(x)$.*

Lemma 6.4.1. *For any s, s', h, h' and A, such that $h \sqsubseteq h'$, we have that $\langle (s, h),\ \mathsf{prod}(A)\rangle \longrightarrow \langle (s', h),\ \textbf{\textit{skip}}\rangle \implies \langle (s, h'),\ \mathsf{prod}(A)\rangle \longrightarrow \langle (s', h'),\ \textbf{\textit{skip}}\rangle$.*

Proof. Follows from the definition of prod. Any memory location allocated in h is also allocated in h' and has the same value. □

Lemma 6.4.2. *For any s, s', h and A such that $\langle (s, h),\ \mathsf{prod}(A)\rangle \longrightarrow \langle (s', h),\ \textbf{\textit{skip}}\rangle$, we have $s \sqsubseteq s'$.*

Proof. The key property to check is that $\mathsf{prod}(A)$ never changes any program variable $x \in \mathrm{dom}\, s$, but only creates new variables that do not exist yet. This property follows from the fact that $\mathsf{prod}(A)$ only assigns a variable x when either

$$\mathsf{concrete}(E \mapsto ?x) = E \mapsto x \qquad \mathsf{concrete}(E \mapsto x) = E \mapsto x$$

$$\mathsf{concrete}(p(E, ?x)) = p(E, x) \qquad \mathsf{concrete}(p(E, x)) = p(E, x)$$

$$\mathsf{concrete}(B) = B \qquad \mathsf{concrete}(x := E) = x = E$$

$$\mathsf{concrete}(b \ ? \ a : a') = b \ ? \ \mathsf{concrete}(a) : \mathsf{concrete}(a')$$

$$\mathsf{concrete}(a * a') = \mathsf{concrete}(a) * \mathsf{concrete}(a')$$

Figure 6.5: Definition of assertion concretization.

(1) there is a logic variable $?x$ in A or (2) when A contains an output variable assignment $y := E$. Our assumption of well-formedness ensures that all logic variables names are distinct from program variable names, and that assertions never re-assign logic variables. Thus, no pre-existing program variables will be modified in the former case. The latter case only occurs when assigning an output parameter from inside a predicate checking routine $prod_p$, and here our preciseness assumption ensures that every output parameter is assigned exactly once on every execution path. □

Definition 6.4.2 (Assertion footprint)**.** *The footprint of an assertion A under a store s and heap h' is defined as*

$$\mathsf{fp}_{s,h'}(A) = dom\, h$$

where h is the smallest heap such that $h \sqsubseteq h'$ and $s, h \vDash A \rightsquigarrow s'$ for some s'.

We now prove our first important property about $\mathsf{prod}(A)$, which describes how this function behaves when A is valid. More precisely, Lemma 6.4.3 says that if an assertion A holds under some store s and heap h and extends this store with new logic variables to form the new store s' and the footprint of A does not overlap with the footprint in `fp`, then $\mathsf{prod}(A)$ executed from that store s and heap h will evaluate to **skip** and the resulting store will be equal to s'. In this new store s', the footprint `fp` will have been updated with the footprint of A and $\mathsf{concrete}(A)$ will hold. The formal definition of $\mathsf{concrete}(A)$ is given in Figure 6.5, but intuitively it means that all occurrences of logic variables $?x$ in A are replaced with corresponding program variables x, which implies that $\mathsf{prod}(A)$ has introduced a program variable x for each logic variable $?x$ and has assigned it the same value.

Lemma 6.4.3. *For any s, h, s' and well-formed A, we have*

$$s, h \vDash A \rightsquigarrow s' \wedge \mathsf{fp}_{s,h}(A) \perp s(\mathtt{fp})$$

$$\Downarrow$$

$$[\langle (s,h),\ \mathsf{prod}(A)\rangle \longrightarrow^* \langle (s',h),\ \textbf{\textit{skip}}\rangle\ \wedge\ s', h \vDash \mathsf{concrete}(A)\ \wedge$$
$$s'(\mathtt{fp}) = s(\mathtt{fp}) \cup \mathsf{fp}_{s',h}(A)] \vee \langle (s,h),\ \mathsf{prod}(A)\rangle \longrightarrow^* \lightning$$

Proof. The proof goes by induction on the execution of $\mathsf{prod}(A)$. If $\mathsf{prod}(A)$ diverges, the lemma holds immediately. If $\mathsf{prod}(A)$ does not diverge, we have the following case analysis on A:

- When A is a boolean expression B, we have $\mathsf{prod}(B) = \mathbf{assert}(B)$. If $s, h \vDash B$, then $[\![B]\!]_s = \mathtt{true}$ and hence $\mathsf{prod}(B)$ evaluates to **skip**. Since there are no free logic variables in B, we have $\mathsf{concrete}(B) = B$ which still holds in store $s' = s$ and since $\mathsf{fp}_{s,h}(B) = \mathsf{fp}_{s',h}(B) = \emptyset$, the runtime footprint $s'(\mathtt{fp})$ is still up to date.
- When $A = B\ ?\ A_1 : A_2$, we have $\mathsf{prod}(A) = \mathbf{if}\ B\ \mathbf{then}\ \mathsf{prod}(A_1)\ \mathbf{else}\ \mathsf{prod}(A_2)$. If $[\![B]\!]_s = \mathtt{true}$, then $s, h \vDash A_1 \rightsquigarrow s'$ holds and we have $\mathsf{fp}_{s',h}(A) = \mathsf{fp}_{s',h}(A_1)$, which we can combine with the induction hypothesis for A_1 to see that

  $$\langle (s,h),\ \mathsf{prod}(A_1)\rangle \longrightarrow^* \langle (s',h),\ \mathbf{skip}\rangle\ \wedge$$
  $$s', h \vDash \mathsf{concrete}(A_1) \wedge s'(\mathtt{fp}) = s(\mathtt{fp}) \cup \mathsf{fp}_{s',h}(A_1)$$

  Still assuming $[\![B]\!]_s = \mathtt{true}$, we know that $\mathsf{prod}(A)$ evaluates to $\mathsf{prod}(A_1)$ and $s', h \vDash \mathsf{concrete}(A) \rightsquigarrow s' \Leftrightarrow s', h \vDash \mathsf{concrete}(A_1) \rightsquigarrow s'$ and hence the conclusion follows. The case for $[\![B]\!]_s = \mathtt{false}$ is symmetrical.
- When $A = E \mapsto ?x$, we have

  $$\begin{aligned}\mathsf{prod}(A) = (&x := E;\ x := \mathsf{in}(x, \mathtt{fp});\\ &\mathbf{assert}(x = 0);\ x := E;\ \mathbf{alloced}(x);\\ &\mathtt{fp} := \mathsf{add}(x, \mathtt{fp});\ x := [x])\end{aligned}$$

  Given that $s, h \vDash A \rightsquigarrow s'$, we know $[\![E]\!]_s = l$ for some memory location l, with $h = \{l \to m\}$ and thus $s' = s[x \to m]$. We know that $\mathsf{fp}_{s,h}(A) = \{l\}$ and from the given that $\mathsf{fp}_{s,h}(A) \perp s(\mathtt{fp})$, we know that $\mathsf{prod}(A)$ will evaluate to $\mathbf{alloced}(x);\ \mathtt{fp} := \mathsf{add}(x, \mathtt{fp});\ x := [x]$, where x initially has the value l. Since $l \in \operatorname{dom} h$, the alloced check will pass, l will be added to $\mathtt{fp}$ and x will be assigned the value m. Since no program variable other

than x was assigned, the updated store will be equal to s' and $\mathsf{concrete}(A)$ will hold. Finally, because of our assumption of assertion well-formedness, we have $\mathsf{fp}_{s',h}(A) = \mathsf{fp}_{s,h}(A) = \{l\}$ and hence the conclusion follows.

- For $A = (y := E)$, we have $\mathsf{prod}(A) = (y := E)$. Given that $s, h \vDash A \rightsquigarrow s'$, we know $[\![E]\!]_s = n$ for some n, and $s' = s[y \to n]$. We can see that $\mathsf{concrete}(A) = (y = e)$ will hold in the updated store s', because y is the only program variable assigned by $\mathsf{prod}(A)$, and it will be assigned the value n. Finally, since $\mathsf{fp}_{s,h}(A) = \mathsf{fp}_{s',h}(A) = \emptyset$, the runtime footprint $s'(\mathtt{fp})$ is still up to date.

- For $A = p(E, ?x)$, we have $\mathsf{prod}(A) = (x := E; \{\mathtt{fp}, x\} := r(\mathtt{fp}, x))$, where the body of r is $(\mathsf{prod}(A'); \ \mathtt{res} := \{\mathtt{fp}, y\})$ with A' the body of predicate $p(x, y)$. Given that $s, h \vDash A \rightsquigarrow s'$, we know $s' = s[x \to m]$ for some m. We also know that $\mathsf{fp}_{s,h}(A) = \mathsf{fp}_{s_{pred},h}(A')$ and that $s_{pred}, h \vDash A' \rightsquigarrow s'_{pred}$ holds with $s_{pred} = \{x \to [\![E]\!]_s\}$ and $s'_{pred}(y) = m$. Combining this with our induction hypothesis, we have

$$\langle (s_{pred}, h),\ \mathsf{prod}(A') \rangle \longrightarrow^* \langle (s'_{pred}, h),\ \mathbf{skip} \rangle \wedge$$

$$s'_{pred}, h \vDash \mathsf{concrete}(A') \wedge s'_{pred}(\mathtt{fp}) = s_{pred}(\mathtt{fp}) \cup \mathsf{fp}_{s'_{pred},h}(A')$$

  Hence $\mathsf{prod}(A)$ will evaluate to **skip**. Since routine r returns the updated footprint and the value of y, which is assigned to x by $\mathsf{prod}(A)$, we see that $\mathsf{concrete}(A) = p(E, x)$ holds in the updated store. Furthermore, since x is the only program variable assigned, this updated store is equal to s'. Finally, since $\mathsf{fp}_{s',h}(A) = \mathsf{fp}_{s'_{pred},h}(A')$ the footprint $s'(\mathtt{fp})$ is updated correctly as well and hence the conclusion holds.

- When $A = (A_1 \ast A_2)$, we have $\mathsf{prod}(A) = \ \mathsf{prod}(A_1);\ \mathsf{prod}(A_2)$. Given that $s, h \vDash A \rightsquigarrow s''$, we know $s, h_1 \vDash A_1 \rightsquigarrow s'$ and $s', h_2 \vDash A_2 \rightsquigarrow s''$ for some h_1 and h_2 such that $h_1 \perp h_2$ and $h_1 \cup h_2 = h$. Using the induction hypothesis for A_1 and A_2 gives us (1)

$$s, h_1 \vDash A_1 \rightsquigarrow s' \wedge \mathsf{fp}_{s,h_1}(A_1) \cap s(\mathtt{fp}) = \emptyset$$

$$\Downarrow$$

$$\langle (s, h_1),\ \mathsf{prod}(A_1) \rangle \longrightarrow^* \langle (s', h_1),\ \mathbf{skip} \rangle \ \wedge$$

$$s', h_1 \vDash \mathsf{concrete}(A_1) \wedge s'(\mathtt{fp}) = s(\mathtt{fp}) \cup \mathsf{fp}_{s',h_1}(A_1)$$

and (2)

$$s', h_2 \vDash A_2 \rightsquigarrow s'' \wedge \mathsf{fp}_{s',h_2}(A_2) \cap s'(\mathtt{fp}) = \emptyset$$

$$\Downarrow$$

$$\langle (s', h_2),\ \mathsf{prod}(A_2)\rangle \longrightarrow^* \langle (s'', h_2),\ \mathbf{skip}\rangle \ \wedge$$

$$s'', h_2 \vDash \mathsf{concrete}(A_2) \wedge s''(\mathtt{fp}) = s'(\mathtt{fp}) \cup \mathsf{fp}_{s'',h_2}(A_2)$$

From $h_1 \perp h_2$ we know that $\mathsf{fp}_{s,h_1}(A_1) \cap \mathsf{fp}_{s',h_2}(A_2) = \emptyset$. Combining this with $\mathsf{fp}_{s,h}(A) = \mathsf{fp}_{s,h_1}(A_1) \cup \mathsf{fp}_{s',h_2}(A_2)$ and $\mathsf{fp}_{s,h}(A) \cap s(\mathtt{fp}) = \emptyset$, leads to $\mathsf{fp}_{s,h_1}(A_1) \cap s(\mathtt{fp}) = \emptyset$ and $\mathsf{fp}_{s',h_2}(A_2) \cap s'(\mathtt{fp}) = \emptyset$. Hence the premise and thus conclusion of (1) and (2) hold. We can apply Lemma 6.4.1 to see $\langle (s, h),\ \mathsf{prod}(A_1)\rangle \longrightarrow^* \langle (s', h),\ \mathbf{skip}\rangle$ and $\langle (s', h),\ \mathsf{prod}(A_2)\rangle \longrightarrow^* \langle (s'', h),\ \mathbf{skip}\rangle$ and hence $\langle (s, h),\ \mathsf{prod}(A_1); \mathsf{prod}(A_2)\rangle \longrightarrow^* \langle (s'', h),\ \mathbf{skip}\rangle$. Lemma 6.4.2 shows us that $s' \sqsubseteq s''$ and because we have $h_1 \perp h_2$ and $h_1 \cup h_2 = h$, we also have $s'', h \vDash \mathsf{concrete}(A_1) * \mathsf{concrete}(A_2)$. And since $s''(\mathtt{fp}) = s'(\mathtt{fp}) \cup \mathsf{fp}_{s'',h_2}(A_2)$ and $s'(\mathtt{fp}) = s(\mathtt{fp}) \cup \mathsf{fp}_{s',h_1}(A_1)$, we have $s''(\mathtt{fp}) = s(\mathtt{fp}) \cup \mathsf{fp}_{s',h_1}(A_1) \cup \mathsf{fp}_{s'',h_2}(A_2)$. Hence, the conclusion follows.

□

We now reformulate this lemma as a separation logic triple.

Lemma 6.4.4. *For well-formed assertions A, A' and $A * A'$, we have*

$$\Delta \vDash \{A * A' \wedge \mathtt{fp} \perp \mathsf{fp}(A)\}$$

$$\mathsf{prod}(A)$$

$$\{\mathsf{concrete}(A) * A' \wedge \mathtt{fp} = \mathtt{fp}_{old} \cup \mathsf{fp}(A)\}$$

where $\mathsf{fp}(A)$ is the footprint of A evaluated in the current state and $\mathtt{fp}_{old}$ refers to the footprint as it was before executing $\mathsf{prod}(A)$.

Proof. Follows directly from Lemma 6.4.3 and the separation logic frame rule. Note that the frame rule is not sound in general for our programming language, because of the **alloced** command: extending the heap can cause **alloced** to **skip** instead of **trap**. The frame rule is however applicable in this specific case, because Lemma 6.4.3 shows $\mathsf{prod}(A)$ does not trap and hence extending the heap will not change the behavior of **alloced**. □

The second important property of prod states that $\mathsf{prod}(A)$ will trap or diverge when the footprint of A overlaps with the footprint in $\mathtt{fp}$.

Lemma 6.4.5. *For any s, h and well-formed A, we have*

$$\Delta \vDash \{\mathtt{fp} \not\perp \mathsf{fp}(A)\}\ \mathsf{prod}(A)\ \{\textbf{\textit{trap}}\}$$

Proof sketch. The proof goes by induction on the execution length of $\mathsf{prod}(A)$. When $\mathsf{prod}(A)$ diverges, the lemma holds immediately. When it does not diverge, we can intuitively see that the lemma holds, because the only assertion that has a non-empty footprint is $E \mapsto ?x$. The footprint of this assertion is the value of E and $\mathsf{prod}(E \mapsto ?x)$ traps when the value of E is already in $\mathtt{fp}$. □

The third and final important property of prod states how $\mathsf{prod}(A)$ behaves when A does *not* hold. More specifically, Lemma 6.4.6 says that if A does not hold in some store s and heap h, and the reason that A does not hold is *not* because h is too large (i.e., defined for more memory locations than the footprint of A), then $\mathsf{prod}(A)$ traps or diverges.

Lemma 6.4.6. *For any well-formed assertion A, we have*

$$\Delta \vDash \{\neg(A * \top)\}\ \mathsf{prod}(A)\ \{\textbf{\textit{trap}}\}$$

Proof. The proof goes by induction on the execution length of $\mathsf{prod}(A)$. If $\mathsf{prod}(A)$ diverges, the lemma holds immediately because our triples only express partial correctness. If $\mathsf{prod}(A)$ does not diverge we have the following case analysis on A:

- For a boolean assertion B, we have $\mathsf{prod}(A) = \mathbf{assert}(B)$. We also have $s, h \vDash \neg(A * \top) \iff s, h \vDash \neg B$, and thus $[\![B]\!]_s = \mathtt{false}$ and hence $\mathbf{assert}(B)$ evaluates to $\mathbf{trap}$.
- For $A = B\ ?\ A_1 : A_2$, we have $\mathsf{prod}(A) = \mathbf{if}\ B\ \mathbf{then}\ \mathsf{prod}(A_1)\ \mathbf{else}\ \mathsf{prod}(A_2)$. Suppose $[\![B]\!]_s = \mathtt{true}$, then from $s, h \vDash \neg(A * \top)$ we have $s, h \vDash \neg(A_1 * \top)$ and we can use the induction hypothesis for A_1 to see that $\{\neg(A_1 * \top)\}\ \mathsf{prod}(A_1)\ \{\mathbf{trap}\}$. The case for $[\![B]\!]_s = \mathtt{false}$ is symmetrical and hence the lemma holds in this case.
- For $A = E \mapsto ?x$, we have

$$\begin{aligned}\mathsf{prod}(A) = (&x := E;\ x := \mathtt{in}(x, \mathtt{fp});\\ &\mathbf{assert}(x = 0);\ x := e;\ \mathbf{alloced}(x);\\ &\mathtt{fp} := \mathsf{add}(x, \mathtt{fp});\ x := [x])\end{aligned}$$

  Since A is well-formed, E will evaluate to some memory location l. Given that $s, h \vDash \neg(A * \top)$, we know $l \notin \operatorname{dom} h$. This means $\mathbf{alloced}(x)$ will trap and hence no matter the outcome of $\mathtt{in}(x, \mathtt{fp})$, $\mathsf{prod}(A)$ will trap.

- For $A = (y := E)$, we have $\mathsf{prod}(A) = (y := E)$. Since A is well-formed, E will evaluate to some value n. From Figure 6.3, we can see that $s, h \vDash \neg(A * \top)$ can never occur and hence we can discard this case.
- For $A = p(E, ?x)$, we have $\mathsf{prod}(A) = (x := E;\ \{\mathtt{fp}, x\} := r(\mathtt{fp}, x))$, where the body of r is $(\mathsf{prod}(A');\ \mathtt{res} := \{\mathtt{fp}, y\})$ with A' the body of predicate $p(x, y)$. Since A is well-formed, we know that E will evaluate to some value n. Given that $s, h \vDash \neg(A * \top)$, we have $s_A, h \vDash \neg(A' * \top)$ with $s_A = \{x \to n\}$. We can use the induction hypothesis to see that $\{\neg(A' * \top)\}\ \mathsf{prod}(A')\ \{\mathbf{trap}\}$ and hence the lemma holds in this case.
- For $A = (A_1 * A_2)$, we have $\mathsf{prod}(A) = \mathsf{prod}(A_1);\ \mathsf{prod}(A_2)$. Given that $s, h \vDash \neg(A * \top)$, we have one of the following three cases.
  - $s, h \vDash \neg(A_1 * \top)$, here we can immediately apply the induction hypothesis to A_1 to see that $\mathsf{prod}(A_1)$ will trap.
  - $s, h \vDash A_1 * \top \rightsquigarrow s' \wedge s', h \vDash \neg(A_2 * \top)$, in this case, if we have $\mathsf{fp}_{s,h}(A_1) \perp s(\mathtt{fp})$, then we can apply Lemma 6.4.3 to see that $\mathsf{prod}(A_1)$ will evaluate to **skip** with a resulting store equal to s'. We can them apply the induction hypothesis to A_2 to see that $\mathsf{prod}(A_2)$ will trap. If however we have $\mathsf{fp}_{s,h}(A_1) \not\perp s(\mathtt{fp})$, then we can apply Lemma 6.4.5 to see that $\mathsf{prod}(A_1)$ will trap.
  - $s, h_1 \vDash A_1 \rightsquigarrow s' \wedge s', h_2 \vDash A_2 \rightsquigarrow s''$ for some h_1 and h_2, with $h_1 \not\perp h_2$. If we have $\mathsf{fp}_{s,h}(A_1) \not\perp s(\mathtt{fp})$, then we can apply Lemma 6.4.5 to see that $\mathsf{prod}(A_1)$ will trap. If we have $\mathsf{fp}_{s,h}(A_1) \perp s(\mathtt{fp})$, we can apply Lemma 6.4.3 to see that $\mathsf{prod}(A_1)$ will evaluate to **skip** with a resulting store equal to s' and a footprint that now includes $\mathsf{fp}_{s,h}(A_1) = \operatorname{dom} h_1$. We can then use the fact that $h_1 \not\perp h_2$ and $\mathsf{fp}_{s',h}(A_2) = \operatorname{dom} h_2$ and Lemma 6.4.5 to see that $\mathsf{prod}(A_2)$ will trap.

□

We now summarize Lemma 6.4.4, Lemma 6.4.5 and Lemma 6.4.6 together in the following theorem.

Theorem 6.4.1. *For well-formed assertions A and A', we have*

$$\Delta \vDash \{\top * A' \wedge \mathsf{fp}(A') \subseteq \mathtt{fp}\}$$

$$\mathsf{prod}(A)$$

$$\{(\top * \mathsf{concrete}(A) * A' \wedge \mathtt{fp} = \mathtt{fp}_{old} \cup \mathsf{fp}(A)) \vee \textbf{\textit{trap}}\}$$

Proof. Suppose first that in the initial state $\top * A$ holds and $\mathtt{fp} \perp \mathsf{fp}(A)$. Because we have $\mathsf{fp}(A') \subseteq \mathtt{fp}$, we know that $\mathsf{fp}(A) \perp \mathsf{fp}(A')$ and thus $\top * A * A' \wedge \mathtt{fp} \perp \mathsf{fp}(A)$ will hold in the initial state. We can then apply the frame rule and Lemma 6.4.4 to see that $\top * \mathsf{concrete}(A) * A' \wedge \mathtt{fp} = \mathtt{fp}_{old} \cup \mathsf{fp}(A)$ will hold after executing $\mathsf{prod}(A)$. Again, in this case applying the frame rule is allowed, because $\mathsf{prod}(A)$ will not trap. On the other hand, if $\top * A$ does not hold initially, we can apply Lemma 6.4.6 to see that $\mathsf{prod}(A)$ will trap. Similarly, if $\mathtt{fp} \not\perp \mathsf{fp}(A)$, we can apply Lemma 6.4.5 to see that $\mathsf{prod}(A)$ will trap. □

6.5 Assertion consumption

We now define $\mathsf{cons}(A)$, a function that models the *consumption* of an assertion, which is required for safely performing outcalls from a hardened module to routines of the context that might not uphold their contract. As explained in Section 5.4.2, this function needs to be called right before making an outcall, to consume (remove) the footprint of the context function's precondition from the current footprint $\mathtt{fp}$. The structure of $\mathsf{cons}(A)$ is identical to that of $\mathsf{prod}(A)$, but it has to *remove* A's footprint instead of adding it. Furthermore, there is no need for $\mathsf{cons}(A)$ to check that A actually holds, because the static verifier already ensured this when checking the verified module. Hence, we do not need to use the **assert** and **alloced** commands in the definition of cons. The function is defined as follows.

$$\mathsf{cons}(y := E) = (y := E)$$

$$\mathsf{cons}(B) = \mathbf{skip}$$

$$\mathsf{cons}(B \ ? \ A_1 : A_2) = \mathbf{if}\ B\ \mathbf{then}\ \mathsf{cons}(A_1)\ \mathbf{else}\ \mathsf{cons}(A_2)$$

$$\mathsf{cons}(E \mapsto ?x) = x := E;\ \mathtt{fp} := \mathtt{rem}(x, \mathtt{fp});\ x := [x]$$

where $\mathtt{rem}(x, y)$ removes x from the list represented by y.

$$\mathsf{cons}(p(E, ?x)) = (x := E;\ \{\mathtt{fp}, x\} := \mathit{cons}_p(\mathtt{fp}, x))$$

where cons_p implements the consumption part of the *predicate checking routine* for p, defined as $\mathbf{routine}\ \mathit{cons}_p(\mathtt{fp}, x) = \mathsf{cons}(A);\ \mathtt{res} := \{\mathtt{fp}, y\}$ with A the body of predicate $\mathbf{pred}\ p(x, y)$

$$\mathsf{cons}(A_1 * A_2) = \mathsf{cons}(A_1);\ \mathsf{cons}(A_2)$$

The following lemma indicates that the net effect of $\mathsf{cons}(A)$ is the concretization of A and the removal of its footprint from $\mathtt{fp}$

Lemma 6.5.1. *For the function* cons *defined above and well-formed assertions* A *and* A'*, we have*

$$\Delta \vDash \{A * A'\}$$

$$\mathsf{cons}(A)$$

$$\{\mathsf{concrete}(A) * A' \wedge \mathtt{fp} = \mathtt{fp}_{old} \setminus \mathsf{fp}(A)\}$$

Proof. Follows directly from the definition of cons. □

We now define a function $\mathsf{harness}_\Gamma(r)$ that can generate outcall stubs for routines $r(\overline{x})$ of the context. This function takes a mapping Γ from routines to contracts, corresponding to the prototypes of functions defined in the context. For $\Gamma(r) = (A_{pre}, A_{post})$, $\mathsf{harness}_\Gamma(r)$ is defined as follows.

$$\begin{array}{l}\textbf{routine } \mathit{stub}_r(\mathtt{fp},\ \overline{x}) = \\ \quad \mathsf{cons}(A_{pre});\ \mathtt{s} := \mathsf{snap}(\mathtt{fp}); \\ \quad \mathtt{res} := r(\overline{x}); \\ \quad \mathtt{s}' := \mathsf{snap}(\mathtt{fp});\ \textbf{assert}(\mathtt{s} = \mathtt{s}'); \\ \quad \mathsf{prod}(A_{post});\ \mathtt{res} := \{\mathtt{fp},\ \mathtt{res}\}\end{array}$$

where we assume all introduced variables are fresh and $\mathsf{snap}(fp)$ returns a snapshot of the contents of the footprint *fp*. This corresponds to calculating the cryptographic hash over the footprint description, as described in Section 5.4.2.

Finally, we can define the full transformation function $[C]_{\Gamma,A}$, using a helper function $[C]'_\Gamma$, as follows:

$$[x := r(\overline{x})]'_\Gamma = \{fp,\ x\} := \mathit{stub}_r(fp, \overline{x})$$

where r is a routine of the context and stub_r is the name of the outcall stub routine generated by $\mathsf{harness}_\Gamma(r)$

$$[C_1;\ C_2]'_\Gamma = [C_1]'_\Gamma;\ [C_2]'_\Gamma$$

$$[\textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2]'_\Gamma = \textbf{if } B \textbf{ then } [C_1]'_\Gamma \textbf{ else } [C_2]'_\Gamma$$

$$[x := \textbf{alloc}]'_\Gamma = x := \textbf{alloc};\ \mathtt{fp} := \mathsf{add}(x, \mathtt{fp})$$

$$[\textbf{dealloc}(x)]'_\Gamma = \textbf{dealloc}(x);\ \mathtt{fp} := \mathtt{rem}(x, \mathtt{fp})$$

$$[C]'_\Gamma = C \qquad \text{(for all other forms of } C\text{)}$$

The full transformation function $[C]_{\Gamma,A}$ then is

$$[C]_{\Gamma,A} = (\mathtt{fp} := \emptyset;\ \mathsf{prod}(A);\ [C]'_\Gamma)$$

One particularity to note is that our formal transformation $[C]'_\Gamma$ has cases for **alloc** and **dealloc**, which were not mentioned in the informal discussion in Chapter 5. This is because in practice our hardened modules allocate and deallocate memory using the standard library functions `malloc()` and `free()`, which are assumed to be part of the untrusted context. Hence in practice the updating of the footprint as shown in the definition of $[C]'_\Gamma$ is performed automatically as part of the standard transformation of outcalls, assuming that these functions are fitted with a proper contract. Transformation rules similar to those for allocation and deallocation in $[C]'_\Gamma$ could be implemented in our translator as an optimization for trusted implementations of `malloc()` and `free()`.

6.6 Safety and precision

We now come to the two crucial properties our transformation must have: safety and precision. We first need a new definition and lemma before we can formally state these main theorems.

Definition 6.6.1 (No-fail)**.** *The function* $\mathsf{nofail}(\Delta)$ *returns a non-failing variant of the program* Δ*. That is,* $\mathsf{nofail}(\Delta)$ *never performs an illegal memory access.*

How nofail could work is of no importance for our formalization, but one can see that the **alloced** command could be used to check each memory location before accessing it, thereby preventing failure.

Lemma 6.6.1. *For a command* C *and well-formed assertions* A_{pre} *and* A_{post} *such that* $\Gamma \vdash \{A_{pre}\}\ C\ \{A_{post}\}$*, we have*

$$\forall \Delta.\ \mathsf{nofail}(\Delta) \vDash \{\top \ast A_{pre} \wedge\ \mathtt{fp} = \mathsf{fp}(A_{pre})\}$$

$$[C]'_\Gamma$$

$$\{(\top \ast A_{post} \wedge \mathtt{fp} = \mathsf{fp}(A_{post})) \vee \boldsymbol{trap}\}$$

Proof. The proof goes by induction on C. Since $\Gamma \vdash \{A_{pre}\}\ C\ \{A_{post}\}$ implies $\Gamma \vDash \{A_{pre}\}\ C\ \{A_{post}\}$ for all commands except routine calls, we know that commands other than routine calls won't fail. We also know that the only commands that change the domain of the heap (i.e., the symbolic footprint)

are allocation, deallocation and routine calls. Hence for all other commands, we can immediately see that `fp` still correctly contains the same footprint after the command has been executed. For $x := \textbf{alloc}$ and $\textbf{dealloc}(x)$, the symbolic footprint is extended respectively shrunk with the value of x, but the transformation $[C]'_\Gamma$ modifies the runtime footprint `fp` accordingly. For a sequence of commands $C_1;\ C_2$, the lemma follows immediately from the rule of composition and applying the induction hypothesis to C_1 and C_2.

Hence the only case left to prove is the case for routine calls, for which the transformation $[C]'_{\Gamma,A_{pre}}$ is $(\{\texttt{fp}, x\} := h_r(\texttt{fp}, \overline{y}))$, where h_r is the name of the stub harness as defined above. Because (1) we assume $\mathsf{nofail}(\Delta)$, (2) $\mathsf{snap}(\texttt{fp})$ never reads outside the current footprint and (3) Lemma 6.5.1 and Theorem 6.4.1, we can see that none of the harness' commands will fail. Suppose $\Gamma(r) = (A'_{pre}, A'_{post})$. At entry to the harness stub, the store and heap are exactly as they would be when calling the original function r (except for `fp`, which we assume to be distinct from all program or logic variables), and hence, according to Lemma 6.5.1, $\mathsf{cons}(A'_{pre})$ will remove the footprint of A'_{pre} from `fp` and create a corresponding program variable for each logic variable in A'_{pre}. The harness then takes a snapshot of the remaining footprint, calls the original routine and checks that the footprint hasn't changed. If this check succeeds, we know all our original assertions from before the function call still hold, except potentially those involving the footprint of A'_{pre}. From Theorem 6.4.1, we can see that $\mathsf{prod}(A'_{post})$ either traps or assures that $\mathsf{concrete}(A'_{post})$ (and hence A'_{post}) holds and adds $\mathsf{fp}(A'_{post})$ to `fp`, such that it now corresponds to the footprint of A_{post}. □

This lemma leads to our safety theorem, which states that if the context does not fail, but does not necessarily uphold its contracts either, then the hardened module will never fail.

Theorem 6.6.1 (Safety)**.** *For any command C, environment Γ, well-formed assertions A_{pre} and A_{post} such that $\Gamma \vdash \{A_{pre}\}\ C\ \{A_{post}\}$, and an arbitrary context Δ, we have $\mathsf{nofail}(\Delta) \vDash \{\top\}\ [C]_{\Gamma,A_{pre}}\ \{\top\}$.*

Proof. The definition of $[C]_{\Gamma,A_{pre}}$ is $\texttt{fp} := \emptyset;\ \mathsf{prod}(A_{pre});\ [C]'_\Gamma$. Hence we can first use Theorem 6.4.1 and the fact that $s, h \vDash \mathsf{concrete}(A_{pre}) \rightsquigarrow s \implies s, h \vDash A_{pre}$ to see that after $\mathsf{prod}(A_{pre})$ we have either trapped or we know that $\top * A_{pre}$ holds and $\texttt{fp} = \mathsf{fp}(A_{pre})$. The theorem then follows by applying Lemma 6.6.1. □

Finally, our precision theorem states that our transformations do not change the expected behavior of the hardened module when the context upholds its contracts.

Theorem 6.6.2 (Precision)**.** *For any command C and well-formed assertions A_{pre} and A_{post} such that $\Gamma \vdash \{A_{pre}\}\ C\ \{A_{post}\}$, we have that $\forall \Delta.\ \Delta \vDash \Gamma \Rightarrow \Delta \vDash \{A_{pre}\}\ [C]_{\Gamma, A_{pre}}\ \{A_{post}\}$.*

Proof. The proof goes by induction on C. Under the assumption that C does not mention `fp`, it is clear for all commands except routine calls that $[C]_{\Gamma, A_{pre}}$ does not change the behavior of C. We can use Lemma 6.5.1, Lemma 6.4.4 and the fact that $s, h \vDash \mathsf{concrete}(A_{pre}) \implies s, h \vDash A_{pre}$ to see that the theorem also holds for routine calls. □

The $\Delta \vDash \Gamma$ condition in the precision theorem states that the context Δ upholds the contracts specified by Γ. Under standard separation logic, this means (amongst other conditions) that context functions cannot read outside the footprint specified by their precondition. However, our precision theorem can actually be slightly stronger than this, because the theorem holds even if the context is allowed to read outside its designated footprint..

6.7 Summary

We started this chapter by formally defining a simple, control-flow safe imperative programming language, similar to C. We then defined an assertion language for meta-theoretical use, and we also defined a more restricted subset of this language for the separation logic contracts that drive our program transformation. These restrictions made the contract language constructive, in the sense that a witness for existentially bound variables can always be derived directly from the logic assertions at hand, rather than requiring a search for a suitable value. We then formally defined the function prod, which models the runtime production of an assertion. This function is sufficient for generating runtime checks for modules that do not make any outcalls to the untrusted context. The function was proved to be correct, by showing that if execution has not trapped after executing $\mathsf{prod}(A)$, then A will hold. We then defined and proved correct the function cons, which models the runtime consumption of an assertion. Finally, both prod and cons were used to define the full transformation function $[C]_{\Gamma, A}$ and this function was proved safe and precise, where safety means that the transformed code never fails, even when it is interacting with an untrusted context, and precision means that the code behaves exactly like the original verified module when interacting with a context that *does* uphold its contracts.

Chapter 7

Conclusion

Nearly every sector of our economy, ranging from healthcare to banking, transportation, education and commerce, increasingly relies on software. As the ubiquity and complexity of software increases, the importance of properly securing the software we use, grows as well. The enormous impact that software bugs can have on society, calls for comprehensive countermeasures. Valuable prior research has come up with various ways to protect programs written in unsafe languages such as C and C++ against a wide variety of low-level attacks, including stack smashing attacks, jump-to-libc attacks and return-oriented programming attacks [122]. What these protection measures have in common, is that they defend against *input-providing* attackers who can interact with the program under attack only by providing input and reading the program's output. In this dissertation, we have adopted the strictly stronger *in-process* attacker model, which assumes that attackers have already somehow gained the ability to execute arbitrary code in the address space of the program under attack. This is a realistic assumption, since practical attacks are often triggered by the injection or loading of untrusted binary code into a process' address space, for instance through kernel-level malware. In this setting, even programs written in safe languages, such as Java or statically verified C, can be exploited by low-level attacks. In particular, the high-level abstractions offered by these languages, such as structured control flow and field access restrictions, can be broken. This essentially means that in order to accurately reason about the security of a program, developers must take into account the low-level implementation details of the compiler and the execution problem that they target. This is in conflict with the accepted principle of source-based reasoning [17], which says that security properties of a software system should follow from review of the system's source code and its source-level semantics alone.

In the first part of this dissertation, we developed a fully abstract compilation scheme that protects against in-process attackers and restores the principle of source-based reasoning for safe source languages. Assuming the low-level target platform provides a protection primitive that allows a region of code and data to be isolated from other code in a program's address space, the compiler will ensure that the high-level abstractions provided by the source programming language are maintained at the low level after compilation.

In the second part of this dissertation, we developed a source-to-source translation scheme that relies on the guarantees provided by this fully abstract compiler, in order to ensure the correct runtime behavior of statically verified C code that interacts with unverified code. The problem with standard modular verification is that, during the static verification process, the verifier can only *assume* that unverified code will behave in accordance with its expected behavior at runtime. Our translation scheme essentially adds a series of runtime checks at the boundary between verified and unverified code, that will detect when unverified code misbehaves.

The rest of this conclusion chapter consists of a more detailed summary of the dissertation in Section 7.1 and finally a discussion of future work and a reflection upon the work performed in Section 7.2.

7.1 Summary

We started this text by discussing the role of abstractions in programming languages. In particular, we pointed out the fact that compilers and programming languages provide abstractions such as module systems, field access modifiers, structured control flow, and platform independence to the programmer. However, standard compilers are not *fully abstract*, which means that two software components that appear equivalent to a third component interacting with them at the source-code level, can be distinguished from each other by a low-level component interacting with them after they have been compiled. In other words, for a standard compiler, high-level contextual equivalence does not imply low-level contextual equivalence. This means that the high-level abstractions provided by the compiler and programming language can be broken by low-level code.

This is especially troublesome when the high-level abstractions are used for security purposes. For instance, access to a private instance field containing a cryptographic key will be restricted at the source-code level, but such variables can simply be read or modified at the assembly level. A fully abstract compiler is able to maintain such properties after compilation. To achieve this property,

the low-level target language must provide some suitable protection primitive. Previous research [3, 62] has shown that address space layout randomization is a sufficiently strong protection primitive to support fully abstract compilation in a probabilistic sense. In Chapter 2, we have shown that a fine-grained, program counter-based memory access control scheme, as provided by state-of-the-art protected module architectures, also qualifies as a suitable protection primitive. We developed a model of such a low-level protection platform and showed how a procedural high-level language can be securely compiled towards it. We also showed that the model is realistic, by creating a prototype that runs on today's commodity computers with an acceptable performance overhead. The resulting security guarantees are strong: any security property that holds at the source-code level and that can be expressed using contextual equivalence, also holds at the assembly level after compilation. The power of a low-level attacker is hence reduced to that of a high-level attacker, because any vulnerability that can be exploited at the low level is explainable at the source-code level.

In Chapter 3, we formalized the languages and compiler described in Chapter 2, and we proved the compilation scheme to be fully abstract. For this, we first defined high- and (revised) low-level execution traces, which were shown to be fully abstract in their own right. We then defined an algorithm that, given two high-level modules and trace-based evidence that their low-level compilations are contextually inequivalent, will generate a high-level context module that can distinguish the two modules at the high level. This algorithm formed the basis of our full abstraction proof.

In Chapter 4, we changed focus from full abstraction to Hoare logic-based formal software verification. We showed that Hoare logic provides a sound formal basis for reasoning about imperative programs, but is limited in its ability to reason about pointer programs, due to the effects of pointer aliasing. Separation logic does away with this limitation, and can thus reason about languages with pointers to shared mutable data structures, such as C, C++ and Java. We then described the symbolic execution algorithm that underlies the VeriFast program verifier, thereby revealing how separation logic can be used for semi-automatic program verification.

We argued that in order to ensure scalability of a verification method, it is essential that the method is modular. That is, it should be possible to soundly verify only a part of an application, leaving the rest of the code unverified. The verification methods studied in Chapter 4 attain this modularity property, by allowing users to specify the behavior of unverified code. The static verifier will then *assume* that the unverified code behaves as described. Unfortunately, this provides only limited guarantees at runtime. Bugs in the unverified part of the program can still impact the state of the verified part, and hence might trigger failures in verified modules.

In Chapter 5, we proposed a way to transform partially verified programs such that the provided runtime guarantees are significantly better. Our code transformation adds a series of runtime checks at the boundary between verified and unverified code, in order to detect when unverified code does not behave according to its contract. If a bug is triggered in the unverified part of the program, this is detected before it can impact the state or control flow of the verified module. A key part of this approach consists of ensuring that memory errors in unverified code cannot corrupt the state of verified code. We solved this problem in two steps. First, we rely on the fully abstract compilation scheme of Chapter 2 for protecting the local variables and control flow metadata of calls to verified functions on the call stack. Secondly, the runtime checks perform an integrity on heap memory: before calling an unverified function from verified code, a cryptographic hash is calculated over the heap memory owned by verified code, and this hash is re-calculated and verified when control returns to verified code. Hence, any illegal heap modifications performed by unverified code that can impact the execution of verified code will be detected upon re-entry.

The combination of the boundary checks and the secure compilation protection of local variables and control flow, results in a very strong modular soundness guarantee: no verified assertion in the verified codebase will ever fail at runtime, even if that code runs as part of a partially verified application. These guarantees are useful for testing, as they help to detects bugs faster, and for security, as they guarantee that verified properties of modules continue to hold in the presence of code injection attacks against the unverified part of the program. We have benchmarked the performance of the transformed code, and have found that the overhead is low if the boundary between verified and unverified code is chosen wisely, thereby demonstrating the real-world feasibility of our solution.

Finally, in Chapter 6 we formalized the languages and program transformations described in Chapter 5 and proved that the transformations are safe and precise. In this context, safety means that a verified assertion in the transformed code never fails at runtime, even when the code is interacting with an unverified context, and precision means that the code behaves exactly like the original verified module does when interacting with a context that *does* uphold its contracts.

7.2 Future work and reflection

7.2.1 Fully abstract compilation

The compilation scheme described in Chapter 2 and Chapter 3 has been extended by Patrignani et al. to support dynamic memory allocation, dynamic dispatch and exceptions [99, 96, 94]. Further extensions include support for secure interactions between modules of multiple mutually-distrusting parties. In this settings, each module would require its own protected memory area, including its own secure call stack. This would make interactions between modules more complicated, because neither module can read from or write to the other module's stack, hence a different kind of calling convention must be established. Furthermore, new attack vectors might appear due to the increased complexity of multiple interacting modules. For instance, an attacker could try to make two low-level modules interact in ways that could never occur at the high level, leading to undefined behavior that is dependent on the specific implementation of a module.

Another extension that would enrich the source language, is adding support for garbage collection. The most common type of garbage collection is the so-called *tracing garbage collection*, which involves scanning over all objects in memory. Hence, the garbage collector would have to have access to every object of the program, whether it is part of a protected module or not. This could prove to be troublesome to implement on the protected module architectures available today, especially in the case of multiple mutually-distrusting modules. Furthermore, the garbage collector might leak information about the allocation status of objects of one module to another module, thereby breaking full abstraction if this information is not available at the source-code level.

Future work could also focus on extending the low-level assembly model, as it currently only models a small subset of the features available on real computing platforms. For instance, real systems have caching memory hierarchies and a real-time clock, the combination of which can potentially break full abstraction.

However, while more accurate models of the low-level execution model can lead to new insights on how to achieve full abstraction on real systems, we should not lose focus on the real-world security improvements we are trying to achieve. As discussed in Section 2.4.1, one could argue that full abstraction is a too strong property to demand for secure compilation. For instance, having to hide the order of a module's functions in memory and having to allocate a fixed amount of memory for any module, are necessary requirements for obtaining full abstraction, but these requirements do not seem to result in any real-world security benefits. Therefore, I believe we should consider full abstraction to

be an ideal property for compilers towards low-level code to have, but for the development of real-world security-oriented compilers, a balanced trade-off should be made between the real-world security benefits of a security feature and its performance overhead.

It is also important to keep in mind that not *all* security problems are solved by secure compilation. For instance, as argued in Section 1.3, logic flaws or other bugs that are present at the source-code level will still be present after fully abstract compilation. True application security should hence not depend on secure compilation alone.

7.2.2 Sound modular software verification

An important future work track for the program transformation described in Chapter 5 and Chapter 6 consists of building and benchmarking a full end-to-end implementation that runs on a desktop-level protected module architecture. Unfortunately, the currently available PMA prototypes are still in an experimental state, prohibiting us from running meaningful macro benchmarks on top of them at this time. However, recent developments regarding Intel's SGX extensions [56] indicate that a low-overhead, hardware-based PMA platform will likely be available on standard desktop systems in the near future. Once these systems are available, a fully abstract compiler from verified C code towards the SGX platform can be developed on top of which the hardening translations can be benchmarked. This end-to-end implementation would allow us to accurately measure the total real-world performance overhead introduced by (1) the runtime checks inserted by our hardening transformations, (2) the runtime checks introduced by the fully abstract compiler and (3) the SGX platform itself. Based on the outcome of these benchmarks, further performance improvements can be developed for different aspects of the system.

Since the micro benchmarks of Section 5.6.1 indicate that considerable time is spent calculating the cryptographic hash over the verified module's footprint, future work can focus on reducing the hashing overhead. However, Section 5.6.4 has pointed out that making a secure copy of the footprint instead of hashing it, results in an overall performance gain of only 20%. Hence a different approach is necessary to make the system truly scalable. As proposed in Section 5.6.4, an interesting approach for improving performance would be to take advantage of hardware page protection support. Whenever a full memory page needs to be integrity protected, it could be marked read-only using the memory management unit of the CPU, which would cause a page fault to be triggered when unverified code tries to access the page in question. When only part of

a memory page must be integrity protected, the system could fall back to the hashing- or copying-based approach.

Reflecting on the practical usefulness of our hardening translations, one could argue that our solution does not improve the overall security of an application, since, although verified code is guaranteed to behave as expected, an attacker can still perform code-injection attacks on unverified code and hence can still execute arbitrary code within an application's address space. This is a valid point for monolithic applications as they are today, but it does not hold for compartmentalized applications in which each compartment can be individually identified and authenticated. For instance, Salus [112] allows applications to be subdivided into compartments, each of which can authenticate the compartments it calls and the compartments from which it is called. Remote authentication and attestation of compartments is also possible. In this setting, an attacker can still inject and execute code into an unverified compartment, but the amount of damage that can be performed will be limited by the amount of trust that was placed in that compartment by other compartments. Since it is reasonable to assume that the amount of trust placed in an unverified compartment is much lower than that placed in a verified compartment, our hardening translation can significantly improve the security of an application.

Another situation in which it is useful from a security perspective to have a software module that is guaranteed not to fail, is when that module has exclusive access to a resource. For instance, if we were to place the address range of a memory-mapped I/O device inside the protected memory area of a hardened module, we would give that module exclusive access to the device. The module can then provide a secure API for accessing the device towards other components, on which it can enforce arbitrary security constraints. Our program transformation ensures that such constraints can never be broken at runtime.

Finally, it is again important to note that not all security bugs in the verified module can be solved by our hardening translation. Since the correctness of our translation relies on the guarantees provided by our fully abstract compilation scheme, the limitations that apply to that scheme also apply here. Furthermore, although sound software verification ensures adherence of the verified code to its specifications, not all security properties can be expressed in the specification language and it is of course still possible for a developer to make mistakes in specifying the expected behavior of the software under verification. Therefore, a comprehensive approach to secure application development should not depend on software verification and secure compilation alone.

Appendix A

Secure compiler source code

```
(* *************** AST elements of the high-level language *********** *)
type hl_field_index =
 FI of (int)
;;

type hl_var_index =
 VI of (int)
;;

type hl_method_index =
 MI of (int)
;;

type hl_label_index =
 LI of (int)
;;

type hl_type =
    TUnit
  | TInt
  | TMethod of (hl_type list * hl_type)
;;

type hl_value =
    VUnit
  | VNull
  | VInt of int
  | VMethod of (hl_method_index)
;;


type hl_field_decl =
 FDecl of (
  hl_type *             (* field type *)
  hl_field_index *      (* field index *)
  hl_value              (* initial value *)
 )
;;
```

```
type hl_var_decl =
 VDecl of (
  hl_type *              (* variable type *)
  hl_var_index *         (* variable index *)
  hl_value               (* initial value *)
 )
;;

type hl_param_decl =
 PDecl of (
  hl_type *              (* parameter type *)
  hl_var_index           (* parameter index *)
 )
;;

type hl_expression =
   EMovi  of (hl_value)
 | EMov   of (hl_var_index)
 | EAdd   of (hl_var_index * hl_var_index)
 | ESub   of (hl_var_index * hl_var_index)
 | EField of (hl_field_index)
 | ECalli of (hl_method_index * hl_var_index list)
 | ECall  of (hl_var_index * hl_var_index list)
;;

type hl_statement =
   SVarAssign   of (hl_var_index * hl_expression)
 | SFieldAssign of (hl_field_index * hl_var_index)
 | SJmp         of (hl_label_index)
 | SBeq         of (hl_var_index * hl_var_index * hl_label_index)
 | SBlt         of (hl_var_index * hl_var_index * hl_label_index)
 | SRet         of (hl_var_index)
;;

type hl_line = (
 hl_label_index *       (* line label *)
 hl_statement           (* statement *)
)

type hl_method_decl =
 MDecl of (
  hl_type *              (* return type *)
  hl_method_index *      (* method index *)
  hl_param_decl list *   (* parameters *)
  hl_var_decl list *     (* variable declarations and initializations *)
  hl_line list           (* method body *)
 )
;;

type hl_module_decl =
 ODecl of (
  hl_field_decl list *   (* fields *)
  hl_method_decl list    (* methods *)
 )
;;


(* ************** Instructions of the low-level language ************* *)
type ll_label =
  LLabel of (string)
;;
```

```
type ll_address =
  LAddr of (int)
;;

type ll_immediate =
    IInt   of (int)
  | IAddr  of (ll_address)
  | ILabelRef of (ll_label)
;;

type ll_register =
    PC
  | R0
  | R1
  | R2
  | R3
  | R4
  | R5
  | R6
  | R7
  | R8
  | R9
  | R10
  | R11
  | SP
;;

type ll_instruction =
    IMovl of (ll_register * ll_register)
  | IMovs of (ll_register * ll_register)
  | IMovi of (ll_register * ll_immediate)
  | IAdd  of (ll_register * ll_register)
  | ISub  of (ll_register * ll_register)
  | ICmp  of (ll_register * ll_register)
  | IJmp  of (ll_register)
  | IJe   of (ll_register)
  | IJl   of (ll_register)
  | ICall of (ll_register)
  | IRet
  | IData of (ll_immediate)
  | IHalt
  | INop
;;

type ll_line = (
  ll_label option *     (* label *)
  ll_address *          (* memory address *)
  ll_instruction        (* instruction *)
)


type ll_mod_descriptor =
  ModDescriptor of (
    int *               (* number of entry points *)
    int *               (* size of code section *)
    int                 (* size of data section *)
  )
;;

type ll_module =
  Module of (
    hl_module_decl *    (* reference to high-level source *)
```

```
    ll_line list *      (* module instructions *)
    ll_address *        (* next free address *)
    ll_mod_descriptor   (* module descriptor *)
 )
;;


(* **** Translation of a high-level object to a low-level module ***** *)
exception Too_Many_Parameters;;


(* Addresses *)
let mod_base_address =
  LAddr(0x10000000)
;;

let mod_end_address =
  LAddr(0x8FFFFFFF)
;;

let mod_entry_base_address =
  mod_base_address
;;

let secure_stack_base_address =
  LAddr(0x6FFFFFFF)
;;

let code_base_address =
  mod_base_address
;;

let code_end_address =
  LAddr(0x4FFFFFFF)
;;

let data_base_address =
  LAddr(0x50000000)
;;

let data_end_address =
  mod_end_address
;;

let null_address =
  IAddr(LAddr(0xFFFFFFFF))
;;

(* Sizes *)
let code_size =
  match (code_base_address, code_end_address) with
  | (LAddr(cb), LAddr(ce)) ->
    ce - cb
;;

let data_size =
  match (data_base_address, data_end_address) with
  | (LAddr(sb), LAddr(se)) ->
    se - sb
;;

(* Label names *)
```

```
let stack_base_label =
  LLabel("stack_base")
;;

let field_label (FI(fi) : hl_field_index) =
  LLabel("field" ^ (string_of_int fi))
;;

let method_entry_point_label (MI(mi) : hl_method_index) =
  LLabel("method" ^ (string_of_int mi))
;;

let method_entry_point_label_suffix
    (MI(mi) : hl_method_index) (suffix : string) =
  LLabel("method" ^ (string_of_int mi) ^ "_" ^ suffix)
;;

let method_exit_point_label (MI(mi) : hl_method_index) =
  LLabel("method" ^ (string_of_int mi) ^ "_exit")
;;

let method_prologue_label (MI(mi) : hl_method_index) =
  LLabel("method" ^ (string_of_int mi) ^ "_prologue")
;;

let method_epilogue_label (MI(mi) : hl_method_index) =
  LLabel("method" ^ (string_of_int mi) ^ "_epilogue")
;;

let method_label_label
    (MI(mi) : hl_method_index) (LI(li) : hl_label_index) =
  LLabel("method" ^ (string_of_int mi) ^ "_label" ^ (string_of_int li))
;;

let return_entry_point_label =
 LLabel("return_entry_point")
;;

let return_entry_point_label_suffix (suffix : string) =
 LLabel("return_entry_point" ^ "_" ^ suffix)
;;

let outcall_helper_label =
  LLabel("outcall_helper")
;;

let full_register_cleanup_helper_label =
  LLabel("register_cleanup_helper")
;;

let return_address_check_helper_label =
  LLabel("rac_helper")
;;

let halt_helper_label =
  LLabel("halt_helper")
;;

let shadow_stack_pointer_field_label =
  LLabel("shadow_stack_pointer")
;;

(* Register for parameter *)
```

```
let register_for_param_index (i : int) =
  match i with
  | 0 -> R4
  | 1 -> R5
  | 2 -> R6
  | 3 -> R7
  | 4 -> R8
  | 5 -> R9
  | 6 -> R10
  | 7 -> R11
  | _ -> raise Too_Many_Parameters
;;

let register_for_param (VI(i) : hl_var_index) =
  register_for_param_index i
;;


(* ************************** Translations ************************** *)
(* Translates a high-level value to a corresponding low-level value *)
let translate_value (v : hl_value) =
  match v with
    | VUnit   -> IInt(0)
    | VNull   -> null_address
    | VInt(i) -> IInt(i)
    | VMethod(mi) -> ILabelRef (method_prologue_label mi)
;;

(* Reserves some memory at the current position in the low-level module,
   for a given high-level field. *)
let generate_field (m : ll_module) (fd : hl_field_decl) =
  match (m, fd) with
  | (Module(src, lines, LAddr(free_addr), d), FDecl(_, index, value)) ->
      let new_line = (Some(field_label index), LAddr(free_addr),
                      IData(translate_value value)) in
      let new_addr = LAddr (free_addr + 1) in
        Module(src, lines @ [new_line], new_addr, d)
;;


(* ***** Common translation helper functions ***** *)
(* Appends a low-level instruction to the low-level module. *)
let append_instruction (m : ll_module) (i : ll_instruction) =
  match m with Module(src, lines, LAddr(free_addr), d) ->
      let new_lines = lines @ [(None, (LAddr free_addr), i)] in
      let new_addr = LAddr(free_addr + 1) in
        Module(src, new_lines, new_addr, d)
;;

(* Generates a label at the current position in the low-level module. *)
let append_label (m : ll_module) (l : ll_label) =
  match m with Module(src, lines, LAddr(free_addr), d) ->
      let new_lines = lines @ [(Some l, (LAddr free_addr), INop)] in
      let new_addr = LAddr(free_addr + 1) in
        Module(src, new_lines, new_addr, d)
;;

(* Generates instructions for loading the address of a given label into
 * a given register at the current position in the low-level module. *)
let generate_load_label (m : ll_module) (l : ll_label) (r : ll_register) =
  append_instruction m (IMovi(r, ILabelRef(l)))
;;
```

```
(* Generates instructions for jumping to a given label at the current
 * position in the low-level module. *)
let generate_label_jump (m : ll_module) (l : ll_label) =
  let m0 = generate_load_label m l R3 in
  append_instruction m0 (IJmp(R3))
;;

(* Generates instructions for calling a given label at the current
 * position in the low-level module. *)
let generate_label_call (m : ll_module) (l : ll_label) =
  let m0 = generate_load_label m l R3 in
  append_instruction m0 (ICall(R3))
;;

(* Generates a list of instructions that checks whether the value in a
 * given register is in- or outside of the module's memory boundaries and
 * jumps to to the label lin if it is inside the memory boundaries or to
 * the label lout otherwise. The given register r cannot be R1 or R3. *)
let generate_address_in_mod_check
   (m : ll_module) (r : ll_register) (lin : ll_label) (lout : ll_label) =
  List.fold_left append_instruction m
    [(IMovi(R3, ILabelRef(lout)));     (* Load out jump address in R3  *)
     (IMovi(R1, IAddr(mod_base_address))); (* Load base address in R1 *)
     (ICmp(r, R1));                    (* Jump to the out label if...  *)
     (IJl(R3));                        (* ... r < base address         *)
     (IMovi(R1, IAddr(mod_end_address)));  (* Load end address in R1  *)
     (ICmp(R1,r));                     (* Jump to the out label if...  *)
     (IJl(R3));                        (* end address < r              *)
     (IMovi(R3, ILabelRef(lin)));      (* Load in jump address into R3 *)
     (IJmp(R3))]                       (* Jump to in label             *)
;;

(* Generates a list of instructions that checks whether R0 is equal to a
 * given immediate and jumps to a given label if so. *)
let generate_equals_check
    (m : ll_module) (i : ll_immediate) (leq : ll_label) =
  List.fold_left append_instruction m
    [IMovi(R3, ILabelRef(leq));
     IMovi(R1, i);
     ICmp(R0, R1);
     IJe(R3)]
;;

(* Generates a list of instructions that pushes the value in a given
 * register onto the run-time stack. *)
let generate_push_onto_stack (m : ll_module) (r : ll_register) =
  List.fold_left append_instruction m
    [IMovi(R2, IInt(1));
     ISub(SP, R2);
     IMovs(SP, r)]
;;

(* ****************** Method entry point generation ***************** *)
(* Generates instructions for swapping the value of the SP register with
 * the shadow stack pointer field. *)
let generate_stack_switch (m : ll_module) =
  List.fold_left append_instruction m
    [IMovi(R2, (ILabelRef shadow_stack_pointer_field_label));
     IMovl(R3, R2); (* Load value of shadow stack pointer field in R3  *)
     IMovs(R2, SP); (* Store value of SP in shadow stack pointer field *)
     IMovi(SP, (IInt 0));
     IAdd(SP, R3)]  (* Store value of R3 in SP                        *)
;;
```

```
(* Generates instructions for clearing registers R1, R2, R3 and the flags
 * register *)
let generate_working_register_cleanup (m : ll_module) =
  List.fold_left append_instruction m
    [IMovi(R1, IInt(0));
     IMovi(R2, IInt(0));
     IMovi(R3, IInt(0));
     ICmp(R1, R2)]
;;

(* Generates instructions that load the value at the top of the stack
 * into R0 and then call the return_address_check_helper to check
 * whether the value is outside of the bounds of the protected module *)
let generate_return_address_check_call (m : ll_module) =
  List.fold_left append_instruction m
    [IMovl(R0, SP);
     IMovi(R3, (ILabelRef return_address_check_helper_label));
     ICall(R3)]
;;

(* Returns the next entry point address. That is, this function returns
 * the closest multiple of 128 greater than or equal to the given
 * address *)
let next_entry_point_address (LAddr(a) : ll_address) =
  let new_a = truncate (128.0 *. (ceil ((float_of_int a) /. 128.0))) in
    LAddr(new_a)
;;



(* Generates a list of instructions forming the entry point for a given
 * high-level method *)
let generate_entry_point (m : ll_module) (md : hl_method_decl) =
  match (m, md) with (Module(src, lines, a, d), MDecl(_,index,_,_,_)) ->
      let m = Module(src, lines, next_entry_point_address a, d) in
      let m = append_label m (method_entry_point_label index) in(*Entry*)
      let m = generate_address_in_mod_check m SP halt_helper_label
              (method_entry_point_label_suffix index "sp_ok") in
      let m = append_label m
              (method_entry_point_label_suffix index "sp_ok") in
      let m = generate_stack_switch m in
      let m = generate_label_call m (method_prologue_label index) in
      let m = append_label m (method_exit_point_label index) in  (*Exit*)
      let m = generate_stack_switch m in
      let m = generate_return_address_check_call m in
      let m = generate_label_call m full_register_cleanup_helper_label in
      append_instruction m IRet
;;


(* Generates a list of instructions forming the return entry point *)
let generate_return_entry_point (m : ll_module) =
  match m with Module(src, lines, a, d) ->
      let m = Module(src, lines, next_entry_point_address a, d) in
      let m = append_label m return_entry_point_label in
      let m = generate_address_in_mod_check m SP halt_helper_label
              (return_entry_point_label_suffix "sp_ok") in
      let m = append_label m
              (return_entry_point_label_suffix "sp_ok") in
      let m = generate_stack_switch m in
      append_instruction m IRet
;;
```

```
(* ********************** Method code generation ******************** *)

(* Generates a list of instructions to initialize a local variable. It
 * assumes the register R0 contains the value 1. *)
let generate_local_var_init (m : ll_module) (var : hl_var_decl) =
  match var with VDecl(_, index, value) ->
    List.fold_left append_instruction m
      [ISub(SP, R0);
       IMovi(R1, translate_value value);
       IMovs(SP, R1)]
;;

(* Generates a list of instructions to push a parameter contained in a
 * register onto the stack. It assumes the register R0 contains the
 * value 1. *)
let generate_param_copy_to_stack
    (m : ll_module) (param : hl_param_decl) =
  match param with
  | PDecl(TUnit, index) ->
    List.fold_left append_instruction m
      [ISub(SP, R0);
       IMovi(R1, IInt(0));
       IMovs(SP, R1)]
  | PDecl(_, index) ->
    List.fold_left append_instruction m
      [ISub(SP, R0);
       IMovs(SP, register_for_param index)]
;;

(* Generates a prologue for a given high-level method. *)
let generate_prologue (m : ll_module) (md : hl_method_decl) =
  match md with MDecl(_, index, params, vars, body) ->
    let m0 = append_label m (method_prologue_label index) in
    let m1 = append_instruction m0 (IMovi(R0, IInt(1))) in
    let m2 = List.fold_left generate_param_copy_to_stack m1 params in
    List.fold_left generate_local_var_init m2 vars
;;

(* Generates instructions that load the address of a given local variable
 * into register R3. *)
let generate_var_address (m : ll_module) (VI(vi) : hl_var_index) =
  List.fold_left append_instruction m
    [IMovi(R3, IInt(0));
     IAdd(R3, SP);
     IMovi(R2, IInt(vi));
     IAdd(R3,R2)]
;;

(* Generates instructions that store the value of a given register into a
 * given local variable. *)
let generate_store_var
    (m : ll_module) (vi : hl_var_index) (r : ll_register) =
  let m0 = generate_var_address m vi in
  append_instruction m0 (IMovs(R3, r))
;;

(* Generates instructions that load the value of a given local variable
 * into a given register. *)
let generate_load_var
    (m : ll_module) (vi : hl_var_index) (r : ll_register) =
  let m0 = generate_var_address m vi in
  append_instruction m0 (IMovl(r, R3))
```

```
;;

(* Generates instructions for storing the value of a given register into
 * a given field *)
let generate_store_field
    (m : ll_module) (fi : hl_field_index) (r : ll_register) =
  List.fold_left append_instruction m
    [IMovi(R3, ILabelRef(field_label fi));
     IMovs(R3, r)]
;;

(* Generates instructions for loading the value of a given field into a
 * given register *)
let generate_load_field
    (m : ll_module) (fi : hl_field_index) (r : ll_register) =
  List.fold_left append_instruction m
    [IMovi(R3, ILabelRef(field_label fi));
     IMovl(r, R3)]
;;

(* Generates instructions for performing a compare between two given local
 * variables and for loading the address of a given label into a given
 * register. Usefull for performing conditional jumps. *)
let generate_cjump (m : ll_module) (x1 : hl_var_index) (x2 : hl_var_index)
    (l : ll_label) (r : ll_register) =
  let m0 = generate_load_var   m  x1 R0 in
  let m1 = generate_load_var   m0 x2 R1 in
  let m2 = generate_load_label m1 l r in
  append_instruction m2 (ICmp(R0, R1))
;;

(* Generates instructions for clearing all registers with a given index
 * or higher. *)
let rec generate_register_clear (m : ll_module) (i : int) =
  try (
      let m = append_instruction m
                (IMovi(register_for_param_index i, IInt 0)) in
      generate_register_clear m (i + 1))
  with Too_Many_Parameters -> m
;;

(* Generates instructions for copying a given local variable to a
 * register with a given index. *)
let generate_param_copy_to_register
    ((m,i) : ll_module * int) (param : hl_var_index) =
  (generate_load_var m param (register_for_param_index i), i+1)
;;

(* Generates instructions for copying a given list of local variables to
 * the parameter registers. *)
let generate_params_copy_to_registers
    (m : ll_module) (params : hl_var_index list) =
  let copied =
     List.fold_left generate_param_copy_to_register (m,0) params in
  generate_register_clear (fst copied) (snd copied)
;;

let is_param (VI(i) : hl_var_index) (param : hl_param_decl) =
  match param with PDecl(typ, VI(index)) ->
    i == index
;;

let is_var (VI(i) : hl_var_index) (var : hl_var_decl) =
```

```
  match var with VDecl(typ, VI(index), init_value) ->
    i == index
;;

let get_type (md : hl_method_decl) (i : hl_var_index) =
  match md with MDecl(_, _, params, vars, _) ->
    try match (List.find (is_param i) params) with
    | PDecl(typ, _) -> typ with
    | Not_found ->
    match (List.find (is_var i) vars) with
    | VDecl(typ, _, _) -> typ
;;

let is_unit_method_ref (t : hl_type) =
  match t with
  | TMethod(_, ret) -> ret == TUnit
  | _               -> false
;;

(* Generates instructions for calculating the value of a high-level
 * expression. *)
let generate_expression
    (m : ll_module) (md : hl_method_decl) (e : hl_expression) =
  match e with
  | EMovi (v) ->
     append_instruction m (IMovi(R0, translate_value v))
  | EMov (x) ->
     generate_load_var m x R0
  | EAdd (x1, x2) ->
     let m = generate_load_var m  x1 R0 in
     let m = generate_load_var m x2 R1 in
     append_instruction m (IAdd(R0,R1))
  | ESub (x1, x2) ->
     let m = generate_load_var m  x1 R0 in
     let m = generate_load_var m x2 R1 in
     append_instruction m (ISub(R0,R1))
  | EField (f) ->
     generate_load_field m f R0
  | ECalli (mi,p) ->
     let m = generate_load_label m (method_prologue_label mi) R0 in
     let m = generate_params_copy_to_registers m p in
     append_instruction m (ICall(R0))
  | ECall (x,p) -> (* outcall *)
     let m = generate_params_copy_to_registers m p in
     let m = generate_load_var m x R0 in
     let m = generate_label_call m outcall_helper_label in
     if (is_unit_method_ref (get_type md x)) then
       append_instruction m (IMovi(R0, IInt(0)))
     else m
;;

(* Generates instructions performing the corresponding operations of a
 * given high-level statement *)
let generate_line_code (mi : hl_method_index) (md : hl_method_decl)
    (m : ll_module)  ((l, s) : hl_line) =
  let m0 = append_label m (method_label_label mi l) in
    match s with
    | SVarAssign (x, e)   ->
       let m1 = generate_expression m0 md e in
       generate_store_var m1 x R0
    | SFieldAssign (f, x) ->
       let m1 = generate_load_var m0 x R0 in
       generate_store_field m1 f R0
```

```
    | SJmp (l)              ->
        generate_label_jump m0 (method_label_label mi l)
    | SBeq (x1, x2, l)     ->
        let m1 = generate_cjump m0 x1 x2 (method_label_label mi l) R3 in
        append_instruction m1 (IJe(R3))
    | SBlt (x1, x2, l)     ->
        let m1 = generate_cjump m0 x1 x2 (method_label_label mi l) R3 in
        append_instruction m1 (IJl(R3))
    | SRet (x)             ->
        generate_load_var m0 x R0
;;

(* Generates instructions for deallocating an activation record from the
 * stack. *)
let generate_stack_teardown (m : ll_module) (md : hl_method_decl) =
  match md with MDecl(_, _, params, vars, _) ->
    List.fold_left append_instruction m
      [IMovi(R1, IInt(List.length params + List.length vars));
       IAdd(SP,R1)]
;;


(* Generates the epilogue for a given high-level method. *)
let generate_epilogue (m : ll_module) (md : hl_method_decl) =
  match md with MDecl(_, index, _, _, _) ->
  let m0 = append_label m (method_epilogue_label index) in
  let m1 = generate_stack_teardown m0 md in
  append_instruction m1 IRet            (* Return back to entry point *)
;;

(* Generates instructions for a given high-level method. *)
let generate_method_code (m : ll_module) (md : hl_method_decl) =
  match md with MDecl(_, index, params, vars, body) ->
      let m0 = generate_prologue m md in
      let m1 = List.fold_left (generate_line_code index md) m0 body in
      generate_epilogue m1 md
;;

(* ***** Helper procedures ***** *)
(* Generates a helper method for checking whether an address in R0 is
 * within the memory bounds of the protected module *)
let generate_return_address_check_helper (m : ll_module) =
  let m0 = append_label m return_address_check_helper_label in
  let m1 = generate_address_in_mod_check m0 R0 halt_helper_label
           (LLabel "rac_helper_ok") in
  let m2 = append_label m1 (LLabel "rac_helper_ok") in
  append_instruction m2 IRet
;;



(* Generates a helper method for performing an outcall to unprotected
 * code. The address in unprotected code to jump to is assumed to be in
 * R0. *)
let generate_outcall_helper (m : ll_module) =
  let m = append_label m outcall_helper_label in
  let m = generate_equals_check m null_address halt_helper_label in
  let m = generate_address_in_mod_check m R0 halt_helper_label
          (LLabel"outcall_helper_out") in
  let m = append_label m (LLabel "outcall_helper_out") in
  let m = generate_stack_switch m in
  let m = generate_load_label m return_entry_point_label R3 in
  let m = generate_push_onto_stack m R3 in
```

```
  let m = generate_working_register_cleanup m in
  append_instruction m (IJmp(R0))
;;

(* Generates a helper method for clearing all register (including the
 * flags register) except R0. *)
let generate_full_register_cleanup_helper (m : ll_module) =
  let m0 = append_label m full_register_cleanup_helper_label in
  List.fold_left append_instruction m0
    [IMovi(R1, IInt(0));
     IMovi(R2, IInt(0));
     IMovi(R3, IInt(0));
     IMovi(R4, IInt(0));
     IMovi(R5, IInt(0));
     IMovi(R6, IInt(0));
     IMovi(R7, IInt(0));
     IMovi(R8, IInt(0));
     IMovi(R9, IInt(0));
     IMovi(R10, IInt(0));
     IMovi(R11, IInt(0));
     ICmp(R1, R2);
     IRet]
;;

(* Generates a helper method for storing 0 into R0 and then halting the
 * system. *)
let generate_halt_helper (m : ll_module) =
  let m = append_label m halt_helper_label in
  List.fold_left append_instruction m
    [IMovi(R0, IInt(0));
     IHalt]
;;

(* Generates the four helper functions defined above. *)
let generate_helper_methods (m : ll_module) =
  let m0 = generate_return_address_check_helper m in
  let m1 = generate_outcall_helper m0 in
  let m2 = generate_full_register_cleanup_helper m1 in
  generate_halt_helper m2
;;

(* ***** Fields ***** *)

(* Reserves a memory cell for the shadow stack pointer field. *)
let generate_shadow_stack_pointer_field (m : ll_module) =
  match m with Module(src, lines, LAddr(free_addr), d) ->
    let new_lines = lines @
                    [(Some shadow_stack_pointer_field_label,
                     LAddr(free_addr),
                     IData(IAddr secure_stack_base_address))] in
    let new_addr = LAddr (free_addr + 1) in
      Module(src, new_lines, new_addr, d)
;;

(* Reserves memory space for all helper fields. *)
let generate_helper_fields (m : ll_module) =
  generate_shadow_stack_pointer_field m
;;

(* Initializes the secure stack. *)
let generate_initial_stack (m : ll_module) =
  match (m, secure_stack_base_address) with
  (Module(src, lines, LAddr(free_addr), d), LAddr(s)) ->
```

```
    let new_lines = lines @ [
      (Some stack_base_label, secure_stack_base_address,
       IData(ILabelRef(halt_helper_label)))] in
    let new_addr = LAddr(s + 1) in
      Module(src, new_lines, new_addr, d)
;;


(* ********************** Module translation ********************** *)
(* The entry point of the compiler. Translates a high-level module
   into a low-level one. *)
let translate_module (o : hl_module_decl) =
  match o with
  | ODecl(fields, methods) ->
    let m0 = Module(o, [], mod_base_address, ModDescriptor(
               (List.length methods) + 1, code_size, data_size)) in
    let m1 = List.fold_left generate_entry_point m0 methods in
    let m2 = generate_return_entry_point m1 in
    let m3 = generate_helper_methods m2 in
    let m4 = List.fold_left generate_method_code m3 methods in
    let m5 = generate_initial_stack m4 in
    let m6 = generate_helper_fields m5 in
    List.fold_left generate_field m6 fields
;;
```

Bibliography

[1] M. Abadi. "Protection in Programming-Language Translations". In: *Proceedings of the 25th International Colloquium on Automata, Languages and Programming*. ICALP '98. London, UK, UK: Springer-Verlag, 1998, pp. 868–883. ISBN: 3-540-64781-3. URL: http://dl.acm.org/citation.cfm?id=646252.686313.

[2] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. "Control-flow integrity principles, implementations, and applications". In: *ACM Trans. Inf. Syst. Secur.* 13 (1 Nov. 2009), 4:1–4:40. ISSN: 1094-9224. DOI: http://doi.acm.org/10.1145/1609956.1609960. URL: http://doi.acm.org/10.1145/1609956.1609960.

[3] M. Abadi and G. Plotkin. "On Protection by Layout Randomization". In: *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium*. Computer Security Foundations Symposium (CSF), 2010 IEEE 23th. Washington, DC, USA: IEEE Computer Society, 2010, pp. 337–351. ISBN: 978-0-7695-4082-5. DOI: 10.1109/CSF.2010.30. URL: http://dx.doi.org/10.1109/CSF.2010.30.

[4] P. Agten, B. Jacobs, and F. Piessens. "Sound Modular Verification of C Code Executing in an Unverified Context". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '15. Mumbai, India: ACM, 2015, pp. 581–594.

[5] P. Agten, B. Jacobs, and F. Piessens. *Sound modular verification of C code executing in an unverified context: extended version*. CW Reports CW676. Department of Computer Science, KU Leuven, Nov. 2014.

[6] P. Agten, W. Joosen, F. Piessens, and N. Nikiforakis. "Seven months' worth of mistakes: A longitudinal study of typosquatting abuse". In: *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS 2015)*. Internet Society, Feb. 2015.

[7] P. Agten, N. Nikiforakis, R. Strackx, W. De Groef, and F. Piessens. "Recent Developments in Low-level Software Security". In: *Proceedings of the 6th IFIP WG 11.2 International Conference on Information Security Theory and Practice: Security, Privacy and Trust in Computing Systems and Ambient Intelligent Ecosystems*. WISTP'12. Egham, UK: Springer-Verlag, 2012, pp. 1–16.

[8] P. Agten, R. Strackx, B. Jacobs, and F. Piessens. "Secure Compilation to Modern Processors". In: *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*. June 2012, pp. 171–185.

[9] P. Agten, R. Strackx, B. Jacobs, and F. Piessens. *Secure compilation to modern processors: extended version*. CW Reports CW619. Department of Computer Science, KU Leuven, Apr. 2012.

[10] P. Agten, S. Van Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens. "JSand: complete client-side sandboxing of third-party JavaScript without browser modifications". In: *Proceedings of the 28th Annual Computer Security Applications Conference*. ACSAC '12. Orlando, Florida: ACM, 2012, pp. 1–10.

[11] K. R. Apt. "Ten Years of Hoare's Logic: A Survey - Part I". In: *ACM Trans. Program. Lang. Syst.* 3.4 (Oct. 1981), pp. 431–483. ISSN: 0164-0925. DOI: 10.1145/357146.357150. URL: http://doi.acm.org/10.1145/357146.357150.

[12] K. R. Apt. "Ten Years of Hoare's Logic: A Survey - Part II: Nondeterminism". In: *Theoretical Computer Science* 28.1–2 (1983), pp. 83–109. ISSN: 0304-3975. DOI: http://dx.doi.org/10.1016/0304-3975(83)90066-X. URL: http://www.sciencedirect.com/science/article/pii/030439758390066X.

[13] T. M. Austin, S. E. Breach, and G. S. Sohi. "Efficient Detection of All Pointer and Array Access Errors". In: *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*. PLDI '94. Orlando, Florida, USA: ACM, 1994, pp. 290–301. ISBN: 0-89791-662-X. DOI: 10.1145/178243.178446. URL: http://doi.acm.org/10.1145/178243.178446.

[14] N. Avonds, R. Strackx, P. Agten, and F. Piessens. "Salus: Non-hierarchical Memory Access Rights to Enforce the Principle of Least Privilege". English. In: *Security and Privacy in Communication Networks*. Ed. by T. Zia, A. Zomaya, V. Varadharajan, and M. Mao. Vol. 127. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Springer International Publishing, 2013, pp. 252–269.

[15] A. Azab, P. Ning, and X. Zhang. “SICE: a hardware-level strongly isolated computing environment for x86 multi-core platforms”. In: *Proceedings of the 18th ACM conference on Computer and communications security*. ACM. 2011, pp. 375–388. URL: http://www4.ncsu.edu/~amazab/SICE-CCS11.pdf.

[16] J. W. d. Bakker. *Mathematical Theory of Program Correctness*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1980. ISBN: 0135621321.

[17] I. G. Baltopoulos and A. D. Gordon. “Secure compilation of a multi-tier web language”. In: *Proceedings of the 4th international workshop on Types in language design and implementation*. TLDI ’09. Savannah, GA, USA: ACM, 2009, pp. 27–38. ISBN: 978-1-60558-420-1. DOI: http://doi.acm.org/10.1145/1481861.1481866. URL: http://doi.acm.org/10.1145/1481861.1481866.

[18] M. Barnett and W. Schulte. “Runtime Verification of .NET Contracts”. In: *J. Syst. Softw.* 65.3 (Mar. 2003), pp. 199–208. ISSN: 0164-1212. DOI: 10.1016/S0164-1212(02)00041-9. URL: http://dx.doi.org/10.1016/S0164-1212(02)00041-9.

[19] J. Berdine, C. Calcagno, and P. W. O'Hearn. “Smallfoot: Modular Automatic Assertion Checking with Separation Logic”. In: *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*. FMCO’05. Amsterdam, The Netherlands: Springer-Verlag, 2006, pp. 115–137. ISBN: 3-540-36749-7, 978-3-540-36749-9. DOI: 10.1007/11804192_6. URL: http://dx.doi.org/10.1007/11804192_6.

[20] J. Berdine, C. Calcagno, and P. W. O’Hearn. “Symbolic Execution with Separation Logic”. In: *Proceedings of the Third Asian Conference on Programming Languages and Systems*. APLAS’05. Tsukuba, Japan: Springer-Verlag, 2005, pp. 52–68. ISBN: 3-540-29735-9, 978-3-540-29735-2. DOI: 10.1007/11575467_5. URL: http://dx.doi.org/10.1007/11575467_5.

[21] R. Bornat. “Proving Pointer Programs in Hoare Logic”. In: *Proceedings of the 5th International Conference on Mathematics of Program Construction*. MPC ’00. London, UK, UK: Springer-Verlag, 2000, pp. 102–126. ISBN: 3-540-67727-5. URL: http://dl.acm.org/citation.cfm?id=648085.747307.

[22] J. Brotherston, R. Bornat, and C. Calcagno. “Cyclic Proofs of Program Termination in Separation Logic”. In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’08. San Francisco, California, USA: ACM, 2008, pp. 101–112. ISBN: 978-1-59593-689-9. DOI: 10.1145/1328438.1328453. URL: http://doi.acm.org/10.1145/1328438.1328453.

[23] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. "An Overview of JML Tools and Applications". In: *Int. J. Softw. Tools Technol. Transf.* 7.3 (June 2005), pp. 212–232. ISSN: 1433-2779. DOI: 10.1007/s10009-004-0167-4. URL: http://dx.doi.org/10.1007/s10009-004-0167-4.

[24] R. M. Burstall. "Some techniques for proving correctness of programs which alter data structures". In: *Machine Intelligence*. Vol. 7. 1972, pp. 23–50.

[25] C. Calcagno, D. Distefano, and V. Vafeiadis. "Bi-abductive Resource Invariant Synthesis". In: *Proceedings of the 7th Asian Symposium on Programming Languages and Systems*. APLAS '09. Seoul, Korea: Springer-Verlag, 2009, pp. 259–274. ISBN: 978-3-642-10671-2. DOI: 10.1007/978-3-642-10672-9_19. URL: http://dx.doi.org/10.1007/978-3-642-10672-9_19.

[26] W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. "Automated Verification of Shape, Size and Bag Properties via User-defined Predicates in Separation Logic". In: *Sci. Comput. Program.* 77.9 (Aug. 2012), pp. 1006–1036. ISSN: 0167-6423. DOI: 10.1016/j.scico.2010.07.004. URL: http://dx.doi.org/10.1016/j.scico.2010.07.004.

[27] E. M. Clarke Jr. "Programming Language Constructs for Which It Is Impossible To Obtain Good Hoare Axiom Systems". In: *J. ACM* 26.1 (Jan. 1979), pp. 129–147. ISSN: 0004-5411. DOI: 10.1145/322108.322121. URL: http://doi.acm.org/10.1145/322108.322121.

[28] M. R. Clarkson and F. B. Schneider. "Hyperproperties". In: *J. Comput. Secur.* 18.6 (Sept. 2010), pp. 1157–1210. ISSN: 0926-227X. URL: http://dl.acm.org/citation.cfm?id=1891823.1891830.

[29] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, T. Moskal Michałand Santen, W. Schulte, and S. Tobies. "VCC: A Practical System for Verifying Concurrent C". In: *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*. TPHOLs '09. Munich, Germany: Springer-Verlag, 2009, pp. 23–42. ISBN: 978-3-642-03358-2. DOI: 10.1007/978-3-642-03359-9_2. URL: http://dx.doi.org/10.1007/978-3-642-03359-9_2.

[30] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula. "Dependent Types for Low-level Programming". In: *Proceedings of the 16th European Conference on Programming*. ESOP'07. Braga, Portugal: Springer-Verlag, 2007, pp. 520–535. ISBN: 978-3-540-71314-2. URL: http://dl.acm.org/citation.cfm?id=1762174.1762221.

[31] B. Cook, A. Gotsman, A. Podelski, A. Rybalchenko, and M. Y. Vardi. "Proving That Programs Eventually Do Something Good". In: *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '07. Nice, France: ACM, 2007, pp. 265–276. ISBN: 1-59593-575-4. DOI: 10.1145/1190216.1190257. URL: http://doi.acm.org/10.1145/1190216.1190257.

[32] S. Cook. "Soundness and Completeness of an Axiom System for Program Verification". In: *SIAM Journal on Computing* 7.1 (1978), pp. 70–90. DOI: 10.1137/0207005. URL: http://dx.doi.org/10.1137/0207005.

[33] Coverity. *Coverity Scan: 2013 Open Source Report*. Tech. rep. Coverity, Inc., Apr. 2013. URL: http://softwareintegrity.coverity.com/rs/coverity/images/2013-Coverity-Scan-Report.pdf.

[34] P.-L. Curien. "Definability and Full Abstraction". In: *Electron. Notes Theor. Comput. Sci.* 172 (Apr. 2007), pp. 301–310. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2007.02.011. URL: http://dx.doi.org/10.1016/j.entcs.2007.02.011.

[35] M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte. "VCC: Contract-based modular verification of concurrent C". In: *31st International Conference on Software Engineering, ICSE 2009*. May 2009, pp. 429–430. DOI: 10.1109/ICSE-COMPANION.2009.5071046.

[36] L. De Moura and N. Bjørner. "Z3: An Efficient SMT Solver". In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'08/ETAPS'08. Budapest, Hungary: Springer-Verlag, 2008, pp. 337–340. ISBN: 3-540-78799-2, 978-3-540-78799-0. URL: http://dl.acm.org/citation.cfm?id=1792734.1792766.

[37] C. Dimoulas, R. B. Findler, C. Flanagan, and M. Felleisen. "Correct Blame for Contracts: No More Scapegoating". In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. Austin, Texas, USA: ACM, 2011, pp. 215–226. ISBN: 978-1-4503-0490-0. DOI: 10.1145/1926385.1926410. URL: http://doi.acm.org/10.1145/1926385.1926410.

[38] D. Distefano and M. J. Parkinson J. "jStar: Towards Practical Verification for Java". In: *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*. OOPSLA '08. Nashville, TN, USA: ACM, 2008, pp. 213–226. ISBN: 978-1-60558-215-3. DOI: 10.1145/1449764.1449782. URL: http://doi.acm.org/10.1145/1449764.1449782.

[39] P. Doughty-White and M. Quick. *Information is Beautiful - Codebases*. Nov. 2014. URL: http://www.informationisbeautiful.net/visualizations/million-lines-of-code/.

[40] K. El Defrawy, A. Francillon, D. Perito, and G. Tsudik. "SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust". In: *Proceedings of the Network & Distributed System Security Symposium (NDSS), San Diego, CA*. 2012. URL: http://francillon.net/~aurel/papers/2012_SMART.pdf.

[41] Ú. Erlingsson. "Low-level Software Security: Attacks and Defenses". In: *Foundations of Security Analysis and Design IV*. Ed. by A. Aldini and R. Gorrieri. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 92–134. ISBN: 3-540-74809-1, 978-3-540-74809-0. URL: http://dl.acm.org/citation.cfm?id=1793914.1793919.

[42] U. Erlingsson, Y. Younan, and F. Piessens. "Low-level software security by example". In: *Handbook of Information and Communication Security*. Springer, 2010, pp. 663–658. ISBN: 978-3-642-04116-7. URL: https://lirias.kuleuven.be/handle/123456789/267049.

[43] R. B. Findler and M. Felleisen. "Contract Soundness for Object-oriented Languages". In: *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '01. Tampa Bay, FL, USA: ACM, 2001, pp. 1–15. ISBN: 1-58113-335-9. DOI: 10.1145/504282.504283. URL: http://doi.acm.org/10.1145/504282.504283.

[44] L. Fix. "Fifteen Years of Formal Property Verification in Intel". In: *25 Years of Model Checking*. Ed. by O. Grumberg and H. Veith. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 139–144. ISBN: 978-3-540-69849-4. DOI: 10.1007/978-3-540-69850-0_8. URL: http://dx.doi.org/10.1007/978-3-540-69850-0_8.

[45] M. Flatt and PLT. *Reference: Racket*. Tech. rep. PLT-TR-2010-1. http://racket-lang.org/tr1/. PLT Inc., 2010.

[46] R. W. Floyd. "Assigning Meanings to Programs". In: *Proceedings of a Symposium on Applied Mathematics*. Ed. by J. T. Schwartz. Vol. 19. Mathematical Aspects of Computer Science. American Mathematical Society. Providence, 1967, pp. 19–31. URL: http://www.eecs.berkeley.edu/%C3%B1ecula/Papers/FloydMeaning.pdf.

[47] C. Fournet, N. Swamy, J. Chen, P.-E. Dagand, P.-Y. Strub, and B. Livshits. "Fully Abstract Compilation to JavaScript". In: *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '13. Rome, Italy: ACM, 2013, pp. 371–

384. ISBN: 978-1-4503-1832-7. DOI: 10.1145/2429069.2429114. URL: http://doi.acm.org/10.1145/2429069.2429114.

[48] *GHOST gethostbyname() heap overflow in glibc (CVE-2015-0235)*. National Vulnerability Database. Apr. 6, 2015. URL: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-0235.

[49] *GNU Bourne-Again Shell (Bash) 'Shellshock' Vulnerability (CVE-2014-6271)*. National Vulnerability Database. Sept. 24, 2014. URL: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-6271.

[50] G. A. Gorelick. *A complete axiomatic system for proving assertions about recursive and non-recursive programs*. Tech. rep. TR-75. Department of Computer Science, University of Toronto, 1975.

[51] M. Greenberg, B. C. Pierce, and S. Weirich. "Contracts Made Manifest". In: *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '10. Madrid, Spain: ACM, 2010, pp. 353–364. ISBN: 978-1-60558-479-9. DOI: 10.1145/1706299.1706341. URL: http://doi.acm.org/10.1145/1706299.1706341.

[52] J. Guo, P. Karpman, I. Nikolic, L. Wang, and S. Wu. *Analysis of BLAKE2*. Cryptology ePrint Archive, Report 2013/467. http://eprint.iacr.org/. 2013.

[53] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: http://doi.acm.org/10.1145/363235.363259.

[54] C. Hoare. "Procedures and parameters: An axiomatic approach". English. In: *Symposium on Semantics of Algorithmic Languages*. Ed. by E. Engeler. Vol. 188. Lecture Notes in Mathematics. Springer Berlin Heidelberg, 1971, pp. 102–116. ISBN: 978-3-540-05377-4. DOI: 10.1007/BFb0059696. URL: http://dx.doi.org/10.1007/BFb0059696.

[55] C. Hoare. "Towards a Theory of Parallel Programming". In: *Operating System Techniques*. Academic Press, 1971, pp. 61–71.

[56] Intel Corporation. *Software Guard Extensions Programming Reference*. Sept. 2013. URL: https://software.intel.com/sites/default/files/329298-001.pdf.

[57] S. S. Ishtiaq and P. W. O'Hearn. "BI As an Assertion Language for Mutable Data Structures". In: *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '01. London, United Kingdom: ACM, 2001, pp. 14–26. ISBN: 1-58113-336-7. DOI: 10.1145/360204.375719. URL: http://doi.acm.org/10.1145/360204.375719.

[58] B. Jacobs and F. Piessens. “Expressive Modular Fine-grained Concurrency Specification”. In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’11. Austin, Texas, USA: ACM, 2011, pp. 271–282. ISBN: 978-1-4503-0490-0. DOI: 10.1145/1926385.1926417. URL: http://doi.acm.org/10.1145/1926385.1926417.

[59] B. Jacobs and F. Piessens. *The VeriFast Program Verifier*. Tech. rep. Katholieke Universiteit Leuven, Aug. 2008. URL: http://www.cs.kuleuven.be/%5C~%7B%7Dbartj/verifast/verifast.pdf.

[60] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. “VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java”. In: *Proceedings of the Third International Conference on NASA Formal Methods*. NFM’11. Pasadena, CA: Springer-Verlag, 2011, pp. 41–55. ISBN: 978-3-642-20397-8. URL: http://dl.acm.org/citation.cfm?id=1986308.1986314.

[61] B. Jacobs, J. Smans, and F. Piessens. “A Quick Tour of the VeriFast Program Verifier”. In: *Proceedings of the 8th Asian Conference on Programming Languages and Systems*. APLAS’10. Shanghai, China: Springer-Verlag, 2010, pp. 304–311. ISBN: 3-642-17163-X, 978-3-642-17163-5. URL: http://dl.acm.org/citation.cfm?id=1947873.1947902.

[62] R. Jagadeesan, C. Pitcher, J. Rathke, and J. Riely. “Local Memory via Layout Randomization”. In: *Computer Security Foundations Symposium (CSF), 2011 IEEE 24th*. June 2011, pp. 161–174.

[63] A. Jeffrey and J. Rathke. “Java Jr: Fully Abstract Trace Semantics for a Core Java Language”. In: *ESOP*. 2005, pp. 423–438.

[64] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. “Cyclone: A Safe Dialect of C”. In: *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*. ATEC ’02. Berkeley, CA, USA: USENIX Association, 2002, pp. 275–288. ISBN: 1-880446-00-6. URL: http://dl.acm.org/citation.cfm?id=647057.713871.

[65] A. Kennedy. “Securing the .NET programming model”. In: *Theor. Comput. Sci.* 364.3 (2006), pp. 311–317. DOI: 10.1016/j.tcs.2006.08.014. URL: http://dx.doi.org/10.1016/j.tcs.2006.08.014.

[66] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. “seL4: Formal Verification of an OS Kernel”. In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP ’09. Big Sky, Montana, USA: ACM, 2009,

pp. 207–220. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629596. URL: http://doi.acm.org/10.1145/1629575.1629596.

[67] N. Kosmatov, G. Petiot, and J. Signoles. "An Optimized Memory Monitoring for Runtime Assertion Checking of C Programs". In: *Runtime Verification - 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings*. 2013, pp. 167–182. DOI: 10.1007/978-3-642-40787-1_10. URL: http://dx.doi.org/10.1007/978-3-642-40787-1_10.

[68] X. Leroy. "Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant". In: *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '06. Charleston, South Carolina, USA: ACM, 2006, pp. 42–54. ISBN: 1-59593-027-2. DOI: 10.1145/1111037.1111042. URL: http://doi.acm.org/10.1145/1111037.1111042.

[69] B. Liskov and S. Zilles. "Programming with abstract data types". In: *SIGPLAN Not.* 9.4 (Mar. 1974), pp. 50–59. ISSN: 0362-1340. DOI: 10.1145/942572.807045. URL: http://doi.acm.org/10.1145/942572.807045.

[70] J. Maerien, P. Agten, C. Huygens, and W. Joosen. "FAMoS: A Flexible Active Monitoring Service for Wireless Sensor Networks". In: *DAIS*. 2012, pp. 104–117.

[71] Z. Manna and A. Pnueli. *Axiomatic Approach to Total Correctness of Programs*. Tech. rep. Stanford, CA, USA: Stanford University, 1973.

[72] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. "TrustVisor: Efficient TCB Reduction and Attestation". In: *Proceedings of the IEEE Symposium on Security and Privacy*. May 2010. URL: http://www.ece.cmu.edu/~jmmccune/papers/MLQZDGP2010.pdf.

[73] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. "Flicker: An Execution Infrastructure for Tcb Minimization". In: *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*. Eurosys '08. Glasgow, Scotland UK: ACM, 2008, pp. 315–328. ISBN: 978-1-60558-013-5. DOI: 10.1145/1352592.1352625. URL: http://doi.acm.org/10.1145/1352592.1352625.

[74] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. "Flicker: An Execution Infrastructure for TCB Minimization". In: *Proceedings of the ACM European Conference in Computer Systems (EuroSys)*. ACM. Apr. 2008, pp. 315–328. URL: http://www.ece.cmu.edu/~jmmccune/papers/mccune_parno_perrig_reiter_isozaki_eurosys08.pdf.

[75] B. Meyer. “Applying "Design by Contract"”. In: *Computer* 25.10 (Oct. 1992), pp. 40–51. ISSN: 0018-9162. DOI: 10.1109/2.161279. URL: http://dx.doi.org/10.1109/2.161279.

[76] J. H. Morris. “Lambda-Calculus Models of Programming Languages”. PhD thesis. Massachusetts Institute of Technology, 1968.

[77] J. H. Morris Jr. “Protection in programming languages”. In: *Commun. ACM* 16.1 (Jan. 1973), pp. 15–21. ISSN: 0001-0782. DOI: 10.1145/361932.361937. URL: http://doi.acm.org/10.1145/361932.361937.

[78] J. Morris. “A General Axiom of Assignment”. English. In: *Theoretical Foundations of Programming Methodology*. Ed. by M. Broy and G. Schmidt. Vol. 91. NATO Advanced Study Institutes Series. Springer Netherlands, 1982, pp. 25–34. ISBN: 978-90-277-1462-6. DOI: 10.1007/978-94-009-7893-5_3. URL: http://dx.doi.org/10.1007/978-94-009-7893-5_3.

[79] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. “CCured: Type-safe Retrofitting of Legacy Software”. In: *ACM Trans. Program. Lang. Syst.* 27.3 (May 2005), pp. 477–526. ISSN: 0164-0925. DOI: 10.1145/1065887.1065892. URL: http://doi.acm.org/10.1145/1065887.1065892.

[80] G. C. Necula, S. McPeak, and W. Weimer. “CCured: Type-safe Retrofitting of Legacy Code”. In: *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’02. Portland, Oregon: ACM, 2002, pp. 128–139. ISBN: 1-58113-450-9. DOI: 10.1145/503272.503286. URL: http://doi.acm.org/10.1145/503272.503286.

[81] J. von Neumann. “First Draft of a Report on the EDVAC”. In: *IEEE Ann. Hist. Comput.* 15.4 (Oct. 1993), pp. 27–75. ISSN: 1058-6180. DOI: 10.1109/85.238389. URL: http://dx.doi.org/10.1109/85.238389.

[82] M. Newman. *Software Errors Cost U.S. Economy $59.5 Billion Annually*. Tech. rep. National Institute of Standards and Technology, Oct. 2002. URL: https://web.archive.org/web/20030210134224/http://www.nist.gov/public_affairs/releases/n02-10.htm.

[83] H. H. Nguyen, V. Kuncak, and W.-N. Chin. “Runtime Checking for Separation Logic”. In: *Proceedings of the 9th International Conference on Verification, Model Checking, and Abstract Interpretation*. VMCAI’08. San Francisco, USA: Springer-Verlag, 2008, pp. 203–217. ISBN: 3-540-78162-5, 978-3-540-78162-2. URL: http://dl.acm.org/citation.cfm?id=1787526.1787545.

[84] T. Nipkow. “Hoare Logics for Recursive Procedures and Unbounded Nondeterminism”. In: *Computer Science Logic (CSL 2002)*. Ed. by J. Bradfield. Vol. 2471. LNCS. Springer, 2002, pp. 103–119.

[85] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. “Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base”. In: *Proceedings of the 22nd USENIX Conference on Security*. SEC’13. Washington, D.C.: USENIX Association, 2013, pp. 479–494.

[86] L. O’Brien. *How Many Lines of Code in Windows?* Dec. 2005. URL: http://www.knowing.net/index.php/2005/12/06/how-many-lines-of-code-in-windows/.

[87] P. W. OHearn. “Resources, Concurrency, and Local Reasoning”. In: *Theor. Comput. Sci.* 375.1-3 (Apr. 2007), pp. 271–307. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2006.12.035. URL: http://dx.doi.org/10.1016/j.tcs.2006.12.035.

[88] P. W. O’Hearn, J. C. Reynolds, and H. Yang. “Local Reasoning About Programs That Alter Data Structures”. In: *Proceedings of the 15th International Workshop on Computer Science Logic*. CSL ’01. London, UK, UK: Springer-Verlag, 2001, pp. 1–19. ISBN: 3-540-42554-3. URL: http://dl.acm.org/citation.cfm?id=647851.737404.

[89] P. W. O’Hearn, H. Yang, and J. C. Reynolds. “Separation and Information Hiding”. In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’04. Venice, Italy: ACM, 2004, pp. 268–280. ISBN: 1-58113-729-X. DOI: 10.1145/964001.964024. URL: http://doi.acm.org/10.1145/964001.964024.

[90] D. von Oheimb. “Hoare Logic for Mutual Recursion and Local Variables”. English. In: *Foundations of Software Technology and Theoretical Computer Science*. Ed. by C. Rangan, V. Raman, and R. Ramanujam. Vol. 1738. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1999, pp. 168–180. ISBN: 978-3-540-66836-7. DOI: 10.1007/3-540-46691-6_13. URL: http://dx.doi.org/10.1007/3-540-46691-6_13.

[91] *OpenSSL ’Heartbleed’ vulnerability (CVE-2014-0160)*. National Vulnerability Database. Apr. 7, 2014. URL: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0160.

[92] S. Owicki and D. Gries. “Verifying Properties of Parallel Programs: An Axiomatic Approach”. In: *Commun. ACM* 19.5 (May 1976), pp. 279–285. ISSN: 0001-0782. DOI: 10.1145/360051.360224. URL: http://doi.acm.org/10.1145/360051.360224.

[93] M. Parkinson and G. Bierman. “Separation Logic and Abstraction”. In: *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’05. Long Beach, California, USA: ACM, 2005, pp. 247–258. ISBN: 1-58113-830-X. DOI: 10.1145/1040305.1040326. URL: http://doi.acm.org/10.1145/1040305.1040326.

[94] M. Patrignani. “The Tome of Secure Compilation: Fully Abstract Compilation to Protected Modules Architectures”. PhD thesis. KU Leuven, May 2015. URL: https://lirias.kuleuven.be/handle/123456789/494704.

[95] M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens. “Secure Compilation to Protected Module Architectures”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 37.2 (Apr. 2015), 6:1–6:50.

[96] M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens. “Secure Compilation to Protected Module Architectures”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 37.2 (Apr. 2015), 6:1–6:50. ISSN: 0164-0925. DOI: 10.1145/2699503. URL: http://doi.acm.org/10.1145/2699503.

[97] M. Patrignani and D. Clarke. “Fully Abstract Trace Semantics for Low-level Isolation Mechanisms”. In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. SAC ’14. Gyeongju, Republic of Korea: ACM, 2014, pp. 1562–1569. ISBN: 978-1-4503-2469-4. DOI: 10.1145/2554850.2554865. URL: http://doi.acm.org/10.1145/2554850.2554865.

[98] M. Patrignani and D. Clarke. “Fully abstract trace semantics for protected module architectures”. In: *Computer Languages, Systems & Structures* (2015). ISSN: 1477-8424. DOI: http://dx.doi.org/10.1016/j.cl.2015.03.002. URL: http://www.sciencedirect.com/science/article/pii/S1477842415000081.

[99] M. Patrignani, D. Clarke, and F. Piessens. “Secure compilation of Object-Oriented components to protected module architectures”. In: *LNCS*. Vol. 8301. Springer International Publishing, Dec. 2013, pp. 176–191. URL: https://lirias.kuleuven.be/handle/123456789/420864.

[100] P. Philippaerts, J. T. Mühlberg, W. Penninckx, J. Smans, B. Jacobs, and F. Piessens. “Software verification with VeriFast: Industrial case studies”. In: *Science of Computer Programming* 82.1 (Mar. 2014), pp. 77–97. URL: https://lirias.kuleuven.be/handle/123456789/388689.

[101] B. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[102] Pingdom. *10 historical software bugs with extreme consequences*. Pingdom Tech Blog. Apr. 6, 2015. URL: http://royal.pingdom.com/2009/03/19/10-historical-software-bugs-with-extreme-consequences/.

[103] R. Pucella and F. B. Schneider. "Independence From Obfuscation: A Semantic Framework for Diversity". In: *Computer Security Foundations Workshop, 2006. 19th IEEE*. 2006, pp. 230–241.

[104] J. C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures". In: *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*. LICS '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 55–74. ISBN: 0-7695-1483-9. URL: http://dl.acm.org/citation.cfm?id=645683.664578.

[105] J. C. Reynolds. *The Craft of Programming*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1981. ISBN: 0131888625.

[106] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. "Return-Oriented Programming: Systems, Languages, and Applications". In: *Trans. Info. & Sys. Sec.* (2011).

[107] D. P. Sahita R Warrier U. "Protecting Critical Applications on Mobile Platforms". In: *Intel Technology Journal* 13 (2 2009), pp. 16–35. URL: http://www.cs.unh.edu/~it666/reading_list/Hardware/intel_techjounral_security.pdf.

[108] C. Schlesinger, K. Pattabiraman, N. Swamy, D. Walker, and B. Zorn. "Modular Protections Against Non-control Data Attacks". In: *Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium*. CSF '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 131–145. ISBN: 978-0-7695-4365-9. DOI: 10.1109/CSF.2011.16. URL: http://dx.doi.org/10.1109/CSF.2011.16.

[109] T. Schreiber. "Auxiliary Variables and Recursive Procedures". In: *TAPSOFT '97*. Vol. 1214. LNCS. Springer-Verlag, 1997, pp. 697–711.

[110] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth. "Reducing TCB complexity for security-sensitive applications: three case studies". In: *EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*. Leuven, Belgium: ACM, 2006, pp. 161–174. ISBN: 1-59593-322-0. URL: http://www.cs.kuleuven.ac.be/conference/EuroSys2006/papers/p161-singaravelu.pdf.

[111] S. Sokolowski. "Axioms for total correctness". English. In: *Acta Informatica* 9.1 (1977), pp. 61–71. ISSN: 0001-5903. DOI: 10.1007/BF00263765. URL: http://dx.doi.org/10.1007/BF00263765.

[112] R. Strackx, P. Agten, N. Avonds, and F. Piessens. "Salus: Kernel Support for Secure Process Compartments". In: *EAI Endorsed Transactions on Security and Safety* 15.3 (Jan. 2015).

[113] R. Strackx and F. Piessens. “Fides: Selectively Hardening Software Application Components Against Kernel-level or Process-level Malware”. In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS ’12. Raleigh, North Carolina, USA: ACM, 2012, pp. 2–13. ISBN: 978-1-4503-1651-4. DOI: 10.1145/2382196.2382200. URL: http://doi.acm.org/10.1145/2382196.2382200.

[114] R. Strackx, F. Piessens, and B. Preneel. “Efficient Isolation of Trusted Subsystems in Embedded Systems”. English. In: *Security and Privacy in Communication Networks*. Ed. by S. Jajodia and J. Zhou. Vol. 50. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Springer Berlin Heidelberg, 2010, pp. 344–361. ISBN: 978-3-642-16160-5. DOI: 10.1007/978-3-642-16161-2_20. URL: http://dx.doi.org/10.1007/978-3-642-16161-2_20.

[115] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. “Breaking the memory secrecy assumption”. In: *EUROSEC*. 2009, pp. 1–8.

[116] M. Tatsuta, W.-N. Chin, and M. F. A. Ameen. “Completeness of Pointer Program Verification by Separation Logic”. In: *Proceedings of the 2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*. SEFM ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 179–188. ISBN: 978-0-7695-3870-9. DOI: 10.1109/SEFM.2009.33. URL: http://dx.doi.org/10.1109/SEFM.2009.33.

[117] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta. “Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework”. In: *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. SP ’13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 430–444. ISBN: 978-0-7695-4977-4. DOI: 10.1109/SP.2013.36. URL: http://dx.doi.org/10.1109/SP.2013.36.

[118] P. Vermeulen. *Gebruikers Betrekken*. Tech. rep. Pb7 Research, Oct. 2014. URL: http://www.mcspr.nl/wp-content/uploads/2014/10/rapport-+-aanbevelingen.pdf.

[119] F. Vogels. “Formalisation and Soundness of Static Verification Algorithms for Imperative Programs (Formalisatie en correctheid van statische verificatiealgoritmes voor imperatieve programma’s)”. Piessens, Frank and Jacobs, Bart (supervisors). PhD thesis. Informatics Section, Department of Computer Science, Faculty of Engineering Science, Dec. 2012, p. 482. URL: https://lirias.kuleuven.be/handle/123456789/362694.

[120] D. A. Wheeler. *SLOCCount*. Feb. 2001. URL: http://www.dwheeler.com/sloccount.

[121] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. “Formal Methods: Practice and Experience”. In: *ACM Comput. Surv.* 41.4 (Oct. 2009), 19:1–19:36. ISSN: 0360-0300. DOI: 10.1145/1592434.1592436. URL: http://doi.acm.org/10.1145/1592434.1592436.

[122] Y. Younan, W. Joosen, and F. Piessens. “Runtime Countermeasures for Code Injection Attacks Against C and C++ Programs”. In: *ACM Comput. Surv.* 44.3 (June 2012), 17:1–17:28. ISSN: 0360-0300. DOI: 10.1145/2187671.2187679. URL: http://doi.acm.org/10.1145/2187671.2187679.

[123] M. Zhivich and R. K. Cunningham. “The Real Cost of Software Errors.” In: *IEEE Security & Privacy* 7.2 (Dec. 6, 2009), pp. 87–90. URL: http://dblp.uni-trier.de/db/journals/ieeesp/ieeesp7.html#ZhivichC09.

List of publications

Articles in International Reviewed Journals

- R. Strackx, P. Agten, N. Avonds, F. Piessens. "Salus: Kernel Support for Secure Process Compartments". In: *EAI Endorsed Transactions on Security and Safety* 15.3 (Jan. 2015)
- M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, F. Piessens. "Secure Compilation to Protected Module Architectures". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 37.2 (Apr. 2015), 6:1–6:50

Contributions at International Conferences and Workshops, Published in Proceedings

- J. Maerien, P. Agten, C. Huygens, W. Joosen. "FAMoS: A Flexible Active Monitoring Service for Wireless Sensor Networks". In: *DAIS*. 2012, pp. 104–117
- P. Agten, R. Strackx, B. Jacobs, F. Piessens. "Secure Compilation to Modern Processors". In: *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th.* June 2012, pp. 171–185
- P. Agten, N. Nikiforakis, R. Strackx, W. De Groef, F. Piessens. "Recent Developments in Low-level Software Security". In: *Proceedings of the 6th IFIP WG 11.2 International Conference on Information Security Theory and Practice: Security, Privacy and Trust in Computing Systems and Ambient Intelligent Ecosystems.* WISTP'12. Egham, UK: Springer-Verlag, 2012, pp. 1–16

- P. Agten, S. Van Acker, Y. Brondsema, P. H. Phung, L. Desmet, F. Piessens. "JSand: complete client-side sandboxing of third-party JavaScript without browser modifications". In: *Proceedings of the 28th Annual Computer Security Applications Conference*. ACSAC '12. Orlando, Florida: ACM, 2012, pp. 1–10

- J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, F. Piessens. "Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base". In: *Proceedings of the 22nd USENIX Conference on Security*. SEC'13. Washington, D.C.: USENIX Association, 2013, pp. 479–494

- N. Avonds, R. Strackx, P. Agten, F. Piessens. "Salus: Non-hierarchical Memory Access Rights to Enforce the Principle of Least Privilege". English. In: *Security and Privacy in Communication Networks*. Ed. by T. Zia, A. Zomaya, V. Varadharajan, M. Mao. Vol. 127. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Springer International Publishing, 2013, pp. 252–269

- P. Agten, W. Joosen, F. Piessens, N. Nikiforakis. "Seven months' worth of mistakes: A longitudinal study of typosquatting abuse". In: *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS 2015)*. Internet Society, Feb. 2015

- P. Agten, B. Jacobs, F. Piessens. "Sound Modular Verification of C Code Executing in an Unverified Context". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '15. Mumbai, India: ACM, 2015, pp. 581–594

Technical Reports

- P. Agten, R. Strackx, B. Jacobs, F. Piessens. *Secure compilation to modern processors: extended version*. CW Reports CW619. Department of Computer Science, KU Leuven, Apr. 2012

- P. Agten, B. Jacobs, F. Piessens. *Sound modular verification of C code executing in an unverified context: extended version*. CW Reports CW676. Department of Computer Science, KU Leuven, Nov. 2014

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
IMINDS-DISTRINET
Celestijnenlaan 200A box 2402
B-3001 Heverlee
http://www.cs.kuleuven.be