KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

The Tome of Secure Compilation

Fully Abstract Compilation to Protected Modules Architectures

Marco Patrignani

Supervisor:
Prof. dr. D. Clarke
Prof. dr. ir. F. Piessens

Dissertation presented in partial fulfilment of the requirements for the degree of Doctor in Engineering

May 2015

The Tome of Secure Compilation

Fully Abstract Compilation
to Protected Modules Architectures

Marco PATRIGNANI

Dissertation presented in partial fulfilment of the requirements for the degree of Doctor in Engineering

May 2015

ISBN

D/

Preface

We do not stop playing because we grow old, we grow old because we stop playing.

Uncertain attribution.

Alas, the end! Well, of course not the end of the manuscript, my dear reader, I but meant the end of my Ph.D. journey, which means I have to thank those that played a part in it.

My advisors have been great. I started with just one, Dave Clarke, but as he moved to Sweden and I did not follow him, I got another for free: Frank Piessens! I've learnt a great deal from you both: what a funny space is, the difference between code and data, and also a bit of how to do research. I won't pretend it's been easy, it's been hell, especially the first years, but you've been there to motivate and support me, as well as to proofread countless drafts, for this, I thank you a bloody lot! Dave deserves extra thanks for the patience he had when dealing with me, as he discovered that I can be quite stubborn in the defence of my own ideas. That mustn't have been nice to endure, sorry boss; please keep writing nice recommendation letters for me!

My committee members: profs. Mads Dam, Bart Jacobs, Bart Preneel, Tamara Rezk and Dirk Vandermeulen also deserve a big thanks. They read this manuscript in great detail and grilled me for a long time to ensure I was fit for the title. Extra thanks to the chair, prof. Vandermeulen, for not letting the grilling last too long!

If I could have decided the order in which I had to write this chapter, I would have started with them: the ladies of the department. One often underestimates and under-appreciates the fine work that the people from the secretary (Liesbet Degent, Anne-sophie Putseys, Esther Renson, Marleen Somers, Karen Spruyt) and the ones from the project office (Katrien Janssen, Ghita Saevels, Annick

Vandijk) do. They are amazing. These years would have been even more hellish hadn't it been for you.

Of course, my fellow Distrinetters have also made my stay at the department very enjoyable. Luckily, I got quiet but funny office mates: Kristof, Nelson, Jose, Gowri and Klaas. The lunch people were a late addition to my working day, I have to admit I've been secretly hoping for quite some time to join them: Pieter, Jesper, Milica, Jan-tobias, Adriaan, Raoul, Mathy, Gitte and Fredric. The football guys proved that CS people are more athletic than it sounds: Antoine, Vladimir, Kristof, Dominique, Roel, Job, Jose, Jan, Gijs, Rinde, Alex, ... and the others. One day I'll reveal you my secret for scoring all the goals for a team in a match (and for being injured every other match). The atmosphere at Distrinet was amazing due to the magnificent work of prof. Wouter Joosen, I think all the social events, the DRADS and the staff meetings make this group an extraordinary one. Keep up the good work, you're doing an amazing job. Concerning DRADS (the research group annual meeting), special 'prost' go to Dominique, Jesper, Thomas, Klaas and Kristof (and to the people of Tongerlo).

Leuven was for me like a home for a number of additional reasons. The Blauwoc volleyball team was great. Thanks to: Vincent, Jacek, Sebbe, Jakob, Linda, Thomas, Kevin, Tibor, Martijn, Jeroen, Tom and Raf ... and the others, you've been a lot of fun! The physics friends have made many evening a very funny one: Thomas, Marketa, Ivan, Gergely, Jasna, Hiwa, Manisha, Simone, Riccardo, the nights we spent together were always great. I'm sure I forgot someone, in that case, I'm sorry, I'll get you a drink.

But of course, there is no home without family. During my doctorate I got married to my wonderful wife Sara and we got our first-born son Simone! Sara's understanding of me makes my life so much easier, as does Simone's amazing sleeping skills. I'm looking forward to the second one! My family in Italy (and London) also deserve a big thanks for the support of these years: thanks mummy, babbo and lucazzo/Lucio as well as the grandparents!

Last but not least, my D&D groups and the C5 people. What is life without roleplaying? Not much. So, thanks to Ben, Jo, Gergely and Jesper for the Leuven sessions. Thanks to Dade, Salpa, Delo e Lucio for the Ravenna sessions. May we never stop. Thanks to Mauri, Felian, Inqui, Ale, Simon, Mola sr., Pluto and all the rest of the C5 gang, long live Confrontation!

This work has been made possible with the financial support of the Research Foundation Flanders (FWO).

A final, big thanks goes to Deepak Garg and the PL and security group of the Max Planck Institute for Software Systems in Saarbrücken for giving me a Postdoc position.

Abstract

Mature manhood: that means to have rediscovered the seriousness one had as a child at play.

Friedrich Nietzsche - Beyond Good and Evil Part IV - Aphorism 94

A compiler is a complex software artefact that, among other things, translates programs written in a source-level language into programs written in a target-level one. Source-level programming languages often provide programmers with means to define and enforce security policies. However, this does not always hold for target languages, so compilers do not always have the means to translate secure source programs into secure target programs. When security policies are not preserved by a compiler during the language translation, the generated target-level code can be subject to malicious security-breaching attacks. When security *is* preserved by a compiler, however, source-level security policies cannot be violated in the generated target-level code. A compiler that preserves source-level security policies in the target-level programs it generates is called a *secure compiler.*

This thesis then presents a secure compiler from an object-oriented Java-like language to untyped assembly code extended with protected module architectures (*PMA*) – an isolation mechanism of modern processors. To prove that compiler secure, it is proven to be *fully abstract*, so it preserves (and reflects) the behaviour of the programs it compiles. Since the behaviour of programs captures also the security policies of that program, a fully abstract compiler is a secure one.

This thesis also studies the behaviour for untyped assembly code extended with *PMA* in terms of trace semantics. Two such trace semantics are developed in this thesis and they are both proven to be as precise as contextual equivalence, i.e., they are *fully abstract*. The use of the trace semantics is crucial in the proof of full abstraction of the compiler, as it both dictates the proof strategy and it simplifies the actual proofs.

Additionally, this thesis studies the security of compilers in a setting where multiple isolation modules can be created at the assembly level. As the compiler for a single *PMA* isolation module turns not to be secure in the multiple module setting, that compiler is extended, so that it can be proven to be secure when multiple isolation modules are considered.

The results of this thesis provide the foundations both for the development of secure compilers for *PMA*-enhanced code and for reasoning about it. A prototype implementation of the secure compiler exist, yet we expect future development in secure compilers for assembly extended with *PMA* to build from the contents of this thesis.

Beknopte samenvatting

As we say in Vlaanderen, I'll do it *tussen de soep en de patatten.*

prof. Frank Piessens.

Een compiler is een complex software-artefact dat onder andere programma's geschreven in een bronprogrammeertaal vertaalt naar programma's in een doeltaal. Bronprogrammeertalen voorzien dikwijls mogelijkheden voor programmeurs om beveiligingsregels te definiëren en af te dwingen. Dit geldt echter niet altijd voor doelprogrammeertalen, en bijgevolg kunnen compilers veilige bronprogramma's niet altijd vertalen naar veilige doelprogramma's. Wanneer de beveiligingsregels niet worden behouden door de compiler tijdens de vertaling, kan de gegenereerde doelcode het doelwit worden van kwaadaardige en beveiligingsovertredende aanvallen. Wanneer de beveiliging echter wel behouden wordt door de compiler, kunnen de beveiligingsregels van de broncode niet overtreden worden in de gegenereerde doelcode. Een compiler die de beveiligingsregels van de broncode bewaart in de gegenereerde doelcode, heet een veilige compiler.

Deze thesis presenteert een veilige compiler van een objectgeörienteerde Java-achtige taal naar ongetypeerde assemblycode uitgebreid met beveiligde module-architecturen (BMA) – een isolatiemechanisme dat voorkomt in moderne processors. Om de veiligheid van deze compiler te bewijzen, wordt er bewezen dat hij volledig abstract is, zodat hij het gedrag van de gecompileerde programma's bewaart (en reflecteert). Aangezien het gedrag van programma's ook de beveiligingsregels van dat programma omvat, is een volledig abstracte compiler een veilige compiler. Deze thesis bestudeert ook het gedrag van ongetypeerde assemblycode uitgebreid met BMA in termen van spoorsemantiek. Twee zulke spoorsemantieken worden ontwikkeld in deze thesis en van beide wordt bewezen dat ze even precies zijn als contextuele equivalentie, m.a.w. dat ze volledig abstract zijn. Het gebruik van spoorsemantiek is cruciaal in het bewijs van de volledige abstractie van de compiler, aangezien het zowel de bewijsstrategie dicteert als de bewijzen zelf vereenvoudigd.

Hiernaast bestudeert deze thesis de veiligheid van compilers in een omgeving waar meerdere isolatiemodules op het assemblyniveau gecreëerd kunnen worden. Aangezien de compiler voor een enkele BMA-isolatiemodule niet veilig blijkt te zijn in een omgeving met meerdere modules, wordt deze compiler uitgebreid, zodat hij veilig bewezen kan worden wanneer meerdere isolatiemodules in overweging genomen worden.

De resultaten van deze thesis verstrekken de basis om zowel veilige compilers voor BMA-versterkte code te ontwikkelen als om hierover te redeneren. Er bestaat een prototype-implementatie van de veilige compiler, en bovendien verwachten we dat toekomstige ontwikkelingen van veilige compilers voor assemblycode uitgebreid met BMA verder bouwen op de inhoud van deze thesis.

Contents

List of Figures

List of Listings

Chapter 1

Introduction

C'mon newcomer, follow me!

Barrett, FF VII

Software has become a crucial part of our everyday lives; we use it in our entertainment systems, to manage our vehicles, to manage our finances and for so much more. Software is written by programmers using high-level languages that let them reason about their code due to a high degree of readability. High-level languages provide, for example, explicit conditional constructs implementing if-then-else choices, named modules and functions, so that programmers can tell more easily what the program will do. However, when they are run, programs are (possibly) compiled to assembly code. Writing assembly code is tedious and error prone and it is a much obsolete and almost completely abandoned practice. What happens in practice is that programmers write human-readable code in some high-level language and then this code gets translated by a compiler to low-level assembly code. Finally, the assembly code gets executed on the processor.

The piece of software that does the translation between the program input by the programmer (called the source program) and the program to execute on the processor (called the target program) is called *compiler* [14,16]. A compiler is a crucial piece of software for programmers: only a correct compiler can ensure that the program will run according to what they have written.[1] Since they are so crucial, compilers have been subject to a vast body of research. Moreover, the duties of a compiler have increased, from mere language translation to include

[1] Assuming they wrote what they had in mind, which is a non-trivial assumption.

lexing, parsing, program analysis, code optimisation and so forth. However, for the remainder of this thesis, we will consider compilers as just being language translators; as we will see, this task alone contains several issues to be tackled.

The concern with language translation is that high-level source languages offer security features to programmers in the form of type systems, module systems, encapsulation primitives and so forth. Unfortunately, most target languages do not offer the same security features as high-level source languages, and, certainly, plain untyped assembly language such as that run in commodity computers does not. That is (part of the) reason of some of the most severe cyber-crimes of the last years, including the 2011 Sony PlayStation Network outage[2] where the data of 77 million accounts was compromised or the theft of 1 billion dollars suffered by banks in 2015 in over 30 countries.[3] The concern is that often, what the programmer believes to be a secure program may be insecure once it is compiled. This difference in security abstractions is very attractive for malicious programmers that want to attack and violate security vulnerabilities of programs. This kind of programmer (which we refer to with the term “attackers”) exploits bugs in the complex software that manage our computers to steal data, perform robberies and all sorts of other crimes. Their modus operandi is often to infect a computer with their own assembly code, which then interacts with what the programmer believed to be secure code, bypassing and violating its security policies.

The first problem considered in this thesis is the following: how can we ensure that what is a secure high-level program is still secure when it is compiled and it runs? To address this concern, a number of alternatives exist, for example, software verification [37, 61], software monitoring [21, 57, 108] and secure compilation [6, 19, 45, 94]. This thesis is concerned with the last one. So the countermeasure to the aforementioned problem that we study in this text is the adoption of a secure compilation scheme. We use the term “compilation scheme” to indicate the guidelines explaining how a compiler is developed, as opposed to the term “compiler” that indicates the tool that is developed. Albeit relevant both from research and practice perspective, the aforementioned different kinds of countermeasures are not considered in this thesis.

The second problem considered in this thesis comes from the ever increasing compartmentalisation of software. A common software practice is to develop programs in compartments: i.e., programs that rely on the functionality of other programs for accomplishing the main service they are built for. These compartments are called packages or modules in real-world programming languages. For example, a shopping application may rely on a PDF viewing

[2] http://en.wikipedia.org/wiki/2011_PlayStation_Network_outage
[3] http://www.bbc.com/news/business-31482985

package to display invoices and on a printing package to print them. The shopping application may be written by a programmer who is different from the one who writes the PDF viewing package, who may be still different from the one who writes the printing package. The complication that arises in this setting is that these programmers can have a varying degree of trust among each other. For example, the shopping application programmer may not trust the PDF viewing package one while it trusts the printing package one. To abstract from the notion of programmer, in the following we will use the term *principal* to denote an entity that develops code.

The problem that arises now is: how can we securely compile the programs of different principals, so that trust assumptions are not violated? The threats that a compiler faces in this setting are more than the threats that arise when a single software developer is considered, and this thesis presents them all. To answer the question above, this thesis presents a secure compilation scheme from a source language that models different software packages belonging to different principals. Principals can specify the varying degree of trust between them. The secure compilation scheme ensures that a malicious attacker operating at the target level cannot violate the security assumptions of source-level code (e.g., by impersonating one of the principals).

Since secure compilation schemes are the keystone of this thesis, let us now informally explain why are they a correct solution for the presented problems. A compilation scheme is secure if it produces target-level programs that are as secure as their source-level counterparts. A secure program is one that enjoys security properties that can be expressed by means of program equivalence, including confidentiality, integrity and memory allocation properties as Example 4 describes in Section 2.5.1.[4] With a secure compilation scheme, the security properties that a programmer defined in source programs cannot be violated by attackers operating at the target language level. By choosing the target language to be assembly code, a secure compilation scheme ensures that the software running in a computer cannot be compromised by other software.[5] The only way to attack software that is securely compiled to assembly code is to gain physical access to the computer and then operate at hardware level, e.g., by taking cool dumps of the memory [56] or by probing it [53, 68, 121]. This level of security is very high. For all companies that own and physically restrict access to their machines (e.g., cloud service providers), a correctly-implemented secure compilers is a crucial building block for achieving a secure business.

Commodity computers run untyped assembly languages, which offers very little security guarantees for the compiler to build upon. To overcome this

[4] Other definitions of secure program exist, but this is the one used in this thesis, as clarified by Definition 1 in Section 2.5.1.

[5] Assuming that the compiler is bug-free, which is a non trivial assumption.

weakness, the secure compilation scheme developed in this thesis relies on an emerging security architecture, which is called Protected Module Architecture (*PMA*) [40, 80, 81, 82, 91, 112, 123]. *PMA* provides isolation facilities at the level of assembly code (in certain implementations), which is a very powerful security feature on its own. Informally, *PMA* partitions the memory into isolated modules, each with its code and data that are only visible to the module itself (*PMA* will be explained in more detail later on, in Section 2.1). Moreover, each module defines special addresses that other code can jump to in order to allow interoperation between modules. *PMA* is a very powerful security architecture, and a legitimate question at this point would be: do we even need a secure compiler once we have *PMA*? The answer to this question is "yes", as this thesis will demonstrate (Section 6.1 and Section 8.2). There are in fact several cases where a naïve implementation of a compiler to *PMA* would produce insecure code starting from a secure source program, and this thesis highlights them.

After pointing out such cases, this thesis describes how to implement a compiler that is secure and unaffected by those security violations. This thesis provides a proof that implementing a compiler according to the guidelines of the compilation scheme is secure. Such a proof can be developed according to different theories, which will be discussed in this thesis (Section 9.4).

To demonstrate that the compilation scheme devised in this thesis is secure, we will prove it to be fully abstract [1]. Informally, a compilation scheme is fully abstract when it translates indistinguishable source-level programs into indistinguishable target-level programs. Two programs are indistinguishable when their behaviour is the same no matter what program they interact with (in Section 2.5 we will formulate this property with more formal precision). For the remainder of this thesis, we will state that two programs are indistinguishable if they are equivalent. For example, consider two instances P_1 and P_2 of the same program (written in a language $\mathcal{S}$) that contain two different values in the same variable v. Denote their fully abstractly-compiled counterparts to a target language $\mathcal{T}$ by $[\![P_1]\!]^{\mathcal{S}}_{\mathcal{T}}$ and $[\![P_2]\!]^{\mathcal{S}}_{\mathcal{T}}$. If P_1 and P_2 are equivalent, then $[\![P_1]\!]^{\mathcal{S}}_{\mathcal{T}}$ and $[\![P_2]\!]^{\mathcal{S}}_{\mathcal{T}}$ must also be. So, if the content of v is *confidential* (e.g., the value stored in v could be **`private`** and never communicated), no program interacting with $[\![P_1]\!]^{\mathcal{S}}_{\mathcal{T}}$ or $[\![P_2]\!]^{\mathcal{S}}_{\mathcal{T}}$ can discern it.

The formal definition of a fully abstract compiler is that it preserves and reflects program equivalence between source and target-level programs. The security aspect of a fully abstract compiler is, however, only entailed by the reflection of program equivalence. Preservation of program equivalence matters for functional correctness. The preservation property holds for compilers that are correct: those that do not introduce faults in the translation, e.g., by translating an expression `1 + 1` into `1 + 2`. As the focus of this thesis is security, most of its efforts are devoted to proving the reflection property (e.g., Chapter 7), while preservation follows directly from assuming that a compiler is correct (Section 6.2 and Property 1 in Chapter 7).

Note that a fully abstract compilation scheme does not eliminate source-level security flaws. It is, in a sense, conservative, as it introduces no more vulnerabilities at the target-level than the ones already exploitable at the source level. Thus, it is important that the source language for the secure compiler has a way to specify security policies.

In this thesis, the source language for the secure compilation scheme will be an object-oriented language called J+E that only allows **private** fields. J+E is an extension of Jeffrey and Rathke's Java Jr. with exceptions and local variables [65]. Having only **private** fields, J+E allows the programmer to define confidentiality, integrity and invariants in the code (as exemplified later in this thesis in Section 2.5.1). For reasoning about equality of J+E programs, the J+E language is formalised in its syntax, static and dynamic semantics. Then, the J+E language is extended to consider multiple principals and the trust relation between them. This extension results in a new language Jem which is also formalised in its syntax and semantics, though most of its formalisation relies on the one of J+E.

The compilation scheme devised in this thesis targets a language called A+I: untyped assembly extended with *PMA*. The formalisation of the A+I language provides its syntax and dynamic semantics, though reasoning about program equivalence with an untyped assembly language is a notoriously complex task. To ease this task, A+I programs are given a trace semantics that indicates what is happening in the code as a sequence of actions such as "function call" or "return". Are we free to change the semantics used for a language, though? Yes, but only if that semantics is proven to be as precise as the operational semantics [99]. Once such a result is established, the trace semantics is called *fully abstract*.[6] Like all semantics, the trace semantics yields a notion of program equivalence called trace equivalence. Trace equivalence is a much simpler equivalence to use than the one yielded by the operational semantics of A+I. Moreover, it simplifies the proof that the compilation scheme from J+E to A+I is secure. As for the J+E case, A+I is also extended to support compilation of Jem and this results in a new language Aim. While A+I only considered a single instance of the isolation facilities provided by *PMA*, Aim considers multiple ones.

The breakthrough of the *PMA* security architecture (which now has an industrial implementation in the Intel SGX [15, 82]), hints at a tougher world, in the future, for attackers. The idea that leads the development of these architectures

[6]To avoid confusion, note that the term *fully abstract* is thus used for two concepts in this thesis: compilation and trace semantics.

is that we need more secure software for managing our everyday lives. The results of this thesis are that in order to have truly secure software, we need to securely compile it. This thesis lays the foundations for the development of such truly secure software by discussing how to develop secure compilers. Moreover it proves that those foundations are strong enough to build upon since the discussed secure compilers are indeed proven to be secure.

Alas, adopting a secure compiler is not easy and a number of downsides are foreseen before secure compilation will become a common practice. Firstly, adopting *PMA* has downsides of its own. In fact, the software architecture of our computers needs to be changed to adapt to the new security architecture. Sadly, we know how most industry is resilient to change. Additionally, the development of a fully-fledged secure compiler for an industry-adoptable programming language is a non-trivial task for future programmers to get right and bug-free. Many compilers are known to be buggy unless they are formally certified to be correct. Such a certification would be needed for a secure compiler as well, and that will require a significant amount of work and research.

1.1 Contributions

This thesis presents three main contributions.

The first contribution is the formalisation of the *PMA* architecture as untyped assembly code (Chapter 4) and the extension of its semantics with a fully abstract trace semantics (Chapter 5). As mentioned, a fully abstract trace semantics simplifies the proof that the compilation scheme form J+E to A+I is fully abstract.

The second contribution is the development of a secure compilation scheme from J+E to A+I (Chapter 6). This secure compilation scheme is accompanied by an exhaustive list of naïve compiler implementations that would make it insecure. To prove that this compilation scheme is secure, it is proven to be fully abstract (Chapter 7).

Finally, the last contribution of the thesis is the development of a secure compilation scheme for multi-principal software from Jem to Aim (Chapter 8). To prove this compilation scheme secure, we provide a novel definition of multi-principal full abstraction. Multi-principal full abstraction is an adaptation of the definition of full abstraction that considers principals and trust degrees among them. Besides providing this definition, we argue both the correctness of the definition and that the compilation scheme from Jem to Aim satisfies it, i.e., it is secure.

1.2 Outline

This thesis is organised as follows.

Chapter 2: Background Notions. This chapter presents background notions that following chapters build upon. Firstly, it presents the *PMA* security architecture, followed by an informal description of the languages of the first secure compilation scheme (A+I and J+E). This chapter then presents the threat model considered for the first secure compilation scheme. Moreover, it introduces the formal tools used for proving the compilation scheme secure. This chapter discusses program equivalence in the form of contextual and trace equivalence. Then it presents fully abstract compilation and how the compilation scheme from J+E to A+I will be proven to be fully abstract.

Chapter 3: Formalisation of the Source Language J+E. This chapter formalises J+E, the source language of the secure compilation scheme. It presents its syntax, static and dynamic semantics.

Chapter 4: Formalisation of the Target Language A+I. This chapter formalises A+I, the target language of the secure compilation scheme. It describes the syntax of A+I and its dynamic semantics (no static semantics is presented as A+I is untyped). Moreover, it encodes the access control policy of *PMA* that the dynamic semantics relies upon.

Chapter 5: Fully Abstract Trace Semantics for A+I. This chapter presents two trace semantics for A+I programs. The first one, $\mathsf{Traces}^{\mathsf{S}}_{\mathsf{A+I}}$, is the simpler one; it models the behaviour of compiled J+E code and it simplifies the proof of secure compilation. The second one, $\mathsf{Traces}^{\mathsf{L}}_{\mathsf{A+I}}$, is more complex and it serves for reasoning about arbitrary A+I programs. Before the semantics are formalised, this chapter discusses which omissions in the trace semantics formalisation do not make it fully abstract. Afterwards, both semantics are proven to be fully abstract.

Chapter 6: A Secure Compiler from J+E to A+I. This chapter describes the secure compilation scheme from J+E to A+I. Firstly, it describes the security violations that a naïve implementation of a compiler from J+E to A+I can suffer. Then, it splits J+E into fragments and it describes how to securely compile each of them: outcalls, dynamic memory allocation and exceptions. Finally, this chapter discusses how to securely compile language features that are not present in J+E.

Chapter 7: Proof of Full Abstraction for $[\![\cdot]\!]^{\mathsf{J+E}}_{\mathsf{A+I}}$. This chapter proves that the compilation scheme of Chapter 6 is fully abstract. It firstly presents the algorithm that is used in one of the theorems, followed by theorem statements and their proofs.

Chapter 8: A Secure Compiler for Multi-Principal Languages. This chapter describes the secure compilation scheme for multi-principal languages. It firstly presents how to turn J+E and A+I into Jem and Aim. The former modification is accomplished by introducing principal annotations, the latter is accomplished by modelling *PMA* with multiple isolated modules. Then, this chapter presents the definition of multi-principal full abstraction and argues its correctness. Before presenting the secure compilation scheme, the security violations arising from a naïve compiler implementation are discussed. Finally, the languages of the secure compilation scheme are formalised and the chapter argues that it is secure.

Chapter 9: Evaluation and Discussion. This chapter discusses and evaluates the presented results. It provides benchmarks evaluating the computational overhead of the secure compilation scheme from J+E to A+I. Moreover, it presents how that secure compilation scheme can scale to the Intel SGX, the first industrial prototype for *PMA*. Finally, this chapter discusses limitations of this work and alterntive formulations of secure compilation.

Chapter 10: Related Work. This chapter presents related work on the areas of security architectures, compilers, secure compilation and fully abstract semantics. Since the focus of this thesis is on secure compilation, that related work section is surveyed in greater detail.

Chapter 11: Conclusion and Future Work. This chapter presents future work and concludes.

1.3 Related Publications

The contents of Chapter 2, 6 and 7, as well as part of the contents of Chapter 9 are taken from:

- Patrignani, M., and Clarke, D. Fully Abstract Trace Semantics for Protected Module Architectures. *Computer Languages, Systems & Structures* (2015). Special issue on the Programming Languages track at the 29th ACM Symposium on Applied Computing.
- Patrignani, M., Clarke, D., and Piessens, F. Secure Compilation of Object-Oriented Components to Protected Module Architectures. In *Proceedings of the 11th Asian Symposium on Programming Languages and Systems (APLAS'13)* (2013), vol. 8301 of *LNCS*, pp. 176–191.

The contents of Chapter 4 and 5 are taken from:

- Patrignani, M., and Clarke, D. Fully Abstract Trace Semantics of Low-level Isolation Mechanisms. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing* (2014), SAC '14, ACM, pp. 1562–1569.
- Patrignani, M., and Clarke, D. Fully Abstract Trace Semantics for Protected Module Architectures. In *Computer Languages, Systems & Structures*, 2015. Special issue on the Programming Languages track at the 27th {ACM} Symposium on Applied Computing.

The contents of Chapter 3 are taken from:

- Patrignani, M., Clarke, D., and Piessens, F. Secure Compilation of Object-Oriented Components to Protected Module Architectures – Extended Version. CW Reports CW646, Dept. of Computer Science, K.U.Leuven, September 2013.

The contents of Chapter 8, as well as the contents of Section 10.3 are based on work yet to be published.

Chapter 2

Background Notions

Quelli che s'innamorano di pratica, sanza scienza, son come 'l nocchiere, ch'entra in navilio sanza timone o bussola, che mai ha certezza dove si vada.

Those who fall in love with practice without science are like a helmsman sailing without rudder nor compass, who is never sure about where he is going.

Leonardo da Vinci

This chapter provides the background notions that the remainder of this work relies upon. Firstly, it presents Protected Module Architectures and their access control policy (Section 2.1). Then, it informally describes the target (Section 2.2) and the source language (Section 2.3) of the secure compiler developed later in this work. This chapter then discusses the threat model (Section 2.4). It also describes how program equivalences can be used to express security properties (Section 2.5) and fully abstract compilation (Section 2.6). Finally, this chapter presents how the proof of full abstraction of the compilation scheme will be carried out (Section 2.7).

This work assumes the reader is familiar with certain basic programming language notions such as program, state, semantics, types etc. For a gentle and superb introduction to these concepts, we refer to the book "Types and

Programming Languages" by Benjamin Pierce [98].

2.1 Protected Module Architectures

PMA provides assembly code with the ability to create a protected module. This protected module is a secure environment for code that needs to be protected from a potentially malicious surrounding environment. Like their high-level counterpart (e.g., ML modules) a protected module offers an interface mechanism to allow interoperation with code that resides outside of the module. Additionally, these modules *isolate* what is placed within the module boundaries.

Figure 2.1 contains a graphical representation of how the memory is affected by the adoption of *PMA*. The memory space (dash-contoured area) represents

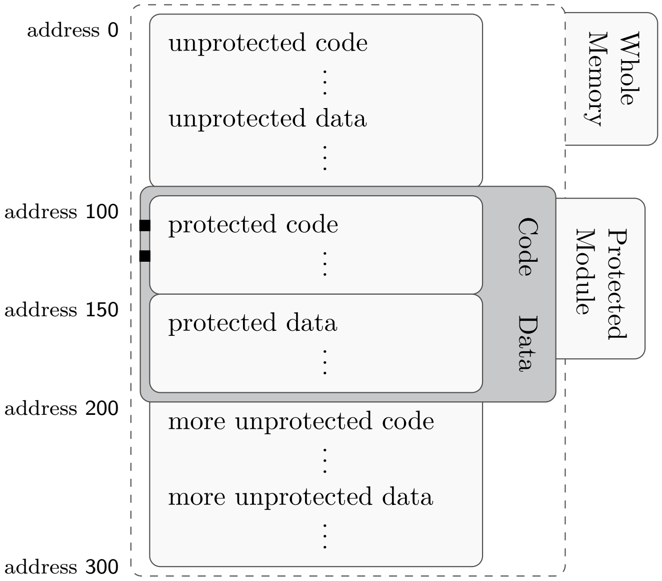

Figure 2.1: Graphical representation of the *PMA* architecture.

the whole address space available to a computer; for the sake of simplicity, we assume it to range from address 0 to address 300. The memory space is split into a protected (dark grey) and an unprotected region (very light grey); the protected region is the protected module and it spans from address 100 to address 200. The protected module is further split into a code and a data section (each ranging for 50 addresses in the example), which are both inaccessible from unprotected code. The only address where unprotected code can jump to are called *entry points* (denoted with ■), they are specific addresses in the code section of the protected module.

The most common way to implement *PMA* is through program counter-based memory access control mechanisms [40, 80, 81, 82, 91, 109, 111, 112, 123]. We review this mechanism from the work of Strackx and Piessens [112]. Informally, this mechanism introduces the same logical division of memory as described above into a protected and an unprotected section. The protected section is further divided into a code and a data section and a number of addresses in the code section are defined to be entry points. Then, based on the location of the program counter, the following access control policy is enforced. The only protected addresses to which instructions in unprotected memory can jump and execute are entry points, all the other addresses of the protected module are inaccessible from unprotected memory. The code section cannot be written, the data section cannot be executed and it is accessible only from the protected section. The size and location of each memory section are specified in a memory descriptor. Table 2.1 summarises the access control model enforced by the protection mechanism. There, indicate read permission with an ‘r’, write permission with a ‘w’ and execution permission with an ‘x’.

Table 2.1: Access control policy of *PMA*.

From\ To	Protected			Unprotected
	Entry Point	Code	Data	
Protected	r x	r x	r w	r w x
Unprotected	x			r w x

In 2013, Intel publicly announced Software Guard Extensions (Intel SGX), a hardware implementation of a Protected Module Architecture [15, 82]. Hence *PMA* support will be broadly available in mainstream processors within a few years. Any processor with *PMA* support can be targeted by secure compilers developed according to the techniques proposed in this thesis.

2.2 The Target Language A+I, Informally

To model a realistic compilation scheme, the target language should be close to what is used by modern processors. For this reason this paper adopts A+I (acronym of $\mathcal{A}$ssembly plus Isolation), a low-level language that models an idealised von Neumann machine enhanced with a protected module architecture [10, 94, 95, 97]. A detailed formalisation of A+I is presented in Chapter 4. This section but gives an informal description of A+I to familiarise the reader with it.

The protection mechanism affects the semantics of the language, preventing the execution of certain instructions, in accordance with the PMA access control policy presented in Section 2.1. Following are some code snippets that exemplify the semantics of A+I. In all examples concerning A+I code, assume the presence of a single protected memory section spanning from address 100 to 200, with a *single* entry point at address 100. Let P_s denote the code located in the protected section and P_u denote the code located in the unprotected one. Each instruction in the code snippets is preceded by the address where it is located; execution starts at address 0.

Example 1 (No execution of code in the protected memory partition) *P_u initialises register* $\mathtt{r}_0$ *to 101 (line 1) and then jumps to that address (line 2).*

```
1 0   movi r0 101  // unprotected code
2 1   jmp r0
3 ...
4 100   add r0 r1  // protected code
5 101   ret
```

Since address 101 is not an entry point of the protected memory section, the jump of P_u does not succeed as it is violating the PMA access control policy. ⊡

When the *PMA* access control policy is violated, different implementations react differently. For the rest of this thesis, consider that the execution is halted when the *PMA* access control policy is violated. Halting can also happen in a variety of ways, for example, in our prototype detailed in Section 9.1.1, the execution is suspended and trapped by the hypervisor [112].

Example 2 (No reading/writing the protected code section) *P_u initialises register* $\mathtt{r}_0$ *to 101 (line 1) and register* $\mathtt{r}_1$ *to 20 (line 2), then it writes the content of* $\mathtt{r}_1$ *at the address in* $\mathtt{r}_0$ *(line 3).*

```
1 0   movi r0 101  // unprotected code
2 1   movi r1 20
3 2   movs r0 r1
4 ...
5 100   add r0 r1    // protected code
6 101   ret
```

Since address 101 is protected, P_u cannot write there, so execution is `halt`*ed, as in Example 1. Analogously, if the instruction of line 2 were replaced with* `movl` $\mathtt{r}_0$ $\mathtt{r}_1$, *the execution is halted. In that case, P_u would be attempting to read the protected memory section, while it does not have that privilege.* $\boxdot$

Example 3 (Interoperation between protected and unprotected code) *P_u initialises register $\mathtt{r}_0$ to 12 (line 1), register $\mathtt{r}_1$ to 10 (line 2), register $\mathtt{r}_5$ to 100 (line 3) and then calls to the protected function located at address 100 (line 4), storing address 4 on the call stack. P_s subtracts registers $\mathtt{r}_0$ and $\mathtt{r}_1$ (line 6) and, if the result is greater than or equal to zero, it returns that result (line 9). Otherwise if the result is less than zero, P_s jumps to address 104 (lines 7,8), and returns 0 (lines 10, 11). Execution then continues in unprotected memory at address 4 (line 5, omitted), which is the address popped from the call stack.*

```
 1  0   movi r0 12   // unprotected code
 2  1   movi r1 10
 3  2   movi r5 100
 4  3   call r5
 5  ...
 6  100  sub r0 r1   // protected code
 7  101  movi r3 104
 8  102  jl r3
 9  103  ret
10  104  movi r0 0
11  105  ret
```

$\boxdot$

2.3 The Source Language J+E, Informally

The source-level language adopted by the compilation scheme is J+E (acronym of Java plus Encapsulation): a strongly-typed, single-threaded, component-based, object-oriented language that enforces `private` fields and `public` methods. J+E extends the Java Jr. language of Jeffrey and Rathke [65] with local variables and exceptions. It was chosen since it provides a clear notion of encapsulation for a source-level component, which makes for simpler reasoning about the secure compilation scheme. In fact, J+E programs are collections of components, which are collections of packages themselves. For the remainder of this work, the focus will be on components rather than on programs.

J+E partitions packages into *import* and *export* ones. Import packages are analogous to the `.h` header file of a C program.[1] They define *interfaces*, which

[1]The kind of C programs one writes when learning C: devoid of preprocessor instructions, macros etc.

are named collections of method signatures, and *externs*, which are references to externally defined objects. Export packages provide an implementation of an import package. They define *classes*, which are named collections of method implementations and fields (also known as instance variables), and *objects*, which provide implementations of classes and bindings from fields to values.

Listing 2.1 illustrates the package system of J+E. There, and in future code examples, we will massage the syntax of method bodies in the presented examples for the sake of readability. It contains two package declarations:

```
package P-Import;
  interface Account {
    public createAccount() : Account;
    public getBalance() : Int;
  }
  extern extAccount : Account;

package P-Export;
  class AccountClass implements P-Import.Account {

    AccountClass() {
      this.counter = 0;
    }
    private counter : Int;

    public createAccount() : P-Import.Account {
      return new P-Export.AccountClass();
    }
    public getBalance() : Int {
      return this.counter;
    }
    public addAmount( arg : Int ) : Unit {
      this.counter + = arg;
    }
  }
  object extAccount : AccountClass { private counter = 0 }
```

Listing 2.1: Example of the package system of J+E.

`P-Import` is an import package and `P-Export` is an export package implementing `P-Import`. `P-Export` provides class `AccountClass` that implements interface `Account` defined in `P-Import`. Object `extAccount` allocated in `P-Export` provides an implementation for the **`extern`** with the same name defined in `P-Import`.

One of the security mechanisms of J+E is given by `private` fields. Since they are not accessible from outside the class declaring them (as described in Section 2.5.1), they can be used to define security properties such as confidentiality and integrity (Definitions 2 to 3 in Section 2.4). In J+E, classes are private to the package that contains their declarations. Objects are allocated in the same package as the class they instantiate. Due to this package system, compiling a package only needs the import packages of any package it depends on. As a result, formal parameters in methods have interface types, since classes that implement those interfaces are unknown. This discipline is called: *programming to an interface*. While this discipline does not restrict the expressiveness of a language, it enforces a programming pattern on programmers, who have a more fine-grained control of how the classes they implement can be instantiated. In fact, since constructors are not exposed in interfaces, cross-package object allocation happens by using factory methods, i.e., methods to which the creation of an object is delegated (as opposed to using the **`new`** expression, which immediately allocates an object) [47]. For example, the name of class `AccountClass` from Listing 2.1 is not visible from outside package `P-Export`. Thus expressions of the form **`new`** `P-Export.AccountClass()` cannot be written outside `P-Export`. Instead, to allocate a `P-Export.AccountClass()`, code outside package `P-Export` must rely on the implementor of that package providing a factory method (in this case it is method `createAccount()`).

2.4 Threat Model

This section firstly gives an informal presentation of the threat model considered in this paper, followed by a more precise definition of the elements that constitute the threat model.

The threat model represents an attacker with kernel-level code injection privileges introducing malware into a software system. Complex software system often allow separation between user-level and kernel-level code.[2] While the former is subject to certain restrictions (e.g., it cannot access specific hardware functionality without requesting it to the kernel), the latter is not. Kernel-level code injection is a critical vulnerability of complex software system where injected code operates with kernel-level privileges and it can thus bypass all existing software-based security mechanisms.

Even though kernel functionalities as well as bugs that can exploit them are thoroughly checked, exploitable vulnerabilities often appear in complex software systems. Notorious examples include OpenBSD's IPv6 remote kernel buffer overflow[3] and a buffer overrun in JPEG processing of Microsoft applications.[4] An attacker who exploits such a vulnerability injects code that can violate the

[2] We deliberately make this simplification, avoiding to discuss rings of protection for the sake of simplicity.
[3] http://www.securityfocus.com/archive/1/462728/30/150/threaded
[4] https://technet.microsoft.com/library/security/ms04-028

security property of the whole software system and disclose confidential data, disrupt running applications and so forth. The attacker's aim is in fact to violate the security policy of existing software running in the system. The injected code is also not subject to most source-level restrictions such as well-typedness.

For the sake of simplicity, no differentiation between kernel and user code is defined in A+I: all code is already operating at the kernel level. Thus, by modelling the attacker as injecting A+I code, we are modelling exactly kernel-level code injection. Let us now define the system under attack.

The system under attack is assumed to be equipped with a single *PMA* instance that provides one protected memory partition (a protected module). The *PMA* instance is responsible for enforcing the access control policy of Section 2.1 on the whole memory. The instance is assumed to be beyond the reach of A+I code, so it cannot be tampered with by the attacker. This assumption seems reasonable, since most *PMA* implementations have small TCBs that can be verified for the absence of exploitable vulnerabilities such as buffer overflows. A compromised *PMA* would render all security guarantees void, as the attacker would be able to circumvent its access control policies.

Given this system, it is desirable to guarantee that at least the software within the protected module is secure. Definition 1 presents the definition of program security property used in the remainder of this thesis.

Definition 1 (Program security property) *A program security property for* J+E *programs is defined as any property that can be expressed by means of contextual equivalence.*

Security properties that can be expressed with contextual equivalence include confidentiality and integrity (as discussed in Example 4 in Section 2.5.1). By the terms confidentiality and integrity we mean the following.

Definition 2 (Confidentiality) *Confidentiality of a value means that it cannot be discerned by other code besides the code declaring it. Thus, a value v in a program P is confidential if P is contextually-equivalent to P' which is P with a different value for v.*

Definition 3 (Integrity) *Integrity of a value means that it cannot be modified by other code besides the code declaring it. Thus, a value v in a program P has integrity if P is contextually-equivalent to P' which is P where every interaction with other code is followed by a check that the value of v is the same as before the interaction.*

According to Definition 1, J+E security properties include the following:

1. confidentiality and integrity of field contents, of object names and of method bodies;
2. the control flow is only dictated by calls, returns and exception throwing and catching;
3. non-reachability of stuck (error) program states.

Item 1 defines *secrets* in J+E software. Field contents are secrets as they can be never revealed outside a certain program if the programmer chooses to. This also holds for object names which, additionally, cannot be forged by malicious programs. Method bodies are confidential and there is no way to reveal them in J+E, so there should be no way for an A+I-level attacker to discern two different implementations of the same method. In different scenarios, method bodies can be made publicly available to the attacker; in that case, the languages used to model those scenario should allow attacker code to discern method bodies. Secrets can be discerned by an attacker *only* if methods reveal them, no other ways should be exploitable by an attacker to discern a secret. Problem 1 below presents an example of how a naïve compiler implementation would generate code whose confidentiality of field contents can be violated. More of these examples are presented throughout Section 6.1.

Problem 1 (Stack security) *Consider the two code snippets below, presenting two classes that define a* `secret` *field with different values and the same method* `doCallback` *that inputs an object and calls method* `callback` *on it. These two classes are implemented by two objects:* o_L *and* o_R*. In order to differentiate between the left-hand side and the right-hand side implementations, a subscript L or R is added. Assume the presence of an external object* `cb` *of type* `External`*, that presents a method* `callback()`*. When presenting snippets side by side, differences are highlighted in a* `red` *font.*

```
package p_L;
class C_L {
  private secret : Int = 0;

  public doCallback( cb : External
        ) : Int {
    var x : Int = secret;
    cb.callback();
    return 0;
  }
}
object o_L : C_L
```

```
package p_R;
class C_R {
  private secret : Int = 1;

  public doCallback( cb : External
        ) : Int {
    var x : Int = secret;
    cb.callback();
    return 0;
  }
}
object o_R : C_R
```

Objects o_L *and* o_R *are equivalent at the source level, but their compiled counterparts are not. Since local variables are placed on the call stack (in* *unprotected memory) and an* A+I*-level attacker can read unprotected memory, she can read the value of* `x` *during the callback* `cb.callback()`*. Variable* `x` *contains the value of* `secret`*, which is a* **`private`** *field and which is different for both objects.*

A naïve compilation scheme does not enforce the confidentiality or integrity of the call stack, which allows attackers to read and write local variables. An attacker can use this vulnerability to read secrets from the stack, similarly to a buffer-overread attack [114]. Alternatively she can even tamper with the control flow by overwriting a return address on the stack, similarly to a return address clobbering attack [41]. ■

Item 2 ensures that the only way to modify the flow of execution is through method calls, returns and exceptions. So, it is not possible to jump in the middle of a method body nor execute only a part of it (unless an exception was thrown at that stage). Once a method body is called, it will carry out its statements in their entirety up to the next method call, return or exception throw.

Item 3 is similar to Item 2: there is no way of disrupting some functionality by supplying ill-typed parameters to method calls. Well-typed programs "cannot go wrong" [84]. They can diverge, which can be an intended behaviour, but their execution can never be stuck, which is never an intended behaviour.

The presence of a protected module provides enforcement of some of the aforementioned properties, but not all of them are. To ensure all of them are enforced, a secure compiler is used.

Definition 4 (Secure compiler) *Let a compiler be a function that maps source-level programs to target-level ones. A compiler is secure if it outputs programs that enjoy precisely the same security properties of their source-level counterparts, no less, no more.*

The adoption of a secure compiler for compiling J+E software ensures that all aforementioned security properties (Items 1 to 3) are enforced in the compiler output. By defining such a secure compiler as a fully abstract compiler we capture exactly the preservation of those properties in the generated target code: a fully abstract compiler makes the software in the protected memory secure.

For a more precise treatment, the threat model consists of the following definitions: the system under attack (Definition 5), the security property of the system (Definition 6), and the attacker to the system (Definition 7).

Definition 5 (System under attack) *The system is a von Neumann machine with a flat address space and* one *PMA instance that provides a* single *protected partition in memory (a protected module). The protected module contains* A+I *code, called the* protected code*, that complies to* J+E *specification. The unprotected memory contains arbitrary* A+I *code.*

In the system under consideration, for the sake of simplicity, only compiled J+E software is considered to be present and no other software written in other languages. Moreover, only one protected module is assumed to be present. A single module suffices to protect the concerns of a single user or of multiple users who trust each other. Addressing the challenges of adopting multiple protected modules in the system, each belonging to mutually-distrusting stakeholders, is addressed in Chapter 8. An alternative characterisation of the system is to consider it to be composed of J+E code that is compiled inside a module and linked to arbitrary A+I code.

Definition 6 (Security property) *The protected code behaves the same as its* J+E *specification and in no other way.*

This property has the security implications described above since it is applied to the J+E language. In different languages, this property may not have the same security relevance. For example, this property in the context of the C language would not entail confidentiality and integrity, as any structure can be inspected by virtue of simple pointer arithmetic.[5]

Definition 7 (Attacker) *The attacker can arbitrarily change the state of the unprotected partition of the memory, moreover, she also knows how to interact with the secure module. Finally, the attacker cannot violate the PMA access control policy (Table 2.1), e.g., by tampering with the PMA implementation.*

The attacker is assumed to know the interfaces implemented by the protected code: the location of each entry point, the types each method expects and the addresses of possibly static objects. The attacker has knowedge of the functionality of existing software in the system so the injected code can interact (possibly safely) with existing software.

In the following, the code injected by the attacker will be referred to with the terms external code or context.

[5]The C standard states that the behaviour of these scenarios is "undefined", but most C compilers allow arbitrary pointer arithmetic.

Limitations This threat model does not cover all possible security threats that the system is subject to, as exemplified below. Source-level security violations, for example a method returning a field-stored private key that was supposed to be secret, are not considered in this thesis. These violations should not be countered at the compiler level, but with source-level artefacts such as type systems. Availability attacks, for example unprotected code that never calls protected code, are also not considered. The attacker is in fact assumed to interact with the software to be protected in order to violate its security properties. Finally, side channel attacks are also not considered. The definition of the attacker's power, in fact, limits the kind of attacks she can mount. The attacker cannot exploit covert channels to mount side-channels attacks such as timing attacks, since these attacks fall outside the scope of the model.

2.5 Program Equivalences and Security

As mentioned before, program equivalence is used in the definition of secure (fully abstract) compilation. Two program equivalences are mainly used throughout this work, contextual equivalence (Section 2.5.1) and trace equivalence (Section 2.5.2).

2.5.1 Contextual Equivalence

The notion of contextual equivalence (Definition 10 below) relies on the definition of context and of divergence, which are now introduced (Definition 8 and 9).

Definition 8 (Context) *A context $\mathbb{C}$ is a program with a hole (denoted by $[\cdot]$), which can be filled by a program P, generating a new program: $\mathbb{C}[P]$.*

Contexts model the code that can interact with a specific piece of software (in this case, the hole-filling program P). Based on the language of P, contexts can assume a variety of forms. For example, if P is the λ-calculus expression $\lambda x.(xx)$, a context is another λ-calculus expression with a hole, such as $(\lambda y.(yy))\ [\cdot]$ or $[\cdot]\ (\lambda y.(yy))$. In this case, when P is plugged in the hole of either context, the resulting program is the diverging term Ω.[6] Analogously, when P is a Java program, contexts are other Java programs which refer to (and use) the classes P defines.

[6] Ω is in fact the term $\lambda x.(xx)\lambda x.(xx)$ which, under any reduction strategy, always reduces to itself, e.g., it diverges.

Definition 9 (Divergence) *A program P diverges if it performs an unbounded number of reduction steps. Denote that P diverges with $P\Uparrow$.*

Definition 10 (Contextual equivalence [99]) *Two programs P_1 and P_2 written in the same language $\mathcal{L}$ are contextually equivalent if they are interchangeable in any context without affecting the observable behaviour of the program. Formally: $P_1 \simeq^{\mathcal{L}} P_2 \triangleq \forall\mathbb{C}.\ \mathbb{C}[P_1]\Uparrow \iff \mathbb{C}[P_2]\Uparrow$.*

Contextual equivalence (also known as observational equivalence) provides a notion of observation of the behaviour of a program (in this case, termination) and states when two programs exhibit the same observable behaviour. Only what can be observed by the context is of any relevance, and this changes from language to language, since different languages have different functionalities. From the security perspective, contexts can model malicious attackers that interoperate with the secure software (the hole-filling program P) and that can attack that software.

Contextual equivalence can be used to model security properties of source code, as described by Example 4.

Example 4 (Security properties via contextual equivalence) *The code snippets of Figure 2.2 describe a confidentiality property. Both snippets define a class with a private field `secret` whose value is never made public. Calling method `setSecret` assigns different values to `secret`. If the two snippets are*

```
class Secret{
  private secret : Int = 0;

  public setSecret( ) : Int {
    secret = 0;
    return 0;
  }
}
```

```
class Secret{
  private secret : Int = 0;

  public setSecret( ) : Int {
    secret = 1;
    return 0;
  }
}
```

Figure 2.2: These code snippets express confidentiality properties.

contextually equivalent, then the value of `secret` is confidential to the code. Since the two snippets assign different values to `secret`, if they are contextually equivalent, then `secret` must not be discernible by external code.

The code snippets of Figure 2.3 describe an integrity property. Both snippets define a method `proxy` that allocate a variable `secret` on the stack. Then, they perform a call to function `callback` on object `cb` (from Problem 1). The left-hand *side snippet also performs a check whether the variable* `secret` *has been modified during the* `callback` *call. If these code snippets are contextually equivalent, then*

```
public proxy( callback : Unit →
    Unit ) : Int {
  var secret = 0;
  cb.callback();
  if(secret == 0)
    return 0;
  return 1;
}
```

```
public proxy( callback : Unit →
    Unit ) : Int {
  var secret = 0;
  cb.callback();

  return 0;

}
```

Figure 2.3: These code snippets express integrity properties.

`secret` *has not been modified during the* `callback`*, so its integrity is preserved.*

The code snippets of Figure 2.4 describe memory size properties. Both snippets define a method `kernel` *that perform a call to a function* `callback`*, then proceed with security-relevant code (omitted for the sake of simplicity). The left-hand side snippet also allocates* `n` *new objects. The code of function* `callback` *could*

```
public kernel( n : Int, callback :
      Unit → Unit ) : Int {
  for (Int i = 0; i < n; i++){
    new Object();
  }
  cb.callback();
  // security-relevant code
  ...
  return 0;
}
```

```
public kernel( n : Int, callback :
      Unit → Unit ) : Int {



  cb.callback();
  // security-relevant code
  ...
  return 0;
}
```

Figure 2.4: These code snippets express unbounded memory size properties.

disrupt the execution flow by overflowing the memory. If these code snippets were contextually equivalent, then the memory size would not affect the computation.

The code snippets of Figure 2.5 describe memory allocation properties. Both snippets define a method `newObjects` *that allocates two objects* `x` *and* `y` *and then return* `x`*. The only difference is the allocation order between the two snippets. If these snippets are contextually equivalent, then the memory allocation order is invisible to code interacting with them.* ⊡

Although other definitions of contextual equivalence exist (for example, Curien [31] uses reduction to the same value instead of divergence), no alternative formulation drops the universal quantification on contexts.

```
public newObjects( ) : Object {
  var x = new Object();
  var y = new Object();
  return x;
}
```

```
public newObjects( ) : Object {
  var y = new Object();
  var x = new Object();
  return x;
}
```

Figure 2.5: These code snippets express memory allocation properties.

Contextual equivalence shows its limitations both in the attacks it can express and complexity it introduces in proofs. Firstly, timing attacks or, more generally, side-channels attacks cannot be expressed with contextual equivalence. However, these attacks are always disregarded by secure compilation techniques; they can be countered using orthogonal protection mechanisms. Secondly, proving properties about contexts, is notoriously complex [13, 45, 62, 97]. To compensate for this difficulty, different forms of equivalence can be used, e.g., trace equivalence [65, 95], weak bisimulation [105], applicative bisimilarity [8] and logical relations [11], but only if they are first proven to be *as precise* as contextual equivalence. Informally, given an equivalence $\approx^{\mathcal{L}}$, if $\approx^{\mathcal{L}}$ is correct (it captures all equivalent cases) and complete (it does not capture additional cases) w.r.t. $\simeq^{\mathcal{L}}$, then it can be used in place of $\simeq^{\mathcal{L}}$. In this case, $\approx^{\mathcal{L}}$ is said to be fully abstract. Examples of such fully abstract trace semantics will be presented in Chapter 5, while Section 10.4 will discuss related works that achieve similar results.

Well-Behaved Contextual Equivalence

Sometimes it can be desirable to restrict the contexts considered for contextual equivalence to a set of contexts behaving in a particular way. For security purposes, this can be used to reduce (or augment) the power of an attacker to model a particular scenario.

A form of contextual equivalence considering a restricted number of contexts is *well-behaved contextual equivalence*, denoted as $P_1 \overset{\mathrm{w}}{\simeq}{}^{\mathcal{T}}_{\mathcal{S}} P_2$. Well-behaved contextual equivalence is defined for programs of a language $\mathcal{T}$ with respect to a different language $\mathcal{S}$. Well-behaved contextual equivalence is analogous to contextual equivalence except that it considers only $\mathcal{T}$ contexts that behave like $\mathcal{S}$ ones, not just arbitrary $\mathcal{T}$ contexts. Thus, well-behaved contexts replicate the expressiveness of source-level contexts at the target level.

The notion of well-behaved contextual equivalence is used for compiler correctness specification [92]. To prove a compiler correct, one often wants to consider target programs that behave as source-level ones, i.e., ruling out any possible target-level attacker. This is the subject of a vast body of research, but it is not the focus of this thesis, so we do not develop the subject further.

2.5.2 Trace Equivalence

Trace equivalence relates two programs when they produce the same traces, where traces are sequences of actions. Trace equivalence (Definition 12) relies on the concept of labelled transition system (LTS), which is now defined.

Definition 11 (LTS) *A labelled transition system is a triplet $(S, \Lambda, \rightarrow)$ where S is a set of states, Λ is a set of labels and $\rightarrow\ \subseteq S \times \Lambda \times S$ is a ternary relation of labelled transitions.*

A transition between two states S_1 and $S_2 \in S$ on a label $\lambda \in \Lambda$ is indicated as $S_1 \xrightarrow{\lambda} S_2$. Labels represent what an entity external to S can observe from the states of S, as these states perform computations. Labels often concern inputs and outputs, as presented in the following example.

Example 5 (LTS [105]) *Consider the LTS of a vending machine that produces tea or coffee for coins, after the appropriate request is made. It is formalised as $(\{S_I, S_R, S_T, S_C\}$, $\{coin, req\text{-}tea, req\text{-}coffee, tea, coffee\}$, $\{S_I \xrightarrow{coin} S_R$, $S_R \xrightarrow{req\text{-}tea} S_T$, $S_R \xrightarrow{req\text{-}coffee} S_C$, $S_T \xrightarrow{tea} S_I$, $S_C \xrightarrow{coffee} S_I\})$ and it is depicted below.*

$$\begin{tikzcd}
 & & S_T \arrow[dll, bend right, "tea"'] \\
S_I \arrow[r, "coin"] & S_R \arrow[ur, "req\text{-}tea"] \arrow[dr, "req\text{-}coffee"] & \\
 & & S_C \arrow[ull, bend left, "coffee"]
\end{tikzcd}$$

S_I models the state of a vending machine waiting for input, coin expresses the user input and S_R models the state in which the machine wait for the type of product to deliver. Based on the two different inputs from S_R, the machine can reach two states: S_T and S_C, the states where the machine produces tea and coffee, respectively. Then, both S_T and S_C transition back to S_I, labelled with the output it provided to the user: tea or coffee. ⊡

Oftentimes labels are also equipped with decorations that indicate their direction: ! is an observable produced from the program, ? is an observable received from it. The aforementioned transitions can thus be decorated as follows $S_I \xrightarrow{coin?} S_R$, $S_R \xrightarrow{req\text{-}coffee?} S_C$ and $S_T \xrightarrow{tea!} S_I$.

Given the LTS of a program P, the behaviour of P can be described with sequences of labels that can be generated according to the LTS. These sequences of labels, denoted with $\overline{\lambda}$, are called *traces*. The set of of all traces that can be generated by a program is given by its trace semantics. Formally, given that $\xRightarrow{\overline{\lambda}}$ is the reflexive and transitive closure of $\xrightarrow{\lambda}$, the trace semantics of a program P, indicated as $\mathsf{Traces}(P)$, is calculated as follows: $\mathsf{Traces}(P) = \{\overline{\lambda} \mid \exists P'.P \xRightarrow{\overline{\lambda}} P'\}$.

Definition 12 (Trace equivalence) *Two programs are trace equivalent if their trace semantics coincide* $P_1 \overset{T}{=} P_2 \triangleq \mathsf{Traces}(P_1) = \mathsf{Traces}(P_2)$.

When working with trace equivalence in place of contextual equivalence for secure compilation, labels model what the external code (i.e., the context in contextual equivalence) can observe about a program. The external code is modelled as a black box that triggers transitions. So, trace equivalence abstracts from the behaviour of the attacker but maintains the reaction of the program to certain actions of the attacker. This abstraction is the great advantage of trace equivalence as opposed to contextual equivalence. When a trace semantics is fully abstract, it is used in place of contextual equivalence. In this case it is crucial that all possible attacker behaviour is captured by the traces, so as to have a precise characterisation of what the attacker can do.

Trace equivalence also comes with a somewhat simpler proof technique than for contextual equivalence. Traces are in fact often defined inductively, and they give a neat, structural argument for proofs adopting them.

Let us now give two informal examples that present how traces look and what behaviour can they capture. Example 6 presents traces in the J+E setting and Example 7 presents traces in the A+I setting. A more formal presentation of trace semantics is delayed until Chapter 5.

Example 6 (Traces for J+E) *At program level, we are interested in observing function calls and returns. Two kind of labels are thus considered: call (*`call`*) to a method m on object o with parameters ($\overline{v}$) and return (*`ret`*) of a value v.*

Considering Example 3, the following two traces give part of its behaviour.

`call` *`extAccount.getBalance()`*$?$ `ret` $0!$

`call` *`extAccount.addAmount( 5 )`*$?$ `ret` $0!$ `call` *`extAccount.getBalance`*$?$ `ret` $5!$

$\boxdot$

Example 7 (Traces for A+I) *At the* A+I *setting, we are interested in traces capturing the same behaviour captured by traces in* J+E*: function calls and returns. In this setting, however, function do not have names, but they can be identified by the address where a* `call` *jump is made; moreover function calls are not made on objects.*

Considering Example 3, the following two traces give part of its behaviour.

$$\texttt{call}\ 100(12, 10)?\ \texttt{ret}\ 2!$$
$$\texttt{call}\ 100(10, 12)?\ \texttt{ret}\ 0!$$

$\boxdot$

2.6 Fully Abstract Compilation

As previously said, secure compilation is concerned with protecting a part of a whole program from a malicious attacker. Therefore, the compilation scheme considered in this thesis is for partial programs. In A+I, the partial programs of interest are those residing within a protected module while the J+E ones of interest are components C. The formal tools used to reason about compiled programs are tailored to reason about partial programs. In fact, both contextual equivalence and trace equivalence describe the behaviour of partial programs. By working with partial programs we give playground to the attacker: all that is not the partial program is the attacker. We could consider partial programs and then model the attacker as inputs to the program, but the partial program approach is more general and well-established in the literature.

A compiler is fully abstract if it translates indistinguishable source-level programs into indistinguishable target-level programs. Moreover, indistinguishable, compiler target-level programs come from indistinguishable source-level ones. Consider a source language $\mathcal{S}$ and a target language $\mathcal{T}$, the compiled version of an $\mathcal{S}$ program P is denoted with $[\![P]\!]^{\mathcal{S}}_{\mathcal{T}}$.

Definition 13 (Fully abstract compilation [1]) *A compiler is fully abstract if it preserves and reflects contextual equivalence:* $P_1 \simeq^{\mathcal{S}} P_2 \iff [\![P_1]\!]^{\mathcal{S}}_{\mathcal{T}} \simeq^{\mathcal{T}} [\![P_2]\!]^{\mathcal{S}}_{\mathcal{T}}$.

The preservation and reflection of contextual equivalence imply that no security flaws are introduced by the compilation scheme. However an already insecure source language can be compiled to a target language but the output of the compilation will still be insecure. A fully abstract compiler is, in a sense, conservative as it introduces no more vulnerabilities at the target-level than the ones already exploitable at the source-level. Only when the source language has means to specify security properties is a fully abstract compilation scheme a secure compilation scheme.

A Note on Reflection Most modern compilation schemes, such as those for Java and C#, are not secure [1,67]. The main obstacle to secure compilation in this cases is the large number of features provided by these languages. The most notorious language feature to securely compile is *reflection*: i.e., the ability to examine and modify the state of programs at runtime. In fact, reflection rules out any sensible notion of security, since the state of a program can be freely inspected. Conversely, with reflection, contextual equivalence is reduced to alpha-equivalence [86,118], so a fully abstract compiler would be one that translates programs with the same syntactic structure into programs with the same sytactic structure and nothing more. A fully-abstract compiler for a language with reflection could then be developed for untyped assembly language, but it would not be a secure one, since reflection nullifies any sensible abstraction, including security.

An additional benefit of fully abstract compilers comes in the form of *source-level reasoning*. Source-level reasoning means that in order to understand how a program behaves, the programmer needs only think about it at the source level, without considering any other (lower) levels. From a security point of view, this property ensures that security properties of implementations follow from reviewing the source code and its source-level semantics [18]. Source-level reasoning simplifies the task of a programmer, who need not be concerned with the behaviour of target languages and can focus only on the source code.

Definition 13 can also be formulated with the addition of a stochastic assumption, for example that certain values can be guessed with negligible odds. This type of full abstraction result is referred to as *probabilistic full abstraction*.

The proof of full abstraction of a compiler is generally split into two theorems based on the preservation ($\Rightarrow$) and reflection ($\Leftarrow$) directions. In both theorems, the statements to be proven have a universal quantification over all possible contexts, due to the expansion of the definition of contextual equivalence. This makes some proofs particularly complicated, as certain languages (e.g., assembly) have contexts that do not offer a clearly inductive (or co-inductive) structure, and so are of little help for the proof. Some works adopt other equivalences, such as trace equivalence, that are as expressive as contextual equivalence in order to simplify these proofs.

While the connection between fully abstract compilation and security is strong, especially in the scope of this paper, this subject has also been studied without security as a main concern [101, 102]. In these works, fully abstract translations were used to compare the expressiveness of different λ-calculi; if such a translation between two calculi exists, then they are equivalently expressive. A discussion on such related work is delayed until Chapter 10.

2.7 Proof Strategy for Secure Compilation

This section describes how we will prove the compilation scheme from J+E to A+I to be fully abstract. A fully abstract compilation scheme preserves and reflects contextual equivalence of source- and target-level programs. Formally: $C_1 \simeq^{\mathcal{S}} C_2 \iff [\![C_1]\!]^{\mathcal{S}}_{\mathcal{T}} \simeq^{\mathcal{T}} [\![C_2]\!]^{\mathcal{S}}_{\mathcal{T}}$ (Definition 13). In order to prove this statement, the equivalence is split into two cases.

- The direction $[\![C_1]\!]^{\mathcal{S}}_{\mathcal{T}} \simeq^{\mathcal{T}} [\![C_2]\!]^{\mathcal{S}}_{\mathcal{T}} \Rightarrow C_1 \simeq^{\mathcal{S}} C_2$ states that the compiler outputs target-level programs that behave the same as the corresponding source programs. This is what most compilers achieve, at times even certifying the result [28, 75]; we are not interested in this direction. Assuming we start from a correct compiler, this direction is easily proven, since the devised compilation scheme only adds checks to an existing compiler and does not change the way source-level expressions are translated into target-level instructions.

  Most importantly, this direction is unrelated to the security of the compilation scheme: a compiler that is proven to satisfy *only* this direction of the equivalence is not a secure one. The following direction is the one that has security implications.

- The direction $C_1 \simeq^{\mathcal{S}} C_2 \Rightarrow [\![C_1]\!]^{\mathcal{S}}_{\mathcal{T}} \simeq^{\mathcal{T}} [\![C_2]\!]^{\mathcal{S}}_{\mathcal{T}}$ states that source-level abstractions are preserved through compilation to the target level. Proving this direction requires reasoning about contexts, which is difficult when contexts are low-level memories lacking any inductive structure. To avoid working with contexts, we replace the notion of contextual equivalence ($\simeq^{\mathcal{T}}$) at the target level, with that of trace equivalence ($\overset{\mathrm{T}}{=}^{\mathcal{T}}$), which provides an inductive principle to use in the proof. This direction is thus restated as $C_1 \simeq^{\mathcal{S}} C_2 \Rightarrow [\![C_1]\!]^{\mathcal{S}}_{\mathcal{T}} \overset{\mathrm{T}}{=}^{\mathcal{T}} [\![C_2]\!]^{\mathcal{S}}_{\mathcal{T}}$; the contrapositive of this statement is proven to hold: $[\![C_1]\!]^{\mathcal{S}}_{\mathcal{T}} \overset{\mathrm{T}}{\neq}^{\mathcal{T}} [\![C_2]\!]^{\mathcal{S}}_{\mathcal{T}} \Rightarrow C_1 \not\simeq^{\mathcal{S}} C_2$.

Given two different traces of their compiled counterparts $[\![C_1]\!]^{\mathcal{S}}_{\mathcal{T}}$ and $[\![C_2]\!]^{\mathcal{S}}_{\mathcal{T}}$, to prove that C_1 and C_2 are not contextually equivalent, it suffices to show that *there exists* a source-level context that behaves differently depending on whether its hole is filled with C_1 or C_2. Such a context is said to *differentiate* C_1 from C_2. This proof relies on an algorithm that creates a source-level context, a *witness* that differentiates C_1 from C_2, and it is often adopted by related work [10, 32, 64, 94, 95, 96, 97].

Chapter 3

Formalisation of the Source Language J+E

The only true wisdom is knowing you know nothing.

Socrates.

You know nothing, Jon Snow.

Ygritte, feeling particularly socratic. A song of Ice and Fire.

This chapter formalises the J+E language. Firstly, it presents the syntax of J+E (Section 3.1), followed by its static and dynamic semantics (Section 3.2 and Section 3.3, respectively). This formalisation borrows extensively from the Java Jr. work of Jeffrey and Rathke [65]; the only additions are local variables and exceptions.

The main differences between J+E and real-world object-oriented languages (i.e., Java) revolve around the qualifiers associated to classes and fields. A qualifier is an annotation that is applied to a syntactic element (interface, class, method or field) of the language and that dictates how other parts of the program can see that element. Java has the following annotations:

`public` for elements that are always accessible from any point in the program;

`protected` for elements that are only accessible from within the same package;

`private` for elements that are only accessible from within the same class.

In Java, these annotations can be applied freely to all above mentioned syntactic elements, but in J+E this does not hold. Firstly, interfaces are **`public`**; they are always in scope. Classes only have the **`protected`** qualifier, a class name cannot be referred to from outside the package that defines it. Methods only have the **`public`** qualifier. However, when they are defined in an interface, they are effectively accessible from any part of the program. When they are only defined in a class, their qualifier is effectively **`protected`**. Finally, fields are **`private`**, so they cannot be accessed from outside a class.

A second difference between J+E and real-world object-oriented languages is that exceptions can only be caught based on their class type in J+E.

Finally, J+E does not provide an extensive `System` runtime with commonly-used classes readily implemented and the only primitive types it supports are **`Unit`**, **`Bool`** and **`Int`**. No characters encoding is provided in J+E, for the sake of simplicity.

3.1 Syntax

J+E supports many of the basic constructs one expects from a programming language (Figure 3.1). Denote a sequence of elements $E_1, \ldots, E_n$ with $\overline{E}$. Given an element E, e.g., a component C, denote the sets of elements that obey to the related grammar with $\widehat{E}$, e.g. the set of all possible components is denoted with $\widehat{C}$. A program in J+E is a collection of components that communicate via interfaces and public objects. A *component*, is a collection of import packages with a single export package.

Methods in J+E may throw exceptions. The top of the class hierarchy is Obj. The only primitive types in J+E are `Unit`, inhabited by `unit`; `Bool`, inhabited by `true` and `false`; and `Int`, inhabited by word-sized integers. The expression E in p is borrowed from Java Jr. It is a type coercion that allows the following: if the expression E is well-typed to run in package p with return type t, then the expression E in p is well-typed to run in any package q with return type t, as long as t is a visible type in q.

programs	$\mathcal{P} ::= \overline{C}$
components	$C ::= \overline{P_i}; P_e$
import packages	$P_i ::= \{\textbf{package } p; \overline{D_i}\}$
export packages	$P_e ::= \{\textbf{package } p; \overline{D_e}\}$
import declarations	$D_i ::= \textbf{interface } i \textbf{ extends } \bar{t}\ \{\overline{M_t}\}$
	$\mid \textbf{extern } o : t;$
export declarations	$D_e ::= \textbf{class } c \textbf{ extends } t \textbf{ implements } \bar{t}\ \{K\ \overline{F_t}\ \overline{M}\}$
	$\mid \textbf{object } o : t \textbf{ implements } \bar{t}\ \{\overline{F}\}$
constructors	$K ::= c(\overline{f} : \bar{t}, \overline{f'} : \overline{t'})\ \{\textbf{super}(\overline{f}); \textbf{this}.\overline{f''} = \overline{f'}\}$
fields	$F ::= \textbf{private } f = v$
field types	$F_t ::= \textbf{private } f : t$
methods	$M ::= \textbf{public } m(\overline{x} : \bar{t}) : t\ \ [\textbf{throws } t]\ \ \{\textbf{return } E;\}$
method types	$M_t ::= \textbf{public } m(\overline{x} : \bar{t}) : t\ \ [\textbf{throws } t]$
expressions	$E ::= v \mid x \mid E.f \mid E.f = E \mid E.m(\overline{E}) \mid E \ \mathsf{op}\ E \mid \textbf{exit } E$
	$\mid E; E \mid E \ \mathsf{in}\ p \mid \textbf{var } x : t = E \mid \textbf{if } (E)\ \{E\} \textbf{ else } \{E\}$
	$\mid \textbf{new } t(\overline{E}) \mid \textbf{try } \{E\} \textbf{ catch } (x : t)\ \{E\} \mid \textbf{throw } E$
types	$t ::= p.c \mid p.i \mid p.c \ \mathsf{in}\ p \mid p.c \ \mathsf{in}\ * \mid \textbf{Obj} \mid \textbf{Unit} \mid \textbf{Bool} \mid \textbf{Int}$
operations	$\mathsf{op} ::= + \mid - \mid \cdot \mid / \mid \wedge, \mid \vee \mid \cdots$
values	$v ::= p.o \mid \mathsf{unit} \mid \mathsf{true} \mid \mathsf{false} \mid n \mid \textbf{throw } v$

Figure 3.1: Syntax of J+E.

3.2 Static Semantics

This section presents the static semantics of J+E (Section 3.2.2) after defining the notion used therein (Section 3.2.1).

3.2.1 Notation

This section introduces some auxiliary functions and notation that the static semantics of J+E relies upon.

Definition 14 (Declaration equivalence) *Define $\overline{D_1} \equiv \overline{D_2}$ whenever two sequences of declarations are equal up to reordering: $\overline{D_1}\,\overline{D_2}\,\overline{D_3} \equiv \overline{D_2}\,\overline{D_1}\,\overline{D_3}$.*

Definition 15 (Package equivalence) *Define $\overline{P}_1 \equiv \overline{P}_2$ when two sequences of package definitions are equal up to reordering of packages and of declarations:*

$$\begin{array}{lll} \overline{P}_1\overline{P}_2\overline{P}_3 & \equiv \overline{P}_2\overline{P}_1\overline{P}_3 & \\ \{\mathit{package}\ p; \overline{D}_1\} & \equiv \{\mathit{package}\ p; \overline{D}_2\} & \mathit{if}\ \overline{D}_1 \equiv \overline{D}_2 \end{array}$$

Definition 16 (Domain) *Define* $\mathtt{dom}(\cdot)$ *as:*

$$\begin{array}{ll} \mathtt{dom}(P_1 \dots P_n) & = \mathtt{dom}(P_1) \cup \dots \cup \mathtt{dom}(P_n) \\ \mathtt{dom}(\{\mathit{package}\ p; \overline{D}\}) & = \{p.n \mid n \in \mathtt{dom}(\overline{D})\} \cup \{p\} \\ \mathtt{dom}(D) & = \{\mathtt{name}(D)\} \end{array}$$

The domain of a list of elements is the set of elements obtained by applying the domain function to all elements of the list.

Definition 17 (Names) *Define* $\mathtt{name}(\cdot)$ *as:*

$$\begin{array}{llll} \mathtt{name}(\mathit{class}\ c \dots) & = c & \mathtt{name}(\mathit{object}\ o \dots) & = o \\ \mathtt{name}(\mathit{interface}\ i \dots) & = i & \mathtt{name}(\mathit{extern}\ o : \overline{t};) & = o \\ \mathtt{name}(\mathit{public}\ m \dots) & = m & \mathtt{name}(m(\overline{x} : \overline{t}) : t;) & = m \\ \mathtt{name}(f : t) & = f & \mathtt{name}(f = v) & = f \end{array}$$

Use n to range over names. Define $\mathtt{fn}(\cdot)$ *as the free names of an entity, namely its name and the names of all other syntactic categories it contains.*

Definition 18 (Paths) *Define $C.p$ as:*

$$C.p = \{\mathit{package}\ p; \overline{D}\} \qquad \mathit{if}\ \{\mathit{package}\ p; \overline{D}\} \in C$$

Define $C.p.n$ (or $C.t$ where $t \equiv p.n$) as:

$$C.p.n = \{\mathit{package}\ p; D\} \qquad \mathit{if}\ \{\mathit{package}\ p; \overline{D}\} \in C, D \in \overline{D}, \mathtt{name}(D) = n$$

Definition 19 (Super types) *Define* $C.t.\mathtt{superTypes}()$ *as:*

$$\begin{array}{rl} C.\mathit{Obj}.\mathtt{superTypes}() & = \epsilon \\ C.t.\mathtt{superTypes}() & = t', \overline{t} \\ & \quad \mathit{if}\ C.t = \{\mathit{package}\ p; \mathit{class}\ c\ \mathit{extends}\ t' \\ & \qquad\qquad \mathit{implements}\ \overline{t}\{K\ \overline{F_t}\ \overline{M}\}\} \\ C.t.\mathtt{superTypes}() & = \overline{t} \\ & \quad \mathit{if}\ C.t = \{\mathit{package}\ p; \mathit{interface}\ i\ \mathit{extends}\ \overline{t}\{\overline{M_t}\}\} \end{array}$$

Definition 20 (Acyclic component) *A component C is acyclic when for all types t defined in C, given the supertypes of t as $C.t.\mathtt{superTypes}() = \overline{t}$, there is no $u \in \overline{t}$ that is a supertype of another supertype of t.*

For the remainder of the paper assume components to be acyclic.

Definition 21 (Additions) *Define the addition of components $C + C'$ as:*

$$
\begin{aligned}
\overline{C} + \epsilon &= \overline{C} \\
\overline{C} + (\{\textit{package } p; \overline{D}\}, \overline{C}') &= (\overline{C}, \{\textit{package } p; \overline{D}\}) + \overline{C}' \\
&\quad \textit{if } p \notin \mathtt{dom}(C) \\
(\overline{C_1}, \{\textit{package } p; \overline{D}\}, \overline{C_2}) +&\\
(\{\textit{package } p; \overline{D}'\}, \overline{C'}) &= (\overline{C_1}, \{\textit{package } p; \overline{D} + \overline{D}'\}, \overline{C_2}) + \overline{C'}
\end{aligned}
$$

Define the addition of sequences of declarations $\overline{D} + \overline{D}'$ as:

$$
\begin{aligned}
\overline{D} + \epsilon &= \overline{D} \\
\overline{D} + (D, \overline{D'}) &= (\overline{D}, D) + \overline{D}' && \textit{if } \mathtt{name}(D) \notin \mathtt{dom}(\overline{D}) \\
(\overline{D}_1, D, \overline{D}_2) + (D', \overline{D}') &= (\overline{D}_1, D', \overline{D}_2) + \overline{D}' && \textit{if } \mathtt{name}(D) = \mathtt{name}(D')
\end{aligned}
$$

$M + M'$, $M_t + M_t'$, $F + F'$ and $F_t + F_t'$ are defined analogously.

Definition 22 (Interfaces) *Define $C_1, \ldots, C_n.\mathtt{interfaces}()$ (and analogous functions) as:*

$$
\begin{aligned}
C_1, \ldots, C_n.\mathtt{interfaces}() &= C_1, \mathtt{interfaces}() \cup \ldots \\
&\quad \cup C_n.\mathtt{interfaces}() \\
P_{i1}, \ldots, P_{in}; P_e.\mathtt{interfaces}() &= P_{i1}.\mathtt{interfaces}() \cup \ldots \\
&\quad \cup P_{in}.\mathtt{interfaces}() \\
\textbf{\textit{interface}}\ i\ \textbf{\textit{extends}}\ \overline{t}\ \{\overline{M_t}\}.\mathtt{interfaces}() &= \textbf{\textit{interface}}\ i\ \textbf{\textit{extends}}\ \overline{t}\ \{\overline{M_t}\} \\
\textbf{\textit{extern}}\ o : t; .\mathtt{interfaces}() &= \emptyset
\end{aligned}
$$

Definition 23 (Headers) *Define $C.t.\mathtt{headers}()$ as:*

$$
\begin{aligned}
C.\textit{Obj}.\mathtt{headers}() &= \epsilon \\
C.t.\mathtt{headers}() &= C.t'.\mathtt{headers}() + \overline{M}.\mathtt{headers}() \\
&\quad \textit{if } C.t = \{\textit{package } p; \textit{class } c \textit{ extends } t' \\
&\qquad \textit{implements } \overline{t}\{K\ \overline{F_t}\ \overline{M}\}\} \\
C.t.\mathtt{headers}() &= C.\overline{t}.\mathtt{headers}() + \overline{M_t} \\
&\quad \textit{if } C.t = \{\textit{package } p; \textit{interface } i \textit{ extends } \overline{t}\{\overline{M_t}\}\}
\end{aligned}
$$

Define $M.\mathtt{headers}()$ *as:*

$$\textit{public}\ m(\overline{x}:\overline{t}):t\{\textit{return}\ E;\}.\mathtt{headers}() = m(\overline{x}:\overline{t}):t;$$

Define $\overline{t}.\mathtt{headers}()$ *as:*

$$t_1.\mathtt{headers}() + \ldots + t_n.\mathtt{headers}() \qquad \textit{if}\ \overline{t} = t_1 \ldots t_n$$

$\overline{M}.\mathtt{headers}()$ *is defined analogously.*

Definition 24 (Header compatibility) *A sequence of headers* $\overline{M_t}$ *is compatible, denoted with* $\vdash \overline{M_t} : \mathsf{compat}$*, if, when a method name occurs more than once, it has the same signature.*

$$\begin{aligned}\vdash M_{t1} \ldots M_{tn} : \mathsf{compat} \triangleq \forall i, j \in 1..ni \neq j \Rightarrow& \\ &M_{ti} = m(\overline{x}:\overline{t}):t; \in \overline{N}\ \textit{and}\ M_{tj} = m(\overline{y}:\overline{u}):u; \in \overline{N} \\ &\textit{and}\ t \equiv u\ \textit{and}\ \overline{t} \equiv \overline{u}\end{aligned}$$

Definition 25 (Interface compatibility) *Two interfaces* I_1 *and* I_2 *are compatible, denoted with* $I_1 \frown I_2$*, if they both have distinct names and the method they define are compatible.*

$$\begin{aligned}I_1 \frown I_2 \triangleq\ & \mathtt{name}(I_1) \neq \mathtt{name}(I_2)\ \textit{and} \\ &\forall M_{t1} \in \mathtt{name}(I_1).\mathtt{headers}(), \forall M_{t2} \in \mathtt{name}(I_2).\mathtt{headers}(), \\ &\textit{if}\ \mathtt{name}(M_{t1}) = \mathtt{name}(M_{t2}),\ \textit{then}\ \vdash M_{t1}, M_{t2} : \mathsf{compat}\end{aligned}$$

Definition 26 (Fields) *Define* $C.t.\mathtt{fields}()$ *as:*

$$\begin{aligned}&C.\textit{Obj}.\mathtt{fields}() = \epsilon \\ &C.t.\mathtt{fields}() = C.t'.\mathtt{fields}() + \overline{F_t} \\ &\qquad \textit{if}\ C.t = \{\textit{package}\ p; \textit{class}\ c\ \textit{extends}\ t'\ \textit{implements}\ \overline{t}\{K\ \overline{F_t}\ \overline{M}\}\}\end{aligned}$$

Definition 27 (Methods) *Define* $C.t.\mathtt{methods}()$ *as:*

$$\begin{aligned}&C.\textit{Obj}.\mathtt{methods}() = \epsilon \\ &C.t.\mathtt{methods}() = C.t'.\mathtt{methods}() + \overline{M} \\ &\qquad \textit{if}\ C.t = \{\textit{package}\ p; \textit{class}\ c\ \textit{extends}\ t'\ \textit{implements}\ \overline{t}\{K\ \overline{F_t}\ \overline{M}\}\}\end{aligned}$$

Definition 28 (Super) *Define* $C.t.\mathtt{super}()$ *as the supertype of type* t *in component* C*. Formally:*

$$\begin{aligned}&C.\textit{Obj}.\mathtt{super}() = \epsilon \\ &C.t.\mathtt{super}() = t' \\ &\qquad \textit{if}\ C.t = \{\textit{package}\ p; \textit{class}\ c\ \textit{extends}\ t'\ \textit{implements}\ \overline{t}\{K\ \overline{F_t}\ \overline{M}\}\}\end{aligned}$$

Definition 29 (Exports) *Define $C.p.$`exports` to be the component containing all the export packages for C, and analogously for $C.p.$`imports`.*

Definition 30 (Method can throw an exception) *Define $M.$`canThrow`(t) to be true if the method signature of M includes **`throws`** t' where $t <: t'$.*

3.2.2 Static Semantics

The static semantics defines typing relations based on the judgements of Figure 3.2. Adopt Γ as the standard type environment that binds variables to

$\vdash \mathcal{P} : \mathsf{prg}$	well-typed program $\mathcal{P}$
$\vdash C : \mathsf{cmp}$	well-typed component C
$C \vdash P : \mathsf{pkg}$	well-typed package P
$C \vdash D : \mathsf{dec\ in}\ p$	well-typed declaration D in package p
$C \vdash t : \mathsf{type\ in}\ p$	t is a valid type in package p
$C \vdash c : \mathsf{cls\ in}\ p$	c is a valid class in package p
$C \vdash i : \mathsf{itf\ in}\ p$	i is a valid interface in package p
$C \vdash t <: t'\ \mathsf{in}\ p$	t is subtype of t' in package p
$C \vdash v : t\ \mathsf{in}\ *$	value v has type t in the whole component
$C \vdash K : \mathsf{cnstr\ in}\ p$	well-typed constructor K in package p
$C \vdash M_t : \mathsf{hdr\ in}\ p.i$	well-typed method header M_t of interface i in package p
$C \vdash M : \mathsf{mth\ in}\ p.c$	well-typed method M of class c in package p
$C; \Gamma; M; \overline{t} \vdash E : t\ \mathsf{in}\ p$	well-typed expression E in package p in current method M according to caught-types $\overline{t}$

Figure 3.2: Typing judgments of J+E.

types, where variables cannot be repeated [98]. Since secure compilation relies on components, typing can also be done on a per-component basis. Thus, when typechecking a component C, the component is added to the typing environment to act as a reference to the standard class table. Moreover, current method M and caught types $\overline{t}$ are added to the environment for checking expressions to handle exceptions. To preserve the encapsulation principle given by packages, types are annotated with package names; thus an expression does not simply have type t but type t in p.

Figures 3.3 to 3.6 present the typing rules [65]. They are mostly standard, except for the following modifications. Rule Declaration-class, Declaration-object, Declaration-interface and Method-Types are modified to ensure the "programming to an interface" paradigm is enforced. Rule Expr-concat and Scope-all have been devised by the authors as they were not presented in the original paper. Rule Expr-try and Expr-throw have been added to support exceptions; Rule Expr-var supports local variables and Rule Expr-exit supports termination via the `exit` command.

$$\text{(Program)}\quad \frac{\mathcal{P} \equiv \overline{C} \qquad \vdash \overline{C} : \mathsf{cmp} \qquad \forall I_1, I_2 \in \mathcal{P}.\mathtt{interfaces}().I_1 \frown I_2}{\vdash \mathcal{P} : \mathsf{prg}}$$

$$\text{(Component)}\quad \frac{C \vdash \overline{P_i} : \mathsf{pkg} \qquad C \vdash P_e : \mathsf{pkg}}{\vdash \overline{P_i}; P_e : \mathsf{cmp}}$$

$$\text{(Packages)}\quad \frac{C \vdash \overline{D} : \mathsf{dec\ in}\ p}{C \vdash \{\mathsf{package}\ p; \overline{D}\} : \mathsf{pkg}}$$

$$\text{(Declaration-class)}\quad \frac{C \vdash t : \mathsf{cls\ in}\ p \qquad \mathsf{name}(K) = c \qquad C \vdash \overline{t} : \mathsf{itf\ in}\ * \qquad C \vdash K : \mathsf{cnstr\ in}\ p \qquad C \vdash \overline{M} : \mathsf{mth\ in}\ p.c \qquad \forall u \in \{t, \overline{t}\}\ C.u.\mathtt{headers}() \subseteq C.p.c.\mathtt{headers}()}{C \vdash \mathtt{class}\ c\ \mathtt{extends}\ t\ \mathtt{implements}\ \overline{t}\ \{K\ \overline{F_t}\ \overline{M}\} : \mathsf{dec\ in}\ p}$$

$$\text{(Declaration-object)}\quad \frac{C \vdash t : \mathsf{cls\ in}\ p \qquad C \vdash t <: \overline{t}\ \mathsf{in}\ p \qquad C \vdash \overline{t} : \mathsf{itf\ in}\ * \qquad C.t.\mathtt{flds} = \{\overline{f} : \overline{t}'\} \qquad C \vdash \overline{v} : \overline{t}'\ \mathsf{in}\ p}{C \vdash \mathtt{object}\ o : t\ \mathtt{implements}\ \overline{t}\ \{\overline{f} = \overline{v}\} : \mathsf{dec\ in}\ p}$$

$$\text{(Declaration-interface)}\quad \frac{C \vdash \overline{t} : \mathsf{itf\ in}\ * \qquad C \vdash \overline{M_t} : \mathsf{hdr\ in}\ p.i \qquad \forall t \in \overline{t}\ C.t.\mathtt{headers}() \subseteq C.p.i.\mathtt{headers}()}{C \vdash \mathtt{interface}\ i\ \mathtt{extends}\ \overline{t}\ \{\overline{M_t}\} : \mathsf{dec\ in}\ p}$$

$$\text{(Declaration-extern)}\quad \frac{C \vdash \overline{t} : \mathsf{type\ in}\ * \qquad \forall I_1, I_2 \in C.\overline{t}.\mathtt{headers}().I_1 \frown I2}{C \vdash \mathtt{extern}\ o : \overline{t}; : \mathsf{dec\ in}\ p}$$

Figure 3.3: J+E typing rules (part I).

(Classes)
$$\frac{C.t = \{\texttt{package}\ p; \texttt{class}\ c\ \texttt{extends}\ t'\ \texttt{implements}\ \bar{t}\ \{K\ \overline{F_t}\ \overline{M}\}\}}{C \vdash t : \mathsf{cls\ in}\ p}$$

(Classes-Obj)
$$\frac{}{C \vdash \texttt{Obj} : \mathsf{cls\ in}\ p}$$

(Interfaces)
$$\frac{C.t = \{\texttt{package}\ p; \texttt{interface}\ i\ \texttt{extends}\ \bar{t}\ \{\overline{M_t}\}\}}{C \vdash t : \mathsf{itf\ in}\ p}$$

(Types-class)
$$\frac{C \vdash t : \mathsf{cls\ in}\ p}{C \vdash t : \mathsf{type\ in}\ p}$$

(Types-interface)
$$\frac{C \vdash t : \mathsf{itf\ in}\ p}{C \vdash t : \mathsf{type\ in}\ p}$$

(Subtype-refl)
$$\frac{C \vdash t : \mathsf{type\ in}\ p}{C \vdash t <: t\ \mathsf{in}\ p}$$

(Subtype-trans)
$$\frac{C \vdash t <: t''\ \mathsf{in}\ p \qquad C \vdash t'' <: t'\ \mathsf{in}\ p}{C \vdash t <: t'\ \mathsf{in}\ p}$$

(Subtype-obj)
$$\frac{C \vdash t : \mathsf{type\ in}\ p}{C \vdash t <: \texttt{Obj}\ \mathsf{in}\ p}$$

(Subtype-def)
$$\frac{C \vdash t : \mathsf{type\ in}\ p \qquad t' \in C.t.\texttt{superTypes}()}{C \vdash t <: t'\ \mathsf{in}\ p}$$

(Scope-all)
$$\frac{C.p = \{\texttt{package}\ p; \overline{D}\} \qquad v \notin \texttt{fn}(C \setminus p) \qquad C \vdash v : t\ \mathsf{in}\ p}{C \vdash v : t\ \mathsf{in}\ *}$$

(Constructors)
$$\frac{C \vdash \overline{u} : \mathsf{type\ in}\ p \qquad C.c.\texttt{flds} = \{\overline{g} : \overline{u}\} \qquad C.p.c.\texttt{super.flds} = \{\overline{f}' : \overline{t}\}}{C \vdash c(\overline{f} : \overline{t}, \overline{h} : \overline{u})\{\texttt{super}(\overline{f}); \texttt{this}.\overline{g} = \overline{h}\} : \mathsf{cnstr\ in}\ p}$$

(Method-Types)
$$\frac{C \vdash t : \mathsf{itf\ in}\ * \qquad C \vdash \overline{t} : \mathsf{itf\ in}\ *}{C \vdash m(\overline{x} : \overline{t}) : t;: \mathsf{hdr\ in}\ p}$$

(Methods)
$$\frac{C; \overline{x} : \overline{t}, \texttt{this} : p.c; M; \emptyset \vdash E : t\ \mathsf{in}\ p \qquad C \vdash t : \mathsf{type\ in}\ p \qquad C \vdash \overline{t} : \mathsf{type\ in}\ p \qquad M = \texttt{public}\ m(\overline{x} : \overline{t}) : t\ \{\texttt{return}\ E;\}}{C \vdash \texttt{public}\ m(\overline{x} : \overline{t}) : t\ \{\texttt{return}\ E;\} : \mathsf{mth\ in}\ p.c}$$

Figure 3.4: J+E typing rules (part II).

3.3 Dynamic Semantics

The dynamic semantics is given in terms of a relation $(\mathcal{P}; S \vdash E) \to (\mathcal{P}'; S' \vdash E')$ that models the evolution of program $\mathcal{P}$ executing expression E with stack S to $\mathcal{P}'$ executing E' with stack S'. A binding B is a list of associations from variables to values, $B ::= \emptyset \mid B; (x \mapsto v)$. The lookup of the value associated to a variable, denoted as $B(x)$, returns v if $(x \mapsto v) \in B$ and is undefined otherwise. A stack S is a list of bindings $S ::= \overline{B}$, lookup and addition are always made to the top of the stack, so if $S = B_1, \cdots, B_n$, then $S(x)$ stands for $B_1(x)$ and $S, (x \mapsto v)$ stands for $B_1, (x \mapsto v)$. The expression being executed is immersed in an evaluation context $\mathbb{E}$, which models the environment in which

(Expr-val-obj)
$$\frac{C.v = \{\texttt{package}\ p; \texttt{object}\ o : t\ \texttt{implements}\ \overline{t}\{\overline{F}\}\}}{C;\Gamma;M;\overline{t} \vdash v : t\ \textsf{in}\ p}$$

(Expr-val-obj-itf)
$$\frac{C.v = \{\texttt{package}\ p; \texttt{object}\ o : t\ \texttt{implements}\ \overline{t}\{\overline{F}\}\} \qquad t' \in \overline{t}}{C;\Gamma;M;\overline{t} \vdash v : t'\ \textsf{in}\ p}$$

(Expr-val-extern)
$$\frac{C.v = \{\texttt{package}\ p; \texttt{extern}\ o : \overline{t};\} \qquad t \in \overline{t}}{C;\Gamma;M;\overline{t} \vdash v : t\ \textsf{in}\ p}$$

(Expr-var)
$$\frac{x : t \in \Gamma \qquad C \vdash t : \textsf{type in}\ p}{C;\Gamma;M;\overline{t} \vdash x : t\ \textsf{in}\ p}$$

(Expr-fld)
$$\frac{C;\Gamma;M;\overline{t} \vdash E : t'\ \textsf{in}\ p \qquad f : t \in C.t'.\texttt{flds}}{C;\Gamma;M;\overline{t} \vdash E.f : t\ \textsf{in}\ p}$$

(Expr-fldup)
$$\frac{C;\Gamma;M;\overline{t} \vdash E : u\ \textsf{in}\ p \qquad C;\Gamma;M;\overline{t} \vdash E' : t\ \textsf{in}\ p \qquad f : t \in C.u.\texttt{fields}()}{C;\Gamma;M;\overline{t} \vdash E.f = E' : t\ \textsf{in}\ p}$$

(Expr-meth)
$$\frac{C;\Gamma;M;\overline{t} \vdash E : u\ \textsf{in}\ p \qquad C;\Gamma;M;\overline{t} \vdash \overline{E} : \overline{t}\ \textsf{in}\ p \qquad m(\overline{x} : \overline{t}) : t \in C.u.\texttt{headers}()}{C;\Gamma;M;\overline{t} \vdash E.m(\overline{E}) : t\ \textsf{in}\ p}$$

(Expr-new)
$$\frac{C \vdash c : \textsf{cls in}\ p \qquad C \vdash \overline{E} : \overline{t} : \textsf{in}\ p \qquad C \vdash C.p.c.\texttt{fields}() : \overline{t}}{C;\Gamma;M;\overline{t} \vdash \textsf{new}\ p.c(\overline{E}) : p.c\ \textsf{in}\ p}$$

(Expr-if)
$$\frac{C;\Gamma;M;\overline{t} \vdash E : u\ \textsf{in}\ p \qquad C;\Gamma;M;\overline{t} \vdash E' : u\ \textsf{in}\ p \qquad C;\Gamma;M;\overline{t} \vdash E_T : t\ \textsf{in}\ p \qquad C;\Gamma;M;\overline{t} \vdash E_F : t\ \textsf{in}\ p}{C;\Gamma;M;\overline{t} \vdash (E == E' ? E_T : E_F) : t\ \textsf{in}\ p}$$

(Expr-concat)
$$\frac{C;\Gamma \vdash E : u\ \textsf{in}\ p \qquad C;\Gamma, E : u\ \textsf{in}\ p \vdash E' : t\ \textsf{in}\ p}{C;\Gamma;M;\overline{t} \vdash E;E' : t\ \textsf{in}\ p}$$

(Expr-coercion)
$$\frac{C;\Gamma;M;\overline{t} \vdash E : t\ \textsf{in}\ p \qquad C \vdash t : \textsf{type in}\ q}{C;\Gamma;M;\overline{t} \vdash E\ \textsf{in}\ p : t\ \textsf{in}\ q}$$

(Expr-subsumption)
$$\frac{C;\Gamma;M;\overline{t} \vdash E : t\ \textsf{in}\ p \qquad C \vdash t <: u\ \textsf{in}\ p}{C;\Gamma;M;\overline{t} \vdash E : u\ \textsf{in}\ p}$$

Figure 3.5: J+E typing rules (part III).

$$\frac{C;\Gamma;M;\overline{t},t\vdash E : t' \text{ in } p \qquad C;\Gamma,x:t;M;\overline{t}\vdash E' : t' \text{ in } p \qquad C\vdash t : \mathsf{cls} \text{ in } p}{C;\Gamma;M;\overline{t}\vdash \textbf{try}\ \{E\}\ \textbf{catch}\ (x:t)\ \{E'\} : t' \text{ in } p}\ (\text{Expr-try})$$

$$\frac{C;\Gamma;M;\overline{t}\vdash E : t \text{ in } p \qquad M.\mathtt{canThrow}(t) \text{ or } t\in\overline{t}}{C;\Gamma;M;\overline{t}\vdash \textbf{throw}\ E : t \text{ in } p}\ (\text{Expr-throw})$$

$$\frac{C;\Gamma;M;\overline{t}\vdash E : t' \text{ in } p}{C;\Gamma;M;\overline{t}\vdash \textbf{var}\ x:t=E : t' \text{ in } p}\ (\text{Expr-var}) \qquad \frac{C;\Gamma;M;\overline{t}\vdash E : t \text{ in } p}{C;\Gamma;M;\overline{t}\vdash \textbf{exit}\ E : t \text{ in } p}\ (\text{Expr-exit})$$

Figure 3.6: J+E typing rules (part IV).

the evaluation takes place. The syntax of evaluation contexts is:

$$\begin{aligned}\mathbb{E} ::= {}& [\cdot] \mid \mathbb{E}.m(\overline{E}) \mid p.o.m(\overline{v},\mathbb{E},\overline{E}) \mid \mathbb{E}.f \mid \mathbb{E}.f=E \mid v.f=\mathbb{E} \mid \textbf{new}\ t(\overline{v},\mathbb{E},\overline{E})\\ &\mid \textbf{if}(\mathbb{E})\{E_T\}\textbf{else}\{E_F\} \mid \mathbb{E};E \mid \mathbb{E} \text{ in } p \mid \textbf{var}\ x:t=\mathbb{E} \mid \textbf{return}\ \mathbb{E}\\ &\mid \mathbb{E} \text{ op } E \mid v \text{ op } \mathbb{E} \mid \textbf{try}\ \mathbb{E}\ \textbf{catch}(x:t)\ E \mid \textbf{throw}\ \mathbb{E} \mid \textbf{exit}\ \mathbb{E}\end{aligned}$$

The following are the redexes of the language:

$$\begin{aligned}\mathbb{R} ::= {}& v.m(\overline{v}) \mid \textbf{return}\ v \mid v.f \mid v.f=u \mid \textbf{new}\ p.c(\overline{v}) \mid \textbf{if}(v)\{E_T\}\textbf{else}\{E_F\}\\ &\mid v \text{ in } p \mid \textbf{try}\{v\}\textbf{catch}(x:t)\{E\} \mid \textbf{exit}\ v \mid \textbf{throw}\ v\end{aligned}$$

Rules for reductions of the form $(\mathcal{P};S\vdash E)\to(C';S'\vdash E')$ are presented in Figures 3.7 to 3.9.

Contextual equivalence for J+E programs is defined based on J+E contexts $\mathbb{C}$, which are components with a hole, denoted with $\mathbb{C}[\cdot]$ [65]. The hole can be filled with another component C to denote the interaction between C and the context. Assume the context defined a `Main` package with a `Main` class and a `main` method that identify where the execution starts.

Definition 31 (Contextual equivalence for J+E)

$$C_1 \simeq^{\text{J+E}} C_2 \triangleq \forall\mathbb{C}.\ \mathbb{C}[C_1]\Uparrow \iff \mathbb{C}[C_2]\Uparrow .$$

(Eval-method)

$$\frac{C \in \mathcal{P} \qquad C.v = \{\textbf{package}\ p; \textbf{object}\ v : t\ \textbf{implements}\ \bar{t}\ \{\overline{F}\}\} \\ \textbf{public}\ m(\overline{x} : \bar{t}) : t'\{\textbf{return}\ E;\} \in C.t.\texttt{mths}}{(\mathcal{P}; S \vdash \mathbb{E}[v.m(\overline{v})]) \to (\mathcal{P}; \emptyset, S \vdash \mathbb{E}[\textbf{return}\ E[v/\texttt{this}, \overline{v}/\overline{x}]\ \mathsf{in}\ p])}$$

(Eval-return)

$$\frac{}{(\mathcal{P}; B, S \vdash \mathbb{E}[\textbf{return}\ v]) \to (\mathcal{P}; S \vdash \mathbb{E}[v])}$$

(Eval-field)

$$\frac{C \in \mathcal{P} \qquad f = u \in \overline{F} \\ C.v = \{\textbf{package}\ p; \textbf{object}\ v : t\ \textbf{implements}\ \bar{t}\ \{\overline{F}\}\}}{(\mathcal{P}; S \vdash \mathbb{E}[v.f]) \to (\mathcal{P}; S \vdash \mathbb{E}[u])}$$

(Eval-field-update)

$$\frac{C \in \mathcal{P} \qquad \mathcal{P}' = \mathcal{P} + C' \qquad (f = u;) \in \overline{F} \qquad \overline{F}' = \overline{F} + (f = u) \\ C.v = \{\textbf{package}\ p; \textbf{object}\ v : t\ \textbf{implements}\ \bar{t}\ \{\overline{F}\}\} \\ C' = C + \{\textbf{package}\ p; \textbf{object}\ v : t\ \textbf{implements}\ \bar{t}\ \{\overline{F}'\}\}}{(\mathcal{P}; S \vdash \mathbb{E}[v.f = u]) \to (\mathcal{P}'; S \vdash \mathbb{E}[u])}$$

(Eval-new)

$$\frac{C \in \mathcal{P} \qquad \mathcal{P}' = \mathcal{P} + C' \qquad C.p.c.\texttt{flds} = \overline{f} : \bar{t} \qquad p.o \notin \texttt{dom}(C) \\ C' = C + \{\textbf{package}\ p; \textbf{object}\ o : p.c\ \textbf{implements}\ \epsilon\{\overline{f} = \overline{v}\}\}}{(\mathcal{P}; S \vdash \mathbb{E}[\textbf{new}\ p.c(\overline{v})]) \to (\mathcal{P}'; S \vdash \mathbb{E}[p.o])}$$

(Eval-if-true)

$$\frac{v = \texttt{true}}{(\mathcal{P}; S \vdash \mathbb{E}[\textbf{if}(v)\{E_T\}\textbf{else}\{E_F\}]) \to (\mathcal{P}; S \vdash \mathbb{E}[E_T])}$$

(Eval-if-false)

$$\frac{v = \texttt{false}}{(\mathcal{P}; S \vdash \mathbb{E}[\textbf{if}(v)\{E_T\}\textbf{else}\{E_F\}]) \to (\mathcal{P}; S \vdash \mathbb{E}[E_F])}$$

Figure 3.7: Dynamic semantics of J+E (part I).

(Eval-coercion)

$$\frac{}{(\mathcal{P}; S \vdash \mathbb{E}[v \text{ in } p]) \to (\mathcal{P}; S \vdash \mathbb{E}[v])}$$

(Eval-local-var)

$$\frac{}{(\mathcal{P}; S \vdash \mathbb{E}[\textbf{var } x : t = v]) \to (\mathcal{P}; S, (x \mapsto v) \vdash \mathbb{E}[\texttt{unit}])}$$

(Eval-lookup)

$$\frac{S(x) = v}{(\mathcal{P}; S \vdash \mathbb{E}[x]) \to (\mathcal{P}; S \vdash \mathbb{E}[v])}$$

(Eval-concatenation)

$$\frac{}{(\mathcal{P}; S \vdash \mathbb{E}[v; E]) \to (\mathcal{P}; S \vdash \mathbb{E}[E])}$$

(Eval-op)

$$\frac{v \texttt{ op } v' = v''}{(\mathcal{P}; S \vdash \mathbb{E}[v \texttt{ op } v']) \to (\mathcal{P}; S \vdash \mathbb{E}[v''])}$$

(Eval-try)

$$\frac{v \neq \texttt{throw } v'}{(\mathcal{P}; S \vdash \mathbb{E}[\textbf{try}\{v\}\textbf{catch}(x : t)\{E\}]) \to (\mathcal{P}; S \vdash \mathbb{E}[v])}$$

(Eval-catch)

$$\frac{C \in \mathcal{P} \qquad C.v = \{\textbf{package } p; \textbf{object } v : t \textbf{ implements } \bar{t}\{\overline{F}\}\} \qquad t <: t'}{(\mathcal{P}; S \vdash \mathbb{E}[\textbf{try}\{\textbf{throw } v\}\textbf{catch}(x : t')\{E\}]) \to (\mathcal{P}; S \vdash \mathbb{E}[E[v/x]])}$$

(Eval-catch-fail)

$$\frac{C \in \mathcal{P} \qquad C.v = \{\textbf{package } p; \textbf{object } v : t \textbf{ implements } \bar{t}\{\overline{F}\}\} \qquad t \not<: t'}{(\mathcal{P}; S \vdash \mathbb{E}[\textbf{try}\{\textbf{throw } v\}\textbf{catch}(x : t')\{E\}]) \to (\mathcal{P}; S \vdash \mathbb{E}[\textbf{throw } v])}$$

(Eval-exit)

$$\frac{}{(\mathcal{P}; S \vdash \mathbb{E}[\textbf{exit } v]) \to (\mathcal{P}; S \vdash v)}$$

Figure 3.8: Dynamic semantics of J+E (part II). The subtyping relation is denoted with $<:$.

(Eval-throw-sequence)

$$\frac{}{(\mathcal{P}; S \vdash \mathbb{E}[\texttt{throw}\ v; E]) \rightarrow (\mathcal{P}; S \vdash \mathbb{E}[\texttt{throw}\ v])}$$

(Eval-throw-throw)

$$\frac{}{(\mathcal{P}; S \vdash \mathbb{E}[\texttt{throw throw}\ v]) \rightarrow (\mathcal{P}; S \vdash \mathbb{E}[\texttt{throw}\ v])}$$

(Eval-throw-var)

$$\frac{}{(\mathcal{P}; S \vdash \mathbb{E}[\texttt{var}\ x : t = \texttt{throw}\ v]) \rightarrow (\mathcal{P}; S, (x \mapsto \texttt{throw}\ v) \vdash \mathbb{E}[\texttt{throw}\ v])}$$

(Eval-throw-new)

$$\frac{}{(\mathcal{P}; S \vdash \mathbb{E}[\texttt{new throw}\ v]) \rightarrow (\mathcal{P}; S \vdash \mathbb{E}[\texttt{throw}\ v])}$$

(Eval-throw-if)

$$\frac{}{(\mathcal{P}; S \vdash \mathbb{E}[\texttt{if}(\texttt{throw}\ v)\{E_T\}\texttt{else}\{E_F\}]) \rightarrow (\mathcal{P}; S \vdash \mathbb{E}[\texttt{throw}\ v])}$$

Figure 3.9: Dynamic semantics of J+E (part III).

Chapter 4

Formalisation of the Target Language A+I

Stop fidgeting with the ruler, minion!

prof. Dave Clarke.

This chapter presents the formalisation of the A+I language, the target language of the secure compilation scheme of Chapter 6. Language A+I is formalised as an untyped assembly language, and this chapter defines its syntax (Section 4.1) and its dynamic semantics (Section 4.2). As the semantics relies on *PMA*, the access control policy presented in Section 2.1 is also formalised. Finally, this chapter defines contextual equivalence for A+I programs and presents examples of contextual equivalence at work to relate equivalent programs (Section 4.3).

4.1 Syntax

A+I programs run on an architecture that models a von Neumann machine consisting of a program counter p, a register file r, a flags register f and memory space m. The program counter indicates the address of the instruction that is executed next. The register file contains 12 general purpose registers $\mathtt{r}_0$ to $\mathtt{r}_{11}$ and a stack pointer register SP, which contains the address of the top of the current call stack. The flags register contains a zero flag ZF and a sign flag SF, which are set or cleared by arithmetic instructions and are used by branching instructions, respectively. Instructions executed by the language are listed in Figure 4.1.

movl $\mathtt{r_d}$ $\mathtt{r_s}$	Load the word from the memory address in register $\mathtt{r_s}$ into register $\mathtt{r_d}$.
movs $\mathtt{r_d}$ $\mathtt{r_s}$	Store the contents of register $\mathtt{r_s}$ at the address found in register $\mathtt{r_d}$.
movi $\mathtt{r_d}$ k	Load the constant value k into register $\mathtt{r_d}$. Note that $\mathtt{k} < 2^\ell$.
add $\mathtt{r_d}$ $\mathtt{r_s}$	Write $\mathtt{r_d} + \mathtt{r_s} \bmod 2^\ell$ into register $\mathtt{r_d}$ and set the ZF flag accordingly.
sub $\mathtt{r_d}$ $\mathtt{r_s}$	Write $\mathtt{r_d} - \mathtt{r_s} \bmod 2^\ell$ into register $\mathtt{r_d}$ and set both the ZF and the SF flags accordingly.
cmp $\mathtt{r_1}$ $\mathtt{r_2}$	Calculate $\mathtt{r_1} - \mathtt{r_2}$ and set both the ZF and the SF flags accordingly.
jmp $\mathtt{r_i}$	Jump to the address located in register $\mathtt{r_i}$.
je $\mathtt{r_i}$	If the ZF flag is set, jump to the address in register $\mathtt{r_i}$.
jl $\mathtt{r_i}$	If the SF flag is set, jump to the address in register $\mathtt{r_i}$.
call $\mathtt{r_i}$	Push the value of the program counter +1 onto the call stack and jump to the address in register $\mathtt{r_i}$.
ret	Pop a value from the call stack and jump to the popped location.
halt	Stop the execution. We consider the value of register $\mathtt{r_0}$ the result of the execution.

Figure 4.1: Instruction set $\mathcal{I}$ of language A+I.

For the sake of simplicity, assume the architecture targeted by the language works with ℓ bit-long words, where ℓ is a power of 2. This allows the formalisation presented to scale to architectures with words of different sizes.

Figure 4.2 presents elements of the formalisation. Words w are sequences of bits 0 or 1 of length ℓ. Instructions i are elements of the set $\mathcal{I}$ and define the programming language executed on the architecture (Figure 4.1). Addresses a are natural numbers, ranging from 0 to $2^\ell - 1$. Memories m are maps from addresses to words. Memory access, denoted as $m(a)$, is defined as follows: $m(a) = w$ if $a \mapsto w \in m$; it is undefined otherwise. Define the domain of a memory as $\mathtt{dom}(m) = \{a \mid a \mapsto w \in m\}$. If two memories m and m' have disjoint domains, they can be merged into another memory. Formally, if $\mathtt{dom}(m) \cap \mathtt{dom}(m') = \emptyset$, then $m + m' = \{a \mapsto w \mid a \mapsto w \in m \text{ or } a \mapsto w \in m'\}$. Registers files r map each register ($\mathtt{r_0}$ to $\mathtt{r_{11}}$ and the stack pointer SP to a word. Flags registers f map the sign and zero flags SF and ZF to a word.

$$
\begin{array}{rl}
\textit{Words} & w ::= [0 \text{ or } 1]^{\ell} \\
\textit{Instructions} & i \in \mathcal{I} \subset \textit{Words} \\
\textit{Numbers} & n ::= n \in \mathbb{N} \\
\textit{Addresses} & a \in 0..2^{\ell} - 1 \\
\textit{Memories} & m ::= \emptyset \\
 & \quad \mid m; a \mapsto w \\
\textit{Register files} & r ::= \mathtt{r}_0 \mapsto w, \ldots, \mathtt{r}_{11} \mapsto w, \mathsf{SP} \mapsto w \\
\textit{Flags registers} & f ::= \mathsf{ZF} \mapsto w, \mathsf{SF} \mapsto w \\
\textit{Memory descriptors} & s ::= (a_b, n_c, n_d, n, a_{uc}, a_{ud}) \\
\textit{Programs} & P ::= (m, s)
\end{array}
$$

Figure 4.2: Elements of the A+I formalisation.

Memory descriptors s are sextuples: $(a_b, n_c, n_d, n, a_{uc}, a_{ud})$ that formalise the concepts of Section 2.2. a_b is the address where the protected memory partition starts, n_c and n_d are the sizes (in number of addresses) of the code and data section respectively and n is the number of entry points. Additionally, a_{uc} states where the code section of the unprotected code starts and a_{ud} states where the data section of the unprotected code starts (and where the unprotected code section ends). This partitioning of unprotected code is not enforced by *PMA* architectures but it helps devising a fully abstract trace semantics, as explained at the end of Section 5.1. Entry points are allocated starting from the base address a_b. Each entry point is $\mathcal{N}_e$ words long. Assume that the entry points do not overflow the protected code section, thus the constraint $n \cdot \mathcal{N}_e < n_c$ holds for the all memory descriptors. Programs P are pairs of a memory m and a memory descriptor s.

4.2 Semantics

Before introducing the semantics, a number of auxiliary notions are defined.

Figure 4.3 defines the access control enforcement rules informally presented in Section 2.2. Read judgments $s \vdash \mathsf{predicate}(a, b, \cdots)$ as: "according to memory descriptor s, predicate holds for addresses a, b, $\cdots$".

$$\frac{a_b \leq p < (a_b + n_c + n_d)}{s \vdash \mathsf{protected}(p)}\ \text{(Aux-protected)} \qquad \frac{p < a_b}{s \vdash \mathsf{unprotected}(p)}\ \text{(Aux-unprotected1)} \qquad \frac{(a_b + n_c + n_d) \leq p}{s \vdash \mathsf{unprotected}(p)}\ \text{(Aux-unprotected2)}$$

$$\frac{a_{uc} \leq a < a_{ud}}{s \vdash \mathsf{unprotectedCode}(a)}\ \text{(Aux-unprotected-code)} \qquad \frac{a_{ud} \leq a \qquad s \nvdash \mathsf{unprotected}(a)}{s \vdash \mathsf{unprotectedData}(a)}\ \text{(Aux-unprotected-data)}$$

$$\frac{p = a_b + (n-1) \cdot \mathcal{N}_e}{s \vdash \mathsf{returnEntryPoint}(p)}\ \text{(Aux-returnEntry)} \qquad \frac{\begin{array}{c} p = a_b + m \cdot \mathcal{N}_e \\ m \in \mathbb{N} \qquad m < n \end{array}}{s \vdash \mathsf{entryPoint}(p)}\ \text{(Aux-entryPoint)} \qquad \frac{\begin{array}{c} (a_b + n_c) \leq p \\ p < (a_b + n_c + n_d) \end{array}}{s \vdash \mathsf{data}(p)}\ \text{(Aux-data)}$$

$$\frac{\begin{array}{c} s \vdash \mathsf{protected}(p) \\ s \vdash \mathsf{protected}(a) \end{array}}{s \vdash \mathsf{readAllowed}(p, a)}\ \text{(Aux-read-1)} \qquad \frac{s \vdash \mathsf{unprotectedData}(a)}{s \vdash \mathsf{readAllowed}(p, a)}\ \text{(Aux-read-2)}$$

$$\frac{s \vdash \mathsf{unprotectedData}(a)}{s \vdash \mathsf{writeAllowed}(p, a)}\ \text{(Aux-write-1)} \qquad \frac{\begin{array}{c} s \vdash \mathsf{protected}(p) \\ s \vdash \mathsf{data}(a) \end{array}}{s \vdash \mathsf{writeAllowed}(p, a)}\ \text{(Aux-write-2)}$$

$$\frac{\begin{array}{c} s \vdash \mathsf{unprotectedCode}(p) \\ s \vdash \mathsf{entryPoint}(p') \end{array}}{s \vdash \mathsf{entryJump}(p, p')}\ \text{(Aux-entry)} \qquad \frac{\begin{array}{c} s \vdash \mathsf{protected}(p) \\ s \vdash \mathsf{unprotectedCode}(p') \end{array}}{s \vdash \mathsf{exitJump}(p, p')}\ \text{(Aux-return)} \qquad \frac{\begin{array}{c} s \vdash \mathsf{protected}(p) \\ s \vdash \mathsf{protected}(p') \\ s \nvdash \mathsf{data}(p') \end{array}}{s \vdash \mathsf{intJump}(p, p')}\ \text{(Aux-internal)}$$

$$\frac{\begin{array}{c} s \vdash \mathsf{unprotectedCode}(p) \\ s \vdash \mathsf{unprotectedCode}(p') \end{array}}{s \vdash \mathsf{extJump}(p, p')}\ \text{(Aux-external)}$$

Figure 4.3: Access control enforcement rules. Assume $s \equiv (a_b, n_c, n_d, n, a_{uc}, a_{ud})$.

Define functions $\mathtt{m}_{\mathsf{sec}}(m, s)$ and $\mathtt{m}_{\mathsf{ext}}(m, s)$, which return the protected and unprotected parts of a memory m according to descriptor s, respectively as:

$$\mathtt{m}_{\mathsf{sec}}(m, s) = \{a \mapsto w \mid a \mapsto w \in m, s \vdash \mathsf{protected}(a)\}$$
$$\mathtt{m}_{\mathsf{ext}}(m, s) = \{a \mapsto w \mid a \mapsto w \in m, s \vdash \mathsf{unprotected}(a)\}$$

In the semantics there are two call stacks, one for the protected code, called the *secure stack*, and one for the unprotected code, called the *insecure stack*. Each stack is preceded by a word containing the location of the current top of the stack: $\mathsf{SP}_{\mathsf{sec}}$ and $\mathsf{SP}_{\mathsf{ext}}$ are memory locations that indicate the top of the secure and insecure stack respectively. Given a memory descriptor $s = (a_b, n_c, n_d, n, a_{uc}, a_{ud})$, the secure stack starts at the beginning of the protected data section and the insecure stack starts at beginning of the unprotected data section, both stack grow up. Thus $\mathsf{SP}_{\mathsf{sec}} = (a_b + n_c)$ and, initially, $\mathsf{SP}_{\mathsf{sec}} \mapsto (a_b+n_c+1)$; analogously, $\mathsf{SP}_{\mathsf{ext}} = (a_{ud})$ and, initially, $\mathsf{SP}_{\mathsf{ext}} \mapsto (a_{ud}+1)$. Call and return instructions see the SP register being set to the correct address when crossing boundaries between protected and unprotected memory by using $\mathsf{SP}_{\mathsf{sec}}$ and $\mathsf{SP}_{\mathsf{ext}}$. The value of the program counter is pushed onto the stack by a `call` instruction, while a `ret` instruction pops one address from the top of the stack and jumps to that location. Updating the stack pointer SP is performed using the auxiliary function $\searrow^{\mathsf{SS}}$: $\widehat{p} \times \widehat{r} \times \widehat{m} \times \widehat{s} \to \widehat{p} \times \widehat{r} \times \widehat{m}$ (Figure 4.4).[1] The stack switching enforcement rules do not modify anything if a `call` or a

$$\text{(Stack-out-to-in)}\quad \frac{\begin{array}{c} s \vdash \mathsf{entryJump}(p, p') \\ m' = m[\mathsf{SP}_{\mathsf{ext}} \mapsto r(\mathsf{SP})] \\ r' = r[\mathsf{SP} \mapsto m(\mathsf{SP}_{\mathsf{sec}})] \\ s \vdash \mathsf{unprotected}\ r(\mathsf{SP}) \\ s \vdash \mathsf{protected}\ r'(\mathsf{SP}) \end{array}}{p, r, m, s \searrow^{\mathsf{SS}} p', r', m'}$$

$$\text{(Stack-in-to-out)}\quad \frac{\begin{array}{c} s \vdash \mathsf{exitJump}(p, p') \\ m' = m[\mathsf{SP}_{\mathsf{sec}} \mapsto r(\mathsf{SP})] \\ r' = r[\mathsf{SP} \mapsto m(\mathsf{SP}_{\mathsf{ext}})] \\ s \vdash \mathsf{protected}\ r(\mathsf{SP}) \\ s \vdash \mathsf{unprotected}\ r'(\mathsf{SP}) \end{array}}{p, r, m, s \searrow^{\mathsf{SS}} p', r', m'}$$

$$\text{(Stack-no-change-i)}\quad \frac{\begin{array}{c} s \vdash \mathsf{intJump}(p, p') \\ s \vdash \mathsf{protected}\ r(\mathsf{SP}) \end{array}}{p, r, m, s \searrow^{\mathsf{SS}} p', r, m}$$

$$\text{(Stack-no-change-e)}\quad \frac{\begin{array}{c} s \vdash \mathsf{extJump}(p, p') \\ s \vdash \mathsf{unprotected}\ r(\mathsf{SP}) \end{array}}{p, r, m, s \searrow^{\mathsf{SS}} p', r, m}$$

Figure 4.4: Stack switch enforcement rules.

`ret` are performed within the same memory partition (Rule Stack-no-change-i and Rule Stack-no-change-e). For a `call` or a `ret` across different memory partitions, Rule Stack-out-to-in updates the top of the unprotected stack ($\mathsf{SP}_{\mathsf{ext}}$) with the current value in the stack pointer. Then, it sets the stack pointer to the location contained in the top of the unprotected stack (contained in $\mathsf{SP}_{\mathsf{sec}}$). Rule Stack-in-to-out performs the dual of Rule Stack-out-to-in.

In the rules, notation $m[a \mapsto w]$ indicates that memory m is updated to a new one that is equal to m except that the value stored at address a is w. Notation $r[\mathtt{r} \mapsto w]$ indicates that the register file r is updated to a new one that is equal to r except that the value stored in register $\mathtt{r}$ is w. Notation $r(\mathtt{r})$ indicates the value contained in register $\mathtt{r}$ in register file r. Given a jump between addresses p and p', the stack switch rules produce a new register file r' and a new memory m' based on old ones r and m. The memory is updated to store the top of the

[1] Recall that $\widehat{E}$ denotes the set of elements E that obey to the related grammar.

current stack, located in SP, in the address storing the top of the current stack. When the stack is changed, the register file is updated to initialise SP to the top of the right stack: the address stored at $\mathsf{SP}_{\mathsf{sec}}$ or $\mathsf{SP}_{\mathsf{ext}}$.

The operational semantics is a small step semantics that describes how each instruction of the language transforms an execution state into a new one. The operational semantics describes programs in terms of the whole memory: both the protected and unprotected partitions.[2]

Definition 32 (Execution state) *An execution state, denoted as Ω, is a quintuple $\Omega = (p, r, f, m, s)$, where p is a program counter, r is a registers file, f is a flags register, m is a memory and s is a memory descriptor.*

Given execution state $\Omega = (p, r, f, m, s)$, let $\lfloor\Omega\rfloor$ be the state for protected programs $(p, r, f, \mathtt{m_{sec}}(m, s), s)$. Dually, let $\lceil\Omega\rceil$ be the state for unprotected programs $(p, r, f, \mathtt{m_{ext}}(m, s), s)$. We overload the $\mathtt{m_{sec}}(\cdot)$ and the $\mathtt{m_{ext}}(\cdot)$ functions to deal with states, so that if $\Omega = (p, r, f, m, s)$, $\mathtt{m_{sec}}(\Omega)= \mathtt{m_{sec}}(m, s)$ and $\mathtt{m_{ext}}(\Omega)= \mathtt{m_{ext}}(m, s)$. We can thus state that a state Ω can be split into a protected state $\lfloor\Omega\rfloor$ and some unprotected memory $\mathtt{m_{ext}}(\Omega)$, formally: $\Omega = \lfloor\Omega\rfloor + \mathtt{m_{ext}}(\Omega)$. Dually, a state is an unprotected state and some protected memory, so $\Omega = \lceil\Omega\rceil + \mathtt{m_{sec}}(\Omega)$.

Relation $\xrightarrow{i} \subseteq \lfloor\Omega\rfloor \times \lfloor\Omega\rfloor$ (Figures 4.5 to 4.6) describes the evaluation of instructions that only affect the protected memory. Dually, relation $\xrightarrow{e} \subseteq \lceil\Omega\rceil \times \lceil\Omega\rceil$ describes the evaluation of instructions that only affect the unprotected memory. Rules for $\xrightarrow{e}$ can be obtained from the rules for $\xrightarrow{i}$ by replacing all `intJump` assumptions with an `extJump` one and are thus omitted. Let $m(p) = \mathtt{inst}$ denote that `inst` is the word allocated in $m(p)$, where $\mathtt{inst} \in \mathcal{I}$.

When an access control violation is detected, or when the secure stack is overflowed, all registers and flags are reset and the execution is halted. Note that the program counter is set to -1 whenever the `halt` instruction is encountered, in order to capture termination. This way, no progress can be made, as $m(-1)$ does not return a valid instruction: the program is in a stuck state.

Definition 33 (Stuck state) *A state $\Omega = (p, r, f, m, s)$ is stuck, denoted as $\Omega^{\perp}$, when the program counter does not point to a valid instruction: $m(p) \notin \mathcal{I}$.*

The operational semantics of A+I is captured by the binary relation over states $\rightarrow \subseteq \Omega \times \Omega$ (Figure 4.7). Relation $\rightarrow$ defines how to evaluate instructions that affect the whole memory. Thus, it relies on $\xrightarrow{i}$ and $\xrightarrow{e}$ and it defines rules for

[2]The trace semantics for A+I in Chapter 5 will describe protected programs only, in terms of the protected memory partition.

$$\text{(Eval-movl)}\quad \frac{\begin{array}{c} m(p) = (\texttt{movl}\ \texttt{r}_\texttt{d}\ \texttt{r}_\texttt{s}) \\ s \vdash \mathsf{intJump}(p, p+1) \\ s \vdash \mathsf{readAllowed}(p, r(\texttt{r}_\texttt{s})) \\ r' = r[\texttt{r}_\texttt{d} \mapsto m(r(\texttt{r}_\texttt{s}))] \end{array}}{(p, r, f, m, s) \xrightarrow{i} (p+1, r', f, m, s)}$$

$$\text{(Eval-movs)}\quad \frac{\begin{array}{c} m(p) = (\texttt{movs}\ \texttt{r}_\texttt{d}\ \texttt{r}_\texttt{s}) \\ s \vdash \mathsf{intJump}(p, p+1) \\ s \vdash \mathsf{writeAllowed}(p, r(\texttt{r}_\texttt{d})) \\ m' = m[r(\texttt{r}_\texttt{d}) \mapsto r(\texttt{r}_\texttt{s})] \end{array}}{(p, r, f, m, s) \xrightarrow{i} (p+1, r, f, m', s)}$$

$$\text{(Eval-movi)}\quad \frac{\begin{array}{c} m(p) = (\texttt{movi}\ \texttt{r}_\texttt{d}\ \texttt{i}) \\ s \vdash \mathsf{intJump}(p, p+1) \\ r' = r[\texttt{r}_\texttt{d} \mapsto i] \end{array}}{(p, r, f, m, s) \xrightarrow{i} (p+1, r', f, m, s)}$$

$$\text{(Eval-compare)}\quad \frac{\begin{array}{c} m(p) = (\texttt{cmp}\ \texttt{r}_\texttt{s}\ \texttt{r}_\texttt{d}) \\ s \vdash \mathsf{intJump}(p, p+1) \\ f' = f[\mathsf{ZF} \mapsto (\texttt{r}_\texttt{s} == \texttt{r}_\texttt{d}); \\ \mathsf{SF} \mapsto (\texttt{r}_\texttt{s} < \texttt{r}_\texttt{d})] \end{array}}{(p, r, f, m, s) \xrightarrow{i} (p+1, r, f', m, s)}$$

$$\text{(Eval-add)}\quad \frac{\begin{array}{c} m(p) = (\mathsf{add}\ \texttt{r}_\texttt{d}\ \texttt{r}_\texttt{s}) \qquad s \vdash \mathsf{intJump}(p, p+1) \qquad v = (r(\texttt{r}_\texttt{d}) + r(\texttt{r}_\texttt{s}))\%2^\ell \\ r' = r[\texttt{r}_\texttt{d} \mapsto v] \qquad f' = f[\mathsf{ZF} \mapsto (v == 0)] \end{array}}{(p, r, f, m, s) \xrightarrow{i} (p+1, r', f', m, s)}$$

$$\text{(Eval-sub)}\quad \frac{\begin{array}{c} m(p) = (\mathsf{sub}\ \texttt{r}_\texttt{d}\ \texttt{r}_\texttt{s}) \qquad s \vdash \mathsf{intJump}(p, p+1) \qquad v = (r(\texttt{r}_\texttt{d}) - r(\texttt{r}_\texttt{s}))\%2^\ell \\ r' = r[\texttt{r}_\texttt{d} \mapsto v] \qquad f' = f[\mathsf{ZF} \mapsto (v == 0); \mathsf{SF} \mapsto (r(\texttt{r}_\texttt{d}) - r(\texttt{r}_\texttt{s}) < 0)] \end{array}}{(p, r, f, m, s) \xrightarrow{i} (p+1, r', f', m, s)}$$

Figure 4.5: Operational semantics of instructions in the protected memory partition (part I).

instructions that affect both protected and unprotected memory at the same time (e.g., a call between protected and unprotected memory). Let us now describe the rules that define the $\rightarrow$ relation. Rule Eval-protected and Eval-unprotected capture reductions that happen only in protected and only in unprotected memory. Rule Eval-movl-out captures a read to unprotected memory while Rule Eval-movs-out captures a write to unprotected memory. Rule Eval-outcall and Eval-returnback ensure that the address to be followed after an outcall is stored in the secure stack and that the address of the returnback entry point is pushed onto the insecure stack. Thus the unprotected code always jumps to the returnback entry point when returning from an outcall. Rule Eval-call and Eval-return capture function calls and returns.

The reflexive-transitive closure of relation $\rightarrow$ is indicated with $\rightarrow^*$. The evaluation of program P is a sequence of steps that takes the initial state of P to another state.

Definition 34 (Initial state) *The initial state of a program* (m, s), *denoted as* $\Omega_0(m, s)$, *is a state* (p_0, r_0, f_0, m, s), *where* $s = (a_b, n_c, n_d, n)$, $p_0 = (a_b + n_c +$

$$\frac{m(p) = (\texttt{call}\ \texttt{r}_\texttt{d}) \qquad p' = r(\texttt{r}_\texttt{d}) \\ s \vdash \mathsf{intJump}(p, p') \\ p, r, m, s \searrow^{\mathsf{SS}} p', r', m' \\ r'' = r'[\mathsf{SP} \mapsto r(\mathsf{SP}) + 1] \\ m'' = m'[r''(\mathsf{SP}) \mapsto p + 1]}{(p, r, f, m, s) \xrightarrow{i} (p', r'', f, m'', s)}\ \text{(Eval-function-call)}$$

$$\frac{m(p) = (\texttt{ret}) \qquad p' = m(r(\mathsf{SP})) \\ s \vdash \mathsf{intJump}(p, p') \\ r' = r[\mathsf{SP} \mapsto r(\mathsf{SP}) - 1] \\ p, r', m, s \searrow^{\mathsf{SS}} p', r'', m'}{(p, r, f, m, s) \xrightarrow{i} (p', r'', f, m', s)}\ \text{(Eval-function-ret)}$$

$$\frac{m(p) = (\texttt{je}\ \texttt{r}_\texttt{i}) \qquad f(\mathsf{ZF}) == 1 \\ p' = r(\texttt{r}_\texttt{i}) \qquad s \vdash \mathsf{intJump}(p, p')}{(p, r, f, m, s) \xrightarrow{i} (p', r, f, m, s)}\ \text{(Eval-je-true)}$$

$$\frac{m(p) = (\texttt{jl}\ \texttt{r}_\texttt{i}) \qquad f(\mathsf{SF}) == 1 \\ p' = r(\texttt{r}_\texttt{i}) \qquad s \vdash \mathsf{intJump}(p, p')}{(p, r, f, m, s) \xrightarrow{i} (p', r, f, m, s)}\ \text{(Eval-jl-true)}$$

$$\frac{m(p) = (\texttt{je}\ \texttt{r}_\texttt{i}) \qquad f(\mathsf{ZF}) == 0 \\ s \vdash \mathsf{intJump}(p, p + 1)}{(p, r, f, m, s) \xrightarrow{i} (p + 1, r, f, m, s)}\ \text{(Eval-je-false)}$$

$$\frac{m(p) = (\texttt{jl}\ \texttt{r}_\texttt{i}) \qquad f(\mathsf{SF}) == 0 \\ s \vdash \mathsf{intJump}(p, p + 1)}{(p, r, f, m, s) \xrightarrow{i} (p + 1, r, f, m, s)}\ \text{(Eval-jl-false)}$$

$$\frac{m(p) = (\texttt{jmp}\ \texttt{r}_\texttt{d}) \qquad p' = r(\texttt{r}_\texttt{d}) \\ s \vdash \mathsf{intJump}(p, p')}{(p, r, f, m, s) \xrightarrow{i} (p', r, f, m, s)}\ \text{(Eval-jump)}$$

$$\frac{m(p) = (\texttt{halt})}{(p, r, f, m, s) \xrightarrow{i} (-1, r, f, m, s)}\ \text{(Eval-halt)}$$

Figure 4.6: Operational semantics of instructions in the protected memory partition (part II).

$n_d + 2)$, $r_0 = [\mathsf{SP} \mapsto m(\mathsf{SP}_{\mathsf{ext}}); \texttt{r}_\texttt{i} \mapsto 0\ {}_{\texttt{i}=0..11}]$, and $f_0 = [\mathsf{ZF} \mapsto 0; \mathsf{SF} \mapsto 0]$.

The evaluation of P terminates if $\exists\Omega'.\Omega_0(P) \to^* \Omega'$ and $\Omega'^{\perp}$; the result of the computation is stored in $\texttt{r}_0$. If the evaluation of program P does not terminate, P diverges. A state Ω performing n reduction steps is indicated as $\Omega \to^n \Omega'$.

Definition 35 (Divergence for A+I programs) *A program P diverges, denoted with $P\Uparrow$ if it executes an unbounded number of reduction steps: $P\Uparrow \triangleq \forall n \in \mathbb{N}, \exists\Omega'.\ \Omega_0(P) \to^n \Omega'$.*

4.3 Contextual Equivalence for A+I

Contextual equivalence relates two programs that cannot be distinguished by any third program interacting with them [99]. This section specialises the general definition of contextual equivalence (Definition 10) to A+I programs.

$$\dfrac{\lfloor\Omega\rfloor \xrightarrow{i} \lfloor\Omega'\rfloor \qquad \Omega = \lfloor\Omega\rfloor + \mathtt{m}_{\mathtt{ext}}(\Omega) \qquad \Omega' = \lfloor\Omega'\rfloor + \mathtt{m}_{\mathtt{ext}}(\Omega)}{\Omega \to \Omega'}\ \text{(Eval-protected)}$$

$$\dfrac{\lceil\Omega\rceil \xrightarrow{e} \lceil\Omega'\rceil \qquad \Omega = \lceil\Omega\rceil + \mathtt{m}_{\mathtt{sec}}(\Omega) \qquad \Omega' = \lceil\Omega'\rceil + \mathtt{m}_{\mathtt{sec}}(\Omega)}{\Omega \to \Omega'}\ \text{(Eval-unprotected)}$$

(Eval-movs-out)
$$\dfrac{\begin{array}{c} m(p) = (\mathtt{movs}\ \mathtt{r_d}\ \mathtt{r_s}) \qquad s \vdash \mathsf{intJump}(p, p+1) \qquad s \vdash \mathsf{writeAllowed}(p, r(\mathtt{r_d})) \\ s \vdash \mathtt{unprotected}(r(\mathtt{r_d})) \qquad m' = m[r(\mathtt{r_d}) \mapsto r(\mathtt{r_s})] \end{array}}{(p, r, f, m, s) \to (p+1, r, f, m', s)}$$

(Eval-movl-out)
$$\dfrac{\begin{array}{c} m(p) = (\mathtt{movl}\ \mathtt{r_d}\ \mathtt{r_s}) \qquad s \vdash \mathsf{intJump}(p, p+1) \qquad s \vdash \mathsf{readAllowed}(p, r(\mathtt{r_s})) \\ s \vdash \mathsf{unprotected}(r(\mathtt{r_s})) \qquad r' = r[\mathtt{r_d} \mapsto m(r(\mathtt{r_s}))] \end{array}}{(p, r, f, m, s) \to (p+1, r', f, m, s)}$$

(Eval-outcall)
$$\dfrac{\begin{array}{c} m(p) = (\mathtt{call}\ \mathtt{r_d}) \qquad p' = r(\mathtt{r_d}) \qquad s \vdash \mathsf{exitJump}(p, p') \\ r' = r[\mathsf{SP} \mapsto r(\mathsf{SP}) + 1] \qquad m' = m[r(\mathsf{SP}) \mapsto p+1] \\ p, r', m', s \searrow^{\mathsf{SS}} p', r'', m'' \qquad r''' = r''[\mathsf{SP} \mapsto r''(\mathsf{SP}) + 1] \\ m''' = m''[r'''(\mathsf{SP}) \mapsto a] \qquad s \vdash \mathsf{returnEntryPoint}(a) \end{array}}{(p, r, f, m, s) \to (p', r''', f, m''', s)}$$

(Eval-call)
$$\dfrac{\begin{array}{c} m(p) = (\mathtt{call}\ \mathtt{r_d}) \qquad p' = r(\mathtt{r_d}) \qquad s \vdash \mathsf{entryJump}(p, p') \\ p, r, m, s \searrow^{\mathsf{SS}} p', r', m' \qquad r'' = r'[\mathsf{SP} \mapsto r'(\mathsf{SP}) + 1] \\ m'' = m'[r''(\mathsf{SP}) \mapsto p+1] \end{array}}{(p, r, f, m, s) \to (p', r'', f, m'', s)}$$

(Eval-returnback)
$$\dfrac{\begin{array}{c} m(p) = (\mathtt{ret}) \qquad p' = m(r(\mathsf{SP})) \qquad s \vdash \mathsf{entryJump}(p, p') \\ r' = r[\mathsf{SP} \mapsto r(\mathsf{SP}) - 1] \qquad p, r', m, s \searrow^{\mathsf{SS}} p', r'', m' \qquad s \vdash \mathsf{returnEntryPoint}(p') \end{array}}{(p, r, f, m, s) \to (p', r'', f, m', s)}$$

(Eval-return)
$$\dfrac{\begin{array}{c} m(p) = (\mathtt{ret}) \qquad p' = m(r(\mathsf{SP})) \qquad s \vdash \mathsf{exitJump}(p, p') \\ r' = r[\mathsf{SP} \mapsto r(\mathsf{SP}) - 1] \qquad p, r', m, s \searrow^{\mathsf{SS}} p', r'', m' \end{array}}{(p, r, f, m, s) \to (p', r'', f, m', s)}$$

Figure 4.7: Operational semantics of whole A+I programs.

Since our focus is on A+I programs P that are placed in protected memory and interact with arbitrary unprotected code, contexts model that unprotected code. Thus for any descriptor s, contexts $\mathbb{M}$ are partial memories with a hole: $\mathbb{M} = m[\cdot]$, where all addresses of $\mathbb{M}$ are unprotected. Formally, given s, $\forall a \in \mathtt{dom}(\mathbb{M}), s \vdash \mathsf{unprotected}(a)$. The hole models the possibility to combine a program P with the memory $\mathbb{M}$ iff they are compatible, denoted as $P \frown \mathbb{M}$, namely when the memories of P and $\mathbb{M}$ have disjoint domains. Let $\mathtt{dom}(\mathbb{M}) =$

$\mathtt{dom}(m)$ if $\mathbb{M} = m[\cdot]$; formally, $P \frown \mathbb{M}$ if $P = (m', s)$ and $\mathtt{dom}(m') \cap \mathtt{dom}(\mathbb{M}) = \emptyset$. If P and $\mathbb{M}$ are compatible, the hole of $\mathbb{M}$ can be filled with P in order to model interaction between P and $\mathbb{M}$. Formally, if $P \frown \mathbb{M}$ then $\mathbb{M}[(m', s)] = (m' + m, s)$.

Programs P_1 and P_2 are contextually equivalent, denoted as $P_1 \simeq^{\mathsf{A+I}} P_2$, when, for *all* contexts they interact with, P_1 diverges if and only if P_2 also diverges.

Definition 36 (Contextual equivalence) $P_1 \simeq^{\mathsf{A+I}} P_2 \triangleq \forall \mathbb{M}.\ P_1 \frown \mathbb{M} \wedge \mathbb{M}[P_1]\Uparrow \iff P_2 \frown \mathbb{M} \wedge \mathbb{M}[P_2]\Uparrow$.

An implication of this definition is that for P_1 and P_2 to be contextually equivalent they must have the same memory descriptor. For the sake of simplicity, always assume the compatibility of a protected program and the context it is plugged in, shortening the above definition to:

$$P_1 \simeq^{\mathsf{A+I}} P_2 \triangleq \forall \mathbb{M}.\ \mathbb{M}[P_1]\Uparrow \iff \mathbb{M}[P_2]\Uparrow$$

Example 8 (Contextually equivalent programs) *The following programs P_L and P_R write the values of* $\mathtt{r}_1$ *and* $\mathtt{r}_2$ *respectively to the protected address 150 (line 2) and then return 0 (line 3). Recall that the protected memory partition spans from address 100 to 200, with one entry point at address 100.*

```
1 100  movi r0 150
2 101  movs r0 r1
3 102  movi r0 0
4 103  ret
```

```
1 100  movi r0 150
2 101  movs r0 r2
3 102  movi r0 0
4 103  ret
```

The only difference between P_L and P_R is in the value stored at address 150. However, an unprotected program cannot read that value. Since that value does not affect the computation of P_L or P_R or the unprotected code, P_L and P_R are contextually equivalent. ⊡

Having defined the assembly language and its operational semantics, the thesis introduces the two different trace semantics for it. Trace equivalence is also introduced, it will be proven the same as contextual equivalence, thereby establishing full abstraction of the trace semantics.

Chapter 5

Fully Abstract Trace Semantics for A+I

Una bella dimostrazione ha la forma di una bella donna: larga in alto, si restringe al centro per riallargarsi un poco alla fine.

A beautiful proof has the shape of a beautiful woman: it starts wide at the top, narrowing at the middle, and widening again at the bottom.

A professor of Mathematical Logic, lecturing on type theory

As seen in Section 4.3, the description of the behaviour of protected A+I code can be rather burdensome since it is expressed in terms of the external code and each protected instruction. A trace semantics can give a simpler description of the behaviour of protected A+I code in terms of a set of *sequences of labels*. These labels capture how communication between protected and unprotected code happens and what is communicated. However, defining a fully abstract trace semantics is not trivial, so this chapter describes possible failures that can occur when devising one (Section 5.1).

As Curien stated [31], two ways to achieve full abstraction for a trace semantics exist, and in this chapter, two trace semantics are devised for the A+I language.

The first way is to change the operational semantics to restrict what is communicated to what is captured by the labels. This is achieved by restricting the ways in which communication is performed, for example by preventing reads and writes to unprotected memory, and the related semantics is called $\mathsf{Traces}^{\mathsf{S}}_{\mathsf{A+I}}$ (Section 5.2). This trace semantics captures the behaviour of a compiled J+E component and it will be used in the proof that the compilation scheme of Chapter 6 is secure.

The second way is to modify the labels so that they capture more precisely what is communicated between protected and unprotected code. In this case, labels should capture the values of all registers and flags as well as what protected code reads and writes in unprotected memory; the related semantics is called $\mathsf{Traces}^{\mathsf{L}}_{\mathsf{A+I}}$ (Section 5.3). This trace semantics captures the behaviour of any program located in the protected memory partition.

Finally, this chapter proves that both trace semantics are fully abstract (Section 5.4).

5.1 Failures of Full Abstraction

This section provides a first attempt to define a trace semantics for A+I, which is used to reveal possible full abstraction failures of the semantics. Then, it discusses adding writeout and readout labels for writes and reads to unprotected memory and how those labels affect the trace semantics. Each problem that is presented and that makes a trace semantics not fully abstract is followed by a description of how to address it.

Example 9 (Describing behaviour with traces) *Consider only the protected code of the snippet from Example 3.*

```
100  sub r0 r1    // protected code
101  movi r3 104
102  jl r3
103  ret
104  movi r0 0
105  ret
```

Since there is a single entry point to this code, located at address 100, this code represents a single function. A possible behaviour of this code can be expressed as follows (· is used to separate actions of the same trace):

$$\texttt{call } 100(\texttt{r}_0, \ldots, \texttt{r}_{11})? \cdot \texttt{ret } \texttt{r}_0!$$

⊡

To describe the behaviour of the code of Example 9 as a trace, we identify the actions that are observable from the point of view of code interacting with the snippet above: `call` and `ret`. These actions are the labels of the trace semantics; they are generated by `call` and `ret` instructions. Not all instructions generate a visible label in a trace, only those whose effect can be observed from the unprotected code.

Following is the syntax of labels of a trace semantics for protected A+I code.

$$\begin{array}{rl} \textit{Labels} & L ::= \alpha \mid \tau_i \\ \textit{Observable actions} & \alpha ::= \surd \mid \gamma? \mid \gamma! \\ \textit{Actions} & \gamma ::= \texttt{call}\ p(\overline{v}) \mid \texttt{ret}\ v \end{array}$$

A label λ can be an observable action α or a non-observable action τ. Decorations ? and ! indicate the direction of the observable action: from unprotected to protected code (?) or vice-versa (!). Address p is an address in memory, $\overline{v}$ is a list of the contents of all registers in a call and v indicates the contents of register $\texttt{r}_0$ in a return. Calls and returns executed by unprotected code are named *calls* and *returnbacks*, dually, if they are executed by protected code they are named *outcalls* and *returns* [10, 65].

This chapter aims at providing fully abstract trace semantics, implying that the trace semantics is the most precise possible from a behavioural characterisation perspective. Informally, a trace semantics is fully abstract when its labels capture all that is being communicated between the protected and the unprotected code but no more. A trace semantics following the discussion above would not be fully abstract due to a number of subtleties, as highlighted in Example 10.

Example 10 (Limitation of the aforementioned trace semantics) *Consider the two protected* A+I *programs below. Call the left one P_L and the right one P_R.*

```
1 100  sub r0 r1
2 101  movi r3 106
3 102  jl r3
4 103  movi r3 10
5 104  movs r3 r4
6 105  call r2
7 106  movi r11 41
8 107  movi r0 0
9 108  ret
```

```
1 100  sub r0 r1
2 101  movi r3 106
3 102  jl r3
4 103  movi r3 10
5 104  movs r3 r5
6 105  call r2
7 106  movi r11 42
8 107  movi r0 0
9 108  ret
```

Both P_L and P_R assign the result of $\texttt{r}_0 - \texttt{r}_1$ to $\texttt{r}_0$ (line 1). If the result of the operation is not less than 0 (line 3), they respectively write the contents of $\texttt{r}_4$ and $\texttt{r}_5$ to the unprotected address 10 (lines 4,5) and call the function whose *address is stored in* $\mathtt{r}_2$ *(line 6). Otherwise they assign different values to* $\mathtt{r}_{11}$ *(line 7) and return 0 (lines 8,9).*

With the trace semantics hinted at after Example 9, the behaviours of P_L *and* P_R *coincides as they generate the same traces. However,* P_L *and* P_R *can be distinguished by an external observer as described below, and the traces they generate should reflect this. Consider trace* $\overline{\alpha_1}$*, which is generated by both* P_L *and* P_R *(omitted details are indicated using* $\ldots$*).*

$$\overline{\alpha_1} = \mathtt{call}\ 100(1, 2, \ldots)? \cdot \mathtt{ret}\ 0!$$

$\overline{\alpha_1}$ *does not capture the different values contained in* $\mathtt{r}_{11}$ *(line 7), which, even if they are not the returned values of the function, still constitutes an observable difference between* P_L *and* P_R*.*

Trace $\overline{\alpha_2}$ *is also generated by both* P_L *and* P_R*.*

$$\overline{\alpha_2} = \mathtt{call}\ 100(2, 1, 40, \ldots)? \cdot \mathtt{call}\ 40(\ldots)!$$

$\overline{\alpha_2}$ *does not capture the different value written at address 10 (line 5), which also constitutes a observable difference between* P_L *and* P_R*.*

Since $\overline{\alpha_1}$ *and* $\overline{\alpha_2}$ *do not capture the observable differences between* P_L *and* P_R*, the trace semantics fails to be fully abstract.* ⊡

To summarise, this example points out that the contents of registers (and flags) can be used to differentiate protected programs as well as writes (and reads) to unprotected memory.

Let us now consider writing and reading to unprotected memory.

Writeouts

Protected code writing a value into the unprotected memory partition is called a *writeout*. Since such values can be observed by unprotected code, writeouts need to be captured in traces. This is done with a writeout label of the following form: $\mathtt{write}(a, v)$ stating what was written (v) and where (a). Following are the subtleties that need to be considered when introducing writeouts into the trace semantics (Examples 11 to 14). In the first case the problem is that the write is not observable, while in the second case the problem is the ordering of writeout labels. In the remaining cases the problem is that control is not returned to the external code, which means that it will not be able to detect the difference introduced by the writeout.

Example 11 (Invisible writeouts) *The following P_L and P_R read a value from an unprotected address 10 and 20, respectively (line 2), and then rewrite the same value back to the same address (line 3).*

```
100  movi r0 10
101  movl r1 r0
102  movs r0 r1
103  movi r0 0
104  ret
```

```
100  movi r0 20
101  movl r1 r0
102  movs r0 r1
103  movi r0 0
104  ret
```

The writeouts of P_L and P_R are invisible. *In fact, they do not alter the contents of unprotected memory, since address 10 (20, resp.) already contains the written value. Thus, P_L and P_R are contextually equivalent. However, they are not trace equivalent, since the following is a trace of P_L and not of P_R:*

$$\texttt{call}\ 100(\cdots)? \cdot \texttt{read}(10,0)\texttt{write}(10,0)\texttt{ret}\ 0!$$

Notice that if the readout were absent, the writeout would distinguish between P_L and P_R, as there are unprotected memories whose existing value at address 10 (20, resp.) differs from what is written by P_L or P_R.

To address this concern, the readout information must be accumulated and used to detect when a writeout is not introducing an observable difference in unprotected memory. $\boxdot$

Example 12 (Order independence of writeouts) *The following P_L and P_R write 0 to addresses 10 and 20 in unprotected memory (lines 4 and 5). The only difference between the two is that P_L writes to address 10 then to address 20 while P_R does the same writes in the opposite order.*

```
100  movi r1 10
101  movi r2 20
102  movi r0 0
103  movs r1 r0
104  movs r2 r0
105  ret
```

```
100  movi r1 10
101  movi r2 20
102  movi r0 0
103  movs r2 r0
104  movs r1 r0
105  ret
```

These programs are contextually equivalent, but if their labels are generated by the orders of the instructions, they will have different labels, since the following will be a trace of P_L and not of P_R.

$$\texttt{call}\ 100\ (\cdots)? \cdot \texttt{write}(10,0)\texttt{write}(20,0)\texttt{ret}\ 0!$$

To address this concern, writeouts need to be sorted when they are added to a trace. A more precise discussion over this solution is delayed until Example 20 since the solution is affected by the solutions of other $\boxdot$

Example 13 (No writeouts with termination) *The following P_L and P_R write 0 and 1 respectively to address 10 in unprotected memory (line 3) and then terminate (line 4).*

```
1 100  movi r1 10
2 101  movi r0 0
3 102  movs r1 r0
4 103  halt
```

```
1 100  movi r1 10
2 101  movi r0 1
3 102  movs r1 r0
4 103  halt
```

The only difference between P_L and P_R is the value written at address 10. However, the unprotected code cannot detect this difference since execution is halted before control is returned to it. Thus, P_L and P_R are contextually equivalent. If the writeout would appear in the traces, P_L and P_R would be trace-inequivalent, since the trace below would belong to P_L and not to P_R.

$$\texttt{call}\ 100\ (\cdots)? \cdot \texttt{write}(10, 0)\surd$$

Consequently, writeouts do not appear if the protected program halts afterwards. ⊡

Example 14 (Writeouts are not executable) *The following P_L and P_R set* $\texttt{r}_0$ *to 20 and 10 respectively (line 1), then write the instruction* $\texttt{jmp r}_0$ *at addresses 20 and 10 respectively (line 2). Finally, they jump to the instruction they just wrote (line 3).*

```
1 100  movi r0 20
2 101  movs r0 “jmp r0”
3 102  call r0
```

```
1 100  movi r0 10
2 101  movs r0 “jmp r0”
3 102  call r0
```

When $\texttt{r}_0$ *is set to 20 (resp. 10), the instruction* $\texttt{jmp r}_0$ *written at address 20 (resp. 10) will diverge when called. Thus, P_L and P_R are contextually equivalent, since no context can differentiate between them. However, P_L and P_R are trace inequivalent, since the following is a trace of P_L and not of P_R, since a trace of P_R would contain a* $\texttt{write}(10, \text{“}\texttt{jmp r}_0\text{”})\texttt{call}\ 10\ (10, \cdots)!$ *label.*

$$\texttt{call}\ 100\ (\cdots)? \cdot \texttt{write}(20, \text{“}\texttt{jmp r}_0\text{”})\texttt{call}\ 20\ (20, \cdots)!$$

The solution to this concern is to split the unprotected memory in a code and a data section and to allow writeouts only to the unprotected data section. A more complete analysis of the solution is delayed until Example 21. ⊡

Readouts

A *readout* occurs when protected code reads unprotected memory. Not all *PMA* implementations allow readouts, they are forbidden in some implementations [81] and discouraged by others [91,112]. When protected code can perform readouts, devising a fully abstract trace semantics is challenging. The readout label $\texttt{read}(a, v)$ states that a value v was read from address a. It is not obvious to decide when such a label should appear and the following examples present when the readout label should appear in traces or not (Examples 15 to 21).

Example 15 (Unobservable readouts) *Consider the two protected* A+I *programs below.*

```
1 100  movi r0 10
2 101  movl r1 r0
3 102  movi r1 0
4 103  movi r0 0
5 104  ret
```

```
1 100  movi r0 20
2 101  movl r1 r0
3 102  movi r1 0
4 103  movi r0 0
5 104  ret
```

P_L and P_R read the contents of unprotected addresses 10 and 20, respectively, and store the result in register $\texttt{r}_1$ (line 2), then they set registers $\texttt{r}_0$ and $\texttt{r}_1$ to 0 (lines 3,4) and return (line 5). In this case, the value read does not influence the behaviour of P_L or P_R, which behave the same, so the readout should not appear in their traces. ⊡

Example 16 (Readouts reduce to a constant) *Consider the two protected* A+I *programs below:*

```
1 100  movi r1 10
2 101  movl r0 r1
3 ···       // manipulate r0
4         // until it contains k
5 102  ret
```

```
1 100  movi r1 10
2 101  movi r0 k
3
4
5 102  ret
```

Here, P_L reads the contents of address 10 into $\texttt{r}_0$ (line 2), performs computations until $\texttt{r}_0$ contains a constant value k (omitted lines), independent of the value read, and then returns (line 5). P_R simply initialises $\texttt{r}_0$ to k (line 2) and returns (line 3).

These programs are contextually equivalent, both always return k, however, P_L also performs a readout. If this readout appears in traces, it would be a failure of full abstraction, since the traces of P_R do not have such a label. The problem here is that the omitted code of P_L always reduces the contents of $\texttt{r}_0$ to a constant, no matter what values it contained beforehand. The trace semantics *must be able to identify that the value read does not affect the execution of the program and thus not include the read label in this case.* ⊡

Example 17 (Observable readouts) *Consider the two protected* A+I *programs below.*

```
1 100  movi r0 10
2 101  movl r1 r0
3 102  movi r0 0
4 103  sub r0 r1
5 104  movi r0 108
6 105  je r0
7 106  movi r0 30
8 107  call r0
9 108  ret
```

```
1 100  movi r0 20
2 101  movl r1 r0
3 102  movi r0 0
4 103  sub r0 r1
5 104  movi r0 108
6 105  je r0
7 106  movi r0 30
8 107  call r0
9 108  ret
```

In this case P_L and P_R read the contents of unprotected addresses 10 and 20, respectively, in register $\mathtt{r}_1$ *(line 2). Then, if those values are less than 0 (lines 3, 4) they jump to address 108 (lines 5, 6) and return (line 9), otherwise they call to a function at address 30 (lines 7, 8).*

The value read in unprotected memory constitutes an observable difference between P_L and P_R, as it alters the execution flow. Thus, the readout value should itself be present in the trace. ⊡

The problem in this case is detecting when does a read affect the behaviour of a program. A read affects the behaviour of a program if some future behaviour of the program depends on the value read; when different values are read, the behaviour of the programs varies. On the other hand, if a read does not affect the behaviour, any value can be read and the program behaves the same. By viewing readout values as inputs, in the former case we can say that different inputs make a program have different behaviours (as in Example 17, while in the latter case different inputs do not vary the behaviour of a program (as in Examples 15 to 16).

The concept described above is analogous to *non-interference* [50, 126]. Non-interference is a property of systems whose input can be classified to be either low or high security (for non sensible and classified material respectively). A system is non-interfering if for a given set of low inputs it will produce the same low outputs, regardless of what the high level inputs are.

In this setting, if we treat readouts as high inputs and future traces as low outputs, we can apply non-interference notions to detect whether a readout affects a program. A readout does not affect a program if it is non-interfering, i.e., for any readout value (high input) the future traces (low output) do not vary. The trace semantics can use the non-interference information to decide whether a readout label should appear on traces or not. In Examples 15 to 16, the readouts are non-interfering, whatever value is read, the behaviour of the program does not vary, thus the trace semantics can exclude these readouts from traces. However, in Example 17 if the value read is 0, the program will behave differently than if it is not 0, so the readout is interfering. Here the trace semantics can tell that the readout must be included in the trace.

The main difference between the way non-interference is used in the literature and in this work is in the treatment of readout values. These values are in the external memory, thus intuitively low security, and they should be kept immutable. However, in order to apply non-interference correctly, they have to vary, thus they are regarded as high security.

Example 18 (Unobservable readouts after writeout) *The following P_L and P_R write 0 to address 10 (line 3), then P_R reads from address 10 (line 4).*

```
1 100  movi r1 10
2 101  movi r0 0
3 102  movs r1 r0
4
5 103  ret
```

```
1 100  movi r1 10
2 101  movi r0 0
3 102  movs r1 r0
4 103  movl r0 r1
5 104  ret
```

These programs are thus contextually equivalent, but the following is a trace of P_R and not of P_L.

$$\texttt{call}\ 100\ (\cdots)? \cdot \texttt{write}(10,0)\texttt{read}(10,0)\texttt{ret}\ 0!$$

To address this concern, reads to an address that was the subject of a writeout should not appear on traces. In fact, the readout value cannot be different from the writeout one, and that information is already known to protected programs.
⊡

Example 19 (Multiple readouts) *The following P_L and P_R read from address 10 in unprotected memory (line 2).*

```
1 100  movi r1 10
2 101  movl r1 r0
3
4 102  ret
```

```
1 100  movi r1 10
2 101  movl r1 r0
3 102  movl r1 r0
4 103  ret
```

The only difference between the two is that P_R reads from address 10 twice, but this does not affect its behaviour, since the same value is read. Thus, these programs are contextually equivalent, but the following is a trace of P_L and not *of P_R.*

$$\texttt{call}\ 100\ (\cdots)? \cdot \texttt{read}(10, v)\texttt{ret}\ 0!$$

To address this concern, multiple readouts to the same address should thus be filtered, only one must be present in the traces as long as no writeouts to the same address have occurred between the readouts. ⊡

Example 20 (Order independence of readouts) *In the following, P_L reads the contents of unprotected address 10 into register $\texttt{r}_1$ (line 2), then it reads the contents of unprotected address 20 into register $\texttt{r}_2$ (line 4). Finally, it calls to a function located at address 20 (line 5, the value of register $\texttt{r}_0$). P_R does the same, but first its reads happen in the reversed order: first it reads address 20 into register $\texttt{r}_2$ (line 2), then address 10 into register $\texttt{r}_1$ (line 2).*

```
1 100  movi r3 10
2 101  movl r1 r3
3 102  movi r0 20
4 103  movl r2 r0
5 104  call r0
```

```
1 100  movi r0 20
2 101  movl r2 r0
3 102  movi r3 10
4 103  movl r1 r3
5 104  call r0
```

These programs are contextually equivalent, but the traces they create are different. The order in which the readouts are executed and accumulated on the traces makes it so that the following trace is generated by P_L and not by P_R.

$$\texttt{call}\ (\cdots)? \cdot \texttt{read}(10, v)\texttt{read}(20, v')\texttt{call}\ 20\ (20, v, v', 10, \cdots)!$$

To address this and the concern of Example 12 readouts and writeouts can be sorted based on the address to which the operation is performed.

This introduces a sort of normal form for traces, which consist of a sorted prefix of readouts and writeouts followed by a call or a return. The normal form effectively merges the solutions to Examples 18 to 19. ⊡

Example 21 (Readouts are not executable) *In the following, P_L always halts while P_R reads the contents of address 10 into $\texttt{r}_1$ (lines 1, 2). If the value read is not an instruction (line 3, omitted for the sake of simplicity), P_R jumps there (line 4), otherwise it halts (line 6).*

```
100  halt
```

```
100  movi r3 107
101  movi r0 10
102  movl r1 r0
// load the encoding for halt in
     r2
103  movi r2  "halt"
104  cmp r1 r2
// if address 10 contains halt
105  je r3
106  call r1 // jump to address 10
107  halt // otherwise halt
```

These two programs are contextually equivalent: they always terminate. However, P_R generates the following trace, which P_L does not:

$$\texttt{call}\ (\cdots)? \cdot \texttt{read}(10, v)\texttt{call}\ 10\ (10, v, \cdots)!$$

The problem is that the trace above will always be followed by termination (in unprotected code), which unprotected code cannot observe. This is due to P_R reading executable unprotected code and P_R behaving differently based on the value read. ⊡

To address this concern and Example 14 unprotected code is split in a code and a data section, just as protected code is. Writeouts and readouts can only be performed on the data section of unprotected code, so protected code cannot read nor write executable unprotected code.

From the threat modeling perspectimes this assumption somewhat reduces the attacker's power, since she is not able to execute the values written by the protected code. However, this assumption seems reasonable, since most times we are interested in modelling the behaviour of code that uses readouts for parameters and not to execute readout values. Future work will consider writeouts and readouts of executable unprotected code.

To include writeouts and readouts in traces, a label for each operation must be added to the syntax of labels. These operations must not be done on the executable part of unprotected memory. These labels must be sorted when added to a trace. Moreover, readout labels must be added only when they influence future computations and readout labels must be added only if they are not followed by termination.

5.2 $\mathsf{Traces}^{\mathsf{S}}_{\mathsf{A+I}}$: Changes to the Semantics

This section describes $\mathsf{Traces}^{\mathsf{S}}_{\mathsf{A+I}}$, the trace semantics that models the behaviour of securely-compiled J+E components.

A notion of execution states is required for the trace semantics as it was for the operational semantics. Execution states for $\mathsf{Traces}^{\mathsf{S}}_{\mathsf{A+I}}$, denoted as Θ, can be (unk, m, s), a state that models when code is executing in unprotected memory [65]. Moreover, Θ can be a state like those of the operational semantics, except that the memory in Θ is not the whole memory, just the protected partition. So, the memory m of (p, r, f, m, s) spans only the protected memory partition indicated by memory descriptor s. Formally: $\Theta ::= (\mathsf{unk}, m, s) \mid (p, r, f, m, s)$, such that $\forall a \in \mathtt{dom}(m), s \vdash \mathsf{protected}(a)$.

The notion of initial state is required for the trace semantics as it was for the operational semantics.

Definition 37 (Initial state for traces) *The initial state for traces of a program* (m, s)*, denoted as* $\Theta_0(m, s)$*, is the state* (unk, m, s)*.*

The semantics of protected programs is changed w.r.t. the semantics specified in Section 4.2 as follows (Figure 5.1):

- when the program counter jumps between the protected and the unprotected memory partitions, or vice-versa, flags are set to 0 (Rule Stack-out-to-in' and Stack-in-to-out');
- in case of a `return`, all registers but $\mathtt{r}_0$ are also set to 0 (Rule Eval-return');
- readouts and writeouts are prohibited (Rule Aux-write-1' and Aux-read-2' replace the access control rules with the same name).

These changes do not limit the expressivity of the language. They ensure communication between protected and unprotected code happens in a specific fashion, namely only via function calls (with 12 parameters) and returns (with 1 returned value).

Following are the labels of $\mathsf{Traces}^{\mathsf{S}}_{\mathsf{A+I}}$, they include those presented in Section 2.2. Observable actions include a tick $\surd$ indicating that the evaluation has terminated. Flags do not appear in traces because they are always set to 0, as are all registers but $\mathtt{r}_0$ in case of a return. Readouts and writeouts are prohibited so there are

$$\frac{s \vdash \mathsf{unprotectedCode}(p) \quad s \vdash \mathsf{unprotectedData}(a)}{s \vdash \mathsf{writeAllowed}(p, a)}\ (\mathsf{Aux\text{-}write\text{-}1'})$$

$$\frac{s \vdash \mathsf{unprotectedCode}(p) \quad s \vdash \mathsf{unprotectedData}(a)}{s \vdash \mathsf{readAllowed}(p, a)}\ (\mathsf{Aux\text{-}read\text{-}2'})$$

$$\frac{\begin{array}{c} s \vdash \mathsf{entryJump}(p, p') \\ m' = m[\mathsf{SP_{ext}} \mapsto r(\mathsf{SP})] \\ r' = r[\mathsf{SP} \mapsto m(\mathsf{SP_{sec}})] \\ f' = [\mathsf{ZF} \mapsto 0; \mathsf{SF} \mapsto 0] \\ s \vdash \mathsf{unprotected}\ r(\mathsf{SP}) \\ s \vdash \mathsf{protected}\ r'(\mathsf{SP}) \end{array}}{p, r, f, m, s \searrow^{\mathsf{SS}} p', r', f', m'}\ (\mathsf{Stack\text{-}out\text{-}to\text{-}in'})$$

$$\frac{\begin{array}{c} s \vdash \mathsf{exitJump}(p, p') \\ m' = m[\mathsf{SP_{sec}} \mapsto r(\mathsf{SP})] \\ r' = r[\mathsf{SP} \mapsto m(\mathsf{SP_{ext}})] \\ f' = [\mathsf{ZF} \mapsto 0; \mathsf{SF} \mapsto 0] \\ s \vdash \mathsf{protected}\ r(\mathsf{SP}) \\ s \vdash \mathsf{unprotected}\ r'(\mathsf{SP}) \end{array}}{p, r, f, m, s \searrow^{\mathsf{SS}} p', r', f', m'}\ (\mathsf{Stack\text{-}in\text{-}to\text{-}out'})$$

$$\frac{\begin{array}{c} m(p) = (\texttt{ret}) \qquad p' = m(r(\mathsf{SP})) \qquad s \vdash \mathsf{exitJump}(p, p') \\ r' = r[\mathsf{SP} \mapsto r(\mathsf{SP}) - 1; \mathtt{r_i} \mapsto 0_{\mathsf{i}=1..11}] \qquad p, r', f, m, s \searrow^{\mathsf{SS}} p', r'', f', m' \end{array}}{(p, r, f, m, s) \to (p', r'', f', m', s)}\ (\mathsf{Eval\text{-}return'})$$

Figure 5.1: Changes to auxiliary functions and to the operational semantics for $\mathsf{Traces}^{\mathsf{S}}_{\mathsf{A+I}}$.

no labels that capture them.

Labels	$L ::= \alpha \mid \tau_i$
Observable actions	$\alpha ::= \surd \mid \gamma? \mid \gamma!$
Actions	$\gamma ::= \texttt{call}\ p\ (r) \mid \texttt{ret}\ p\ r(\mathtt{r_0})$

The following rules define the relation $\xrightarrow{\lambda}\!\!\!\!\rightarrow \subseteq \Theta \times \widehat{\lambda} \times \Theta$ which describes how labels are generated (Figure 5.2). Internal instructions, generated by a $\xrightarrow{i}$ transition, do not produce a visible label (Rule Trace-s-internal). If a state is stuck, then the label for termination is produced (Rule Trace-s-termination). A call to an entry point generates a call label (Rule Trace-s-call) while a return to the returnback entry point generates a returnback label (Rule Trace-s-returnback). Calling to an unprotected address generates an outcall label (Rule Trace-s-outcall), while returning to any unprotected address generates a return label (Rule Trace-s-return).

The reflexive and transitive closure of $\xrightarrow{\lambda}\!\!\!\!\rightarrow$, denoted with $\overset{\overline{\alpha}}{\Longrightarrow}\!\!\!\!\rightarrow \subseteq \Theta \times \overline{\alpha} \times \Theta$, is responsible for the accumulation of labels into traces (Figure 5.3).

The $\mathsf{Traces}^{\mathsf{S}}_{\mathsf{A+I}}$ traces of a program P are defined as follows:

$$\mathsf{Traces}^{\mathsf{S}}_{\mathsf{A+I}}(P) = \{\overline{\alpha} \mid \exists \Theta.\ \Theta_0(P) \overset{\overline{\alpha}}{\Longrightarrow}\!\!\!\!\rightarrow \Theta\}$$

$$\text{(Trace-s-internal)}\quad \frac{(p,r,f,m,s) \xrightarrow{i} (p',r',f',m',s) \qquad s \vdash \mathsf{intJump}(p,p')}{(p,r,f,m,s) \xrightarrow{\tau_i}\!\!\!\rightarrow (p',r',f',m',s)}$$

$$\text{(Trace-s-termination)}\quad \frac{(p,r,f,m,s) \xrightarrow{i} (p',r',f',m',s) \qquad (p',r',f',m',s)^{\perp}}{(p,r,f,m,s) \xrightarrow{\surd}\!\!\!\rightarrow (p',r',f',m',s)}$$

$$\text{(Trace-s-call)}\quad \frac{s \vdash \mathsf{entryPoint}(p) \qquad f = [\mathsf{SF} \mapsto 0; \mathsf{ZF} \mapsto 0]}{(\mathsf{unk},m,s) \xrightarrow{\texttt{call}\ p\ (r)?}\!\!\!\rightarrow (p,r,f,m,s)}$$

$$\text{(Trace-s-returnback)}\quad \frac{s \vdash \mathsf{returnEntryPoint}(p) \qquad f = [\mathsf{SF} \mapsto 0; \mathsf{ZF} \mapsto 0]}{(\mathsf{unk},m,s) \xrightarrow{\texttt{ret}\ p\ r(\texttt{r}_0)?}\!\!\!\rightarrow (p,r,f,m,s)}$$

$$\text{(Trace-s-outcall)}\quad \frac{s \vdash \mathsf{exitJump}(p,p') \qquad m(p) = (\texttt{call}\ p') \qquad m' = m[r(\mathsf{SP}) \mapsto p+1]}{(p,r,f,m,s) \xrightarrow{\texttt{call}\ p'\ (r)!}\!\!\!\rightarrow (\mathsf{unk},m',s)}$$

$$\text{(Trace-s-return)}\quad \frac{p' \in 0...2^{\ell} \qquad s \vdash \mathsf{exitJump}(p,p') \qquad m(p) = (\texttt{ret})}{(p,r,f,m,s) \xrightarrow{\texttt{ret}\ p'\ r(\texttt{r}_0)!}\!\!\!\rightarrow (\mathsf{unk},m,s)}$$

Figure 5.2: Rules of the $\mathsf{Traces}^{\mathsf{S}}_{\mathsf{A+I}}$ trace semantics.

$$\text{(Trace-s-refl)}\quad \frac{}{\Theta \overset{\epsilon}{\Longrightarrow}\!\!\!\!\rightarrow \Theta} \qquad \text{(Trace-s-tau-i)}\quad \frac{\Theta \xrightarrow{\tau_i}\!\!\!\rightarrow \Theta'}{\Theta \overset{\epsilon}{\Longrightarrow}\!\!\!\!\rightarrow \Theta'}$$

$$\text{(Trace-s-trans)}\quad \frac{\Theta \overset{\overline{\alpha}}{\Longrightarrow}\!\!\!\!\rightarrow \Theta'' \qquad \Theta'' \overset{\overline{\alpha}'}{\Longrightarrow}\!\!\!\!\rightarrow \Theta'}{\Theta \overset{\overline{\alpha}\cdot\overline{\alpha}'}{\Longrightarrow}\!\!\!\!\rightarrow \Theta'} \qquad \text{(Trace-s-action)}\quad \frac{\Theta \xrightarrow{\alpha}\!\!\!\rightarrow \Theta'}{\Theta \overset{\alpha}{\Longrightarrow}\!\!\!\!\rightarrow \Theta'}$$

Figure 5.3: Reflexive and transitive closure of the $\mathsf{Traces}^{\mathsf{S}}_{\mathsf{A+I}}$ trace semantics rules.

5.3 $\mathsf{Traces}^{\mathsf{L}}_{\mathsf{A+I}}$: Expressive Labels

This section presents $\mathsf{Traces}^{\mathsf{L}}_{\mathsf{A+I}}$, a trace semantics that changes the labels of Section 2.2 to include all possible observable behaviour, including readouts and writeouts. The semantics used for $\mathsf{Traces}^{\mathsf{L}}_{\mathsf{A+I}}$ is the operational semantics of A+I, as presented in Section 4.2.

The states of the $\mathsf{Traces}^{\mathsf{L}}_{\mathsf{A+I}}$ semantics are indicated with Θ, they do not change from the definition given for the $\mathsf{Traces}^{\mathsf{S}}_{\mathsf{A+I}}$ semantics. The syntax of labels, however, changes as indicated below, including a readout and a writeout label.

$$
\begin{array}{rl}
\textit{Labels} & \mathfrak{l} ::= \tau \mid \mathfrak{a} \\
\textit{Observable actions} & \mathfrak{a} ::= \mathfrak{g}? \mid \mathfrak{d}! \mid \surd \\
\textit{Actions} & \mathfrak{g} ::= \texttt{call}\ p(r; f) \mid \texttt{ret}\ p(r; f) \\
\textit{Prefixable actions} & \mathfrak{d} ::= \mathfrak{g} \mid \mathfrak{o}(a, v).\mathfrak{d} \\
\textit{Prefixes} & \mathfrak{o} ::= \texttt{read} \mid \texttt{write}
\end{array}
$$

To ensure that the issues of Examples 11 to 12 and Examples 18 to 20 (Section 5.1 and 5.1) do not arise, $\mathfrak{d}!$ labels are converted to a normal form.

The normal form of $\mathfrak{d}!$ labels is achieved by applying the rewrite rules presented in Figure 5.4. Equations (Constraint-write) to (Constraint-read) ensure that

$$
\begin{array}{rlr}
\texttt{write}(a,v)\texttt{read}(a,v') \Rightarrow v = v' & & (\textsf{Constraint-write}) \\
\texttt{read}(a,v)\texttt{read}(a,v') \Rightarrow v = v' & & (\textsf{Constraint-read}) \\
\texttt{write}(a,v)\texttt{write}(a',v') \rightsquigarrow \texttt{write}(a',v')\texttt{write}(a,v) & \text{if } a' < a & (\textsf{Write-order}) \\
\texttt{read}(a,v)\texttt{read}(a',v') \rightsquigarrow \texttt{read}(a',v')\texttt{read}(a,v) & \text{if } a' < a & (\textsf{Read-order}) \\
\texttt{read}(a,v)\texttt{write}(a',v') \rightsquigarrow \texttt{write}(a',v')\texttt{read}(a,v) & \text{if } a' < a & (\textsf{WR-order}) \\
\texttt{write}(a,v)\texttt{read}(a',v') \rightsquigarrow \texttt{read}(a',v')\texttt{write}(a,v) & \text{if } a' < a & (\textsf{RW-order}) \\
\texttt{write}(a,v)\texttt{read}(a,v) \rightsquigarrow \texttt{write}(a,v) & & (\textsf{Write-no-read}) \\
\texttt{read}(a,v)\texttt{write}(a,v) \rightsquigarrow \texttt{read}(a,v) & & (\textsf{Read-no-write}) \\
\texttt{write}(a,v)\texttt{write}(a,v') \rightsquigarrow \texttt{write}(a,v') & & (\textsf{Write-drop}) \\
\texttt{read}(a,v)\texttt{read}(a,v) \rightsquigarrow \texttt{read}(a,v) & & (\textsf{Read-drop})
\end{array}
$$

Figure 5.4: Rewrite rules to reduce a $\mathfrak{d}!$ label to its normal form.

labels generated by the semantics are consistent [100]. Equation (Write-order), (Read-order), (WR-order) and (RW-order) ensure the prefix of reads and writes are sorted based on the address field. If two actions are performed at the same address, their order is the same as the order in which the program performed those actions. Equations (Write-no-read) to (Read-no-write) ensure that reading the same writeout value (resp. writing the same readout value) does not appear in labels. Equations (Write-drop) to (Read-drop) eliminate all but the last writeouts and readouts to the same address.

The rewrite rules of Figure 5.4 are convergent so their application always returns a unique result (Theorem 1 in Section 5.3.1). We can thus define the normal-form function $\mathsf{norm}(\cdot)$ to denote the application of those rewrite rules. This function inputs a $\mathfrak{d}$ label and returns it in normal form, i.e., a sequence of $\texttt{write}(a,v)$ and $\texttt{read}(a,v)$ label sorted on the address parameter a.

The rules that define the single label relation $\xrightarrow{\mathfrak{l}}\!\!\!\!\rightarrow \subseteq \Theta \times \widehat{\mathfrak{l}} \times \Theta$ (Figure 5.5) rely on the semantics presented in Section 4.2. Rules for generating $\texttt{call}$, $\texttt{return}$ and τ

(Trace-internal)

$$\dfrac{(p,r,f,m,s) \xrightarrow{i} (p',r',f',m',s) \qquad s \vdash \mathsf{intJump}(p,p')}{(p,r,f,m,s) \xrightarrow{\tau_i}\!\!\!\!\rightarrow (p',r',f',m',s)}$$

(Trace-termination)

$$\dfrac{(p,r,f,m,s) \xrightarrow{i} (p',r',f',m',s) \qquad (p',r',f',m',s)^{\perp}}{(p,r,f,m,s) \xrightarrow{\surd}\!\!\!\!\rightarrow (p',r',f',m',s)}$$

(Trace-call)

$$\dfrac{s \vdash \mathsf{entryPoint}(p)}{(\mathsf{unk},m,s) \xrightarrow{\texttt{call}\ p(r;f)?}\!\!\!\!\rightarrow (p,r,f,m,s)}$$

(Trace-returnback)

$$\dfrac{s \vdash \mathsf{returnEntryPoint}(p)}{(\mathsf{unk},m,s) \xrightarrow{\texttt{ret}\ p(r;f)?}\!\!\!\!\rightarrow (p,r,f,m,s)}$$

(Trace-outcall)

$$\dfrac{s \vdash \mathsf{exitJump}(p,p') \qquad m(p) = (\texttt{call}\ p') \qquad m' = m[r(\mathsf{SP}) \mapsto p+1]}{(p,r,f,m,s) \xrightarrow{\texttt{call}\ p'(r;f)!}\!\!\!\!\rightarrow (\mathsf{unk},m',s)}$$

(Trace-return)

$$\dfrac{m(p) = (\texttt{ret}) \qquad p' \in 0 ... 2^{\ell} \qquad s \vdash \mathsf{exitJump}(p,p')}{(p,r,f,m,s) \xrightarrow{\texttt{ret}\ p'(r;f)!}\!\!\!\!\rightarrow (\mathsf{unk},m,s)}$$

(Trace-tau-compression)

$$\dfrac{\Theta \xrightarrow{\tau}\!\!\!\!\rightarrow \Theta' \qquad \Theta' \xrightarrow{\mathfrak{a}}\!\!\!\!\rightarrow \Theta''}{\Theta \xrightarrow{\mathfrak{a}}\!\!\!\!\rightarrow \Theta''}$$

(Trace-writeout)

$$\dfrac{\begin{array}{c} m(p) = \texttt{movs}\ \texttt{r}_{\texttt{d}}\ \texttt{r}_{\texttt{s}} \qquad s \vdash \mathsf{intJump}(p,p+1) \qquad s \vdash \mathsf{unprotectedData}(a) \qquad a = r(\texttt{r}_{\texttt{d}}) \\ v = r(\texttt{r}_{\texttt{s}}) \qquad (p+1,r,f,m,s) \xrightarrow{\mathfrak{d}!}\!\!\!\!\rightarrow (\mathsf{unk},m,s) \qquad v' \in \mathcal{W} \end{array}}{(p,r,f,m,s) \xrightarrow{\texttt{write}(a,v)\mathfrak{d}!}\!\!\!\!\rightarrow (\mathsf{unk},m,s)}$$

(Trace-writeout-termination)

$$\dfrac{\begin{array}{c} m(p) = \texttt{movs}\ \texttt{r}_{\texttt{d}}\ \texttt{r}_{\texttt{s}} \qquad s \vdash \mathsf{intJump}(p,p+1) \qquad s \vdash \mathsf{unprotectedData}(a) \\ (p+1,r,f,m,s) \xrightarrow{\overline{\mathfrak{o}(a,v)}\surd}\!\!\!\!\rightarrow (p',r',f',m',s') \end{array}}{(p,r,f,m,s) \xrightarrow{\overline{\mathfrak{o}(a,v)}\surd}\!\!\!\!\rightarrow (p',r',f',m',s')}$$

(Trace-readout)

$$\dfrac{\begin{array}{c} m(p) = \texttt{movl}\ \texttt{r}_{\texttt{d}}\ \texttt{r}_{\texttt{s}} \qquad s \vdash \mathsf{intJump}(p,p+1) \qquad s \vdash \mathsf{unprotectedData}(a) \qquad a = r(\texttt{r}_{\texttt{s}}) \\ v \in \mathcal{W} \qquad r'' = r[\texttt{r}_{\texttt{d}} \mapsto v] \qquad (p+1,r'',f,m,s) \xrightarrow{\mathfrak{d}!}\!\!\!\!\rightarrow (\mathsf{unk},m,s) \end{array}}{(p,r,f,m,s) \xrightarrow{\texttt{read}(a,v)\mathfrak{d}!}\!\!\!\!\rightarrow (\mathsf{unk},m,s)}$$

Figure 5.5: Rules for the $\mathsf{Traces}^{\mathsf{L}}_{\mathsf{A+I}}$ trace semantics.

labels resemble the rules for the $\mathsf{Traces}^{\mathsf{S}}_{\mathsf{A+I}}$ semantics. Rule Trace-tau-compression ensures that τ labels are not accumulated, so readout and writeout labels are not interleaved with τs. For writeouts, Rule Trace-writeout ensures writeout labels are always created, dually, for readouts, Rule Trace-readout ensures readout labels are always created. Rule Trace-writeout-termination addresses Example 13, so no writeout label is created when a program terminates.

The reflexive transitive closure of the $\xrightarrow{\mathfrak{l}}\!\!\!\!\rightarrow$ relation is captured by relation $\xRightarrow{\overline{\mathfrak{a}}} \subseteq \Theta \times \widehat{\overline{\mathfrak{a}}} \times \Theta$ (Figure 5.6). The only difference with the way this is performed with regards to $\mathsf{Traces}^{\mathsf{S}}_{\mathsf{A+I}}$ (Figure 5.3) is that when a label is produced in $\mathsf{Traces}^{\mathsf{L}}_{\mathsf{A+I}}$, it is converted to a normal form via the $\mathsf{norm}(\cdot)$ function and stripped of its non-interfering reads via the $\mathsf{StripNI}(\cdot)$ function (Figure 5.7 defined below).

$$\text{(Trace-l-refl)}\quad \frac{}{\Theta \xRightarrow{\epsilon} \Theta} \qquad \text{(Trace-l-tau-i)}\quad \frac{\Theta \xrightarrow{\tau_i}\!\!\!\!\rightarrow \Theta'}{\Theta \xRightarrow{\epsilon} \Theta'} \qquad \text{(Trace-l-trans)}\quad \frac{\Theta \xRightarrow{\overline{\mathfrak{a}}} \Theta'' \qquad \Theta'' \xRightarrow{\overline{\mathfrak{a}}'} \Theta'}{\Theta \xRightarrow{\overline{\mathfrak{a}}\cdot\overline{\mathfrak{a}}'} \Theta'}$$

$$\text{(Trace-l-action)}\quad \frac{\Theta \xrightarrow{\mathfrak{a}}\!\!\!\!\rightarrow \Theta' \qquad \mathsf{StripNI}(\Theta, \mathsf{norm}(\mathfrak{a})) = \mathfrak{a}'}{\Theta \xRightarrow{\mathfrak{a}'} \Theta'}$$

Figure 5.6: Reflexive and transitive closure of relation $\longrightarrow\!\!\!\!\rightarrow$ for $\mathsf{Traces}^{\mathsf{L}}_{\mathsf{A+I}}$.

The trace semantics of a state is defined as follows:

$$\mathsf{Tr\text{-}state}(\Theta) = \{\overline{\mathfrak{a}} \mid \exists\Theta'.\Theta \xRightarrow{\overline{\mathfrak{a}}} \Theta'\}$$

Thus, the $\mathsf{Traces}^{\mathsf{L}}_{\mathsf{A+I}}$ traces of a program P are defined as the traces of its initial state:

$$\mathsf{Traces}^{\mathsf{L}}_{\mathsf{A+I}}(P) = \mathsf{Tr\text{-}state}(\Theta_0(P))$$

The greatest concern when adding readouts is detecting whether a readout is non-interfering, as explained in Examples 15 to 17. In fact, non-interfering readouts must not have a corresponding label in traces. To understand whether a readout to a certain address is non-interfering, we rely on judgment $\mathsf{NI}(\Theta, a)$. That judgment tells whether an address a is non-interfering for a state Θ as follows. Consider Θ that performs different reductions to different states Θ' and Θ''. These reductions generate labels that include a readout of different values v and w to the same address a. If the behaviour of the different states Θ' and Θ'' is the same (i.e., they have the same trace semantics), then the readouts to a are non-interfering. In fact, if the readout would affect the behaviour of the code (i.e., if it were interfering), a difference in the traces would be detected.

Formally:

$$\begin{aligned}\mathsf{NI}(\Theta, a) \triangleq \forall v, w.\quad & \Theta \xRightarrow{\mathfrak{a}_1} \Theta' \text{ and } \Theta \xRightarrow{\mathfrak{a}_2} \Theta'' \\ & \text{and } \mathfrak{a}_1 = \overline{\mathfrak{o}(a', v')}\mathtt{read}(a, v)\mathfrak{d}! \\ & \text{and } \mathfrak{a}_2 = \overline{\mathfrak{o}(a', v')}\mathtt{read}(a, w)\mathfrak{d}! \\ & \text{and } \mathsf{Tr\text{-}state}(\Theta') = \mathsf{Tr\text{-}state}(\Theta'')\end{aligned}$$

The definition of $\mathsf{NI}(\cdot)$ relies on the formalisation of $\mathsf{Tr\text{-}state}(\Theta)$ which returns the set of traces that can be generated from Θ. $\mathsf{Tr\text{-}state}(\cdot)$ is used to access the behaviour of the program after either value is read from address a, no difference can be found there for the readout to be non-interfering. It is not sufficient to check the single immediate action $\mathfrak{d}$ following the readout, as the readout value could be stored in memory and be used only for successive computations. The prefix $\overline{\mathfrak{o}(a', v')}$ makes it possible to identify a readout that happens at any point during the first action.

Note that the definition of $\mathsf{Tr\text{-}state}(\cdot)$ and that of $\mathsf{NI}(\cdot)$ are mutually recursive. However, they are still well-founded since $\mathsf{Tr\text{-}state}(\cdot)$ uses $\mathsf{NI}(\cdot)$ when filtering a label $\mathfrak{d}!$ generated as $\Theta \xRightarrow{\mathfrak{d}!} \Theta'$ and then $\mathsf{NI}(\cdot)$ relies on $\mathsf{Tr\text{-}state}(\cdot)$ on the traces generated from Θ' onwards.

With this information, define a function $\mathsf{StripNI}(\Theta, \mathfrak{a})$ that returns $\mathfrak{a}'$ which is $\mathfrak{a}$ stripped of its non-interfering reads, provided that $\mathfrak{a}$ is generated from Θ (Figure 5.7). Since this function preserves the ordering of the labels in $\mathfrak{a}$, when applied to labels in normal form it still produces labels in normal form.

$$\begin{aligned}\mathsf{StripNI}(\Theta, \mathfrak{g}) &= \mathfrak{g} \\ \mathsf{StripNI}(\Theta, \mathtt{write}(a, v)\mathfrak{d}) &= \mathtt{write}(a, v)\mathfrak{d}' && \text{if } \mathsf{StripNI}(\Theta, \mathfrak{d}) = \mathfrak{d}' \\ \mathsf{StripNI}(\Theta, \mathtt{read}(a, v)\mathfrak{d}) &= \mathfrak{d}' && \text{if } \mathsf{StripNI}(\Theta, \mathfrak{d}) = \mathfrak{d}' \text{ and } \mathsf{NI}(\Theta, a) \\ \mathsf{StripNI}(\Theta, \mathtt{read}(a, v)\mathfrak{d}) &= \mathtt{read}(a, v)\mathfrak{d}' && \text{if } \mathsf{StripNI}(\Theta, \mathfrak{d}) = \mathfrak{d}' \text{ and } \neg\mathsf{NI}(\Theta, a)\end{aligned}$$

Figure 5.7: Function to strip a label of its non-interfering reads.

This trace semantics addresses all examples presented in Section 5.1. Example 11 and 12 are addressed by the rewrite rules of Figure 5.4. Example 13 is addressed by Rule Trace-writeout-termination. Example 14 is addressed by the auxiliary function Rule Aux-write-1. Example 15 and 16 are addressed by the $\mathsf{StripNI}(\cdot, \cdot)$ function. Example 17 is addressed by Rule Trace-readout. Examples 18 to 20 are addressed by the rewrite rules of Figure 5.4. Example 21 is addressed by the auxiliary function Rule Aux-read-2.

5.3.1 Properties of the Rewrite Rules of Figure 5.4

The rewrite rules of Figure 5.4 are confluent (Lemma 1) and terminating (Lemma 2), thus they are convergent (Theorem 1). This implies that when applied to a prefix $\overline{\mathfrak{o}(a,v)}$, they will always return its unique normal form.

In the following, denote a prefix sequence $\overline{\mathfrak{o}(a,v)}$ with $\mathfrak{p}$.

Lemma 1 (Confluence) *The rewrite rules are confluent. For any* $\mathfrak{p}$*, for all* $\mathfrak{p}'$ *and* $\mathfrak{p}''$ *such that* $\mathfrak{p} \rightsquigarrow \mathfrak{p}'$ *and* $\mathfrak{p} \rightsquigarrow \mathfrak{p}''$*, there exists* $\mathfrak{p}'''$ *such that* $\mathfrak{p}' \rightsquigarrow^* \mathfrak{p}'''$ *and* $\mathfrak{p}'' \rightsquigarrow^* \mathfrak{p}'''$*.*

Proof of Lemma 1. This proof proceeds by induction over the length of $\overline{\mathfrak{o}(a,v)}$.

Base case (length 0 or 1) No reduction rules apply, so the theorem holds.

Inductive case (length > 1) For any combination of the first two actions of the prefix ($\mathfrak{o}_0(a_0,v_0)$ and $\mathfrak{o}_1(a_1,v_1)$) only one rule is applicable, as presented in the case analysis below. Thus, $\mathfrak{p}'$ and $\mathfrak{p}''$ are the same and so both flow into the same $\mathfrak{p}'''$.

$\texttt{read}(a,v)\texttt{read}(a,v)$ only Equation (Read-drop) applies.

$\texttt{read}(a,v)\texttt{read}(a,w)$ **and** $v \neq w$ firstly, Equation (Constraint-read) ensures that $v = w$, then only Equation (Read-drop) applies.

$\texttt{read}(a,v)\texttt{read}(b,v)$ **and** $a \neq b$ in this case if $a < b$ no rule apply, while if $a > b$ only Equation (Read-order) applies.

$\texttt{read}(a,v)\texttt{write}(a,v)$ only Equation (Read-no-write) applies.

$\texttt{read}(a,v)\texttt{write}(a,w)$ **and** $v \neq w$ no rule applies.

$\texttt{read}(a,v)\texttt{write}(b,v)$ **and** $a \neq b$ in this case if $a < b$ no rule apply, while if $a > b$ only Equation (RW-order) applies.

$\texttt{write}(a,v)\texttt{write}(a,v)$ only Equation (Write-drop) applies.

$\texttt{write}(a,v)\texttt{write}(a,w)$ **and** $v \neq w$ only Equation (Write-drop) applies.

$\texttt{write}(a,v)\texttt{write}(b,v)$ **and** $a \neq b$ in this case if $a < b$ no rule apply, while if $a > b$ only Equation (Write-order) applies.

$\texttt{write}(a,v)\texttt{read}(a,v)$ only Equation (Write-no-read) applies.

$\texttt{write}(a,v)\texttt{read}(a,w)$ **and** $v \neq w$ firstly, Equation (Constraint-read) ensures that $v = w$, then only Equation (Write-no-read) applies.

$\texttt{write}(a,v)\texttt{read}(b,v)$ **and** $a \neq b$ and $a \neq b]$ in this case if $a < b$ no rule apply, while if $a > b$ only Equation (WR-order) applies.

□

Lemma 2 (Termination) *The rewrite rules are terminating: for any prefix, all possible sequences of application of the rewrite rules to $\overline{\mathfrak{o}(a,v)}$ are finite.*

Proof of Lemma 2. The proof proceeds by structural induction on $\overline{\mathfrak{o}(a,v)}$.

Base case: $\overline{\mathfrak{o}(a,v)} = \epsilon$ As no rewrite rules apply here, this case is terminating.

Inductive case: $\overline{\mathfrak{o}(a,v)} = \mathfrak{o}_0(a_0,v_0)\cdots\mathfrak{o}_n(a_n,v_n)$ The inductive hypothesis IH tells us that applying the rewriting rules to a prefix $\mathfrak{o}_1(a_1,v_1)\cdots\mathfrak{o}_n(a_n,v_n)$ of length n is terminating in q steps. Apply Lemma 1 to know that the confluent form of $\mathfrak{o}_1(a_1,v_1)\cdots\mathfrak{o}_n(a_n,v_n)$ is $\mathfrak{o}'_1(a'_1,v'_1)\cdots\mathfrak{o}'_n(a'_n,v'_n)$. The following alternatives arise based on $\mathfrak{o}_0(a_0,v_0)$ and $\mathfrak{o}'_1(a'_1,v'_1)$.

$a_0 < a'_1$. In this case no rewrite rules apply for prefixes of index 0 and 1. Their application terminates in q steps for prefixes 1 onwards as stated in the IH. So this case terminates in q steps.

$a_0 > a'_1$. In this case, $\mathfrak{o}_0$ and $\mathfrak{o}'_1$ are swapped in place by applying one rewriting rule. We can then apply the IH to state that applying the rewrite rules to $\mathfrak{o}_0(a_0,v_0)\mathfrak{o}'_2(a'_2,v'_2)\cdots\mathfrak{o}'_n(a'_n,v'_n)$ is terminating. By applying Lemma 1, we can define the confluence form of that prefix with $\mathfrak{p}$. $\mathfrak{o}_1(a_1,v_1)\mathfrak{p}$ is thus terminating since no rewrite rules apply to prefixes of index 0 and 1 while their application terminates in q steps for prefixes 1 onwards as stated in the IH. So this case terminates in $q+1$ steps.

$a_0 = a'_1 = a$. The following cases arise:

$\mathfrak{o}_0(a,v_0) = \mathtt{read}(a,v_0)$ and $\mathfrak{o}'_1(a,v'_1) = \mathtt{read}(a,v'_1)$. Firstly, Equation (Constraint-read) is applied, thus $v_0 = v'_1$. Then, Equation (Read-drop) drops one of those actions, so we can apply the IH since the prefix is of length n. So this case terminates in $q+2$ steps.

$\mathfrak{o}_0(a,v_0) = \mathtt{read}(a,v_0)$ and $\mathfrak{o}'_1(a,v'_1) = \mathtt{write}(a,v'_1)$. The following cases arise:

$v_0 = v'_1 = v$. Equation (Read-no-write) is applied, and only the 0-indexed action is kept. We can apply the IH to the prefix $\mathfrak{o}_0(a_0,v_0)\mathfrak{o}'_2(a'_2,v'_2)\cdots\mathfrak{o}'_n(a'_n,v'_n)$ which is of length n. So this case terminates in $q+1$ steps.

$v_0 \neq v'_1$. In this case no rewrite rules apply for prefixes of index 0 and 1. Their application terminates for prefixes 1 onwards as stated in the IH. So this case terminates in q steps.

$\mathfrak{o}_0(a, v_0) = \texttt{write}(a, v_0)$ **and** $\mathfrak{o}'_1(a, v'_1) = \texttt{read}(a, v'_1)$**.** Firstly, Equation (Constraint-write) is applied, thus $v_0 = v'_1$.
Then, Equation (Write-drop) drops the 1-indexed action, so we can apply the IH to the prefix $\mathfrak{o}_0(a_0, v_0)\mathfrak{o}'_2(a'_2, v'_2) \cdots \mathfrak{o}'_n(a'_n, v'_n)$ since it is of length n. So this case terminates in $q + 2$ steps.

$\mathfrak{o}_0(a, v_0) = \texttt{write}(a, v_0)$ **and** $\mathfrak{o}'_1(a, v'_1) = \texttt{write}(a, v'_1)$**.** Apply Equation (Write-drop) so that only the 1-indexed action is kept, so we can apply the IH and this case holds. So this case terminates in $q + 1$ steps.

Since all cases terminate in a finite number of steps, applying the rewrite rules always terminates. □

Theorem 1 (Convergence) *The rewrite rules are convergent, i.e., they are confluent and terminating.*

Proof of Theorem 1. By Lemma 1 and Lemma 2. □

5.4 Proof of Full Abstraction for the Trace Semantics

This section presents the proof through which both $\mathsf{Traces}^{\mathsf{S}}_{\mathsf{A+I}}$ and $\mathsf{Traces}^{\mathsf{L}}_{\mathsf{A+I}}$ are proven to be fully abstract w.r.t. the corresponding operational semantics.

A fully abstract trace semantics is both sound and complete with respect to the operational semantics. Soundness means that the trace semantics captures all behaviours expressible with the operational semantics. Thus, for all contexts, two trace equivalent programs cannot be told apart. Completeness means that the trace semantics does not capture additional behaviours that are not expressible with the operational semantics. Thus, for all trace-inequivalent programs, there exists a context that can differentiate them.

Full abstraction of trace semantics is formally stated as: $P_1 \stackrel{\mathrm{T}}{=}^{\mathsf{A+I}} P_2 \iff P_1 \simeq^{\mathsf{A+I}} P_2$; its proof is split in two cases, one for each direction of the co-implication (Theorem 2 and Theorem 3, respectively).

Call the *interface* of a state its registers, flags and unprotected memory. Two states Ω_1 and Ω_2 have the same interface, denoted as $\Omega_1 \doteq \Omega_2$, if they have the same registers, flags and unprotected memory. Formally, $\Omega_1 \doteq \Omega_2$ if $\Omega_1 = (p_1, r, f, m_1, s_1)$ and $\Omega_2 = (p_2, r, f, m_2, s_2)$ and $\mathtt{m_{ext}}(m_1, s_1) = \mathtt{m_{ext}}(m_2, s_2)$.

Given $\Omega = (p, r, f, m, s)$, define $\lfloor \Omega \rfloor$ to be the state $\Theta = (p, r, f, \mathtt{m_{sec}}(m, s), s)$ if $s \vdash \mathtt{protected}(p)$ and $(\mathtt{unk}, m, s)$ otherwise.

To prove Theorem 2, both programs must be proven to offer the same interface to the unprotected program. This is captured by an interface-preservation lemma (Lemma 3) which must be proven for each trace semantics since it depends on the labels of each trace semantics. Lemma 3 says that two states with the same interface still have the same interface after they perform the same observable action. Thus unprotected programs do not see differences, in terms of flags, registers and unprotected memory, between P_1 and P_2.

Lemma 3 (Interface preservation after same observable action) *If* $\Theta_1 \overset{\overline{\alpha}}{\Longrightarrow\!\!\!\!\!\!\gg} \xrightarrow{\alpha}\!\!\!\!\!\!\twoheadrightarrow \Theta_1'$ *and* $\Theta_1 = \lfloor \Omega_1 \rfloor$ *and* $\Omega_1 \twoheadrightarrow^* \Omega_1'$ *and* $\Theta_1' = \lfloor \Omega_1' \rfloor$ *and* $\Theta_2 \overset{\overline{\alpha}}{\Longrightarrow\!\!\!\!\!\!\gg} \xrightarrow{\alpha}\!\!\!\!\!\!\twoheadrightarrow$ Θ_2' *and* $\Theta_2 = \lfloor \Omega_2 \rfloor$ *and* $\Omega_2 \twoheadrightarrow^* \Omega_2'$ *and* $\Theta_2' = \lfloor \Omega_2' \rfloor$ *and* $\Omega_1 \mathrel{\mathring{=}} \Omega_2$ *then* $\Omega_1' \mathrel{\mathring{=}} \Omega_2'$ *(assuming there is no overflow of the secure stack).*

We overload the hole-filling notation and allow a hole to be filled by a state $\Omega = (p, r, f, m, s)$ as follows: $\mathbb{M}[\Omega] = (p, r, f, m + m', s)$, if $(m, s) \frown \mathbb{M}$. Given an instruction $i \in \mathcal{I}$, identify a transition triggered by the execution of that instruction as $\xrightarrow{i}$.

Proof of Lemma 3. By Definition 36 the thesis $P_1 \simeq^{\mathsf{A+I}} P_2$ becomes $\forall \mathbb{M}.\ \mathbb{M}[P_1]\Uparrow \iff \mathbb{M}[P_2]\Uparrow$.

The proof is split in two cases, one for each side of the co-implication.

1. Direction $\Rightarrow$, so the thesis is $\forall \mathbb{M}.\ \mathbb{M}[P_1]\Uparrow \Rightarrow \mathbb{M}[P_2]\Uparrow$.

   Apply the definition of contextual equivalence (Definition 36) and the thesis becomes $\forall \mathbb{M}.\ \mathbb{M}[\Omega_0(P_1)]\Uparrow \Rightarrow \mathbb{M}[\Omega_0(P_2)]\Uparrow$.

   Let $\Omega_1 = \mathbb{M}[\Omega_0(P_1)]$ and $\Omega_2 = \mathbb{M}[\Omega_0(P_2)]$.

   The thesis is $\forall \mathbb{M}.\ \forall n \in \mathbb{N}. \exists \Omega_1'.\ \Omega_1 \twoheadrightarrow^n \Omega_1' \Rightarrow \forall m \in \mathbb{N}. \exists \Omega_2'.\ \Omega_2 \twoheadrightarrow^m \Omega_2'$.

   The proof proceeds by induction on m.

   Base case: $m = 0$**.** Straightforward: $\Omega_2 \twoheadrightarrow^0 \Omega_2$.

   Inductive case: $m = h + 1$**.** The thesis is: $\exists \Omega_2'. \Omega_2 \twoheadrightarrow^{h+1} \Omega_2'$.

   The inductive hypothesis (IH) is: $\forall \mathbb{M}.\ \forall n \in \mathbb{N}. \exists \Omega_1'.\ \mathbb{M}[\Omega_0(P_1)] \twoheadrightarrow^n \Omega_1' \Rightarrow \mathbb{M}[\Omega_0(P_2)] \twoheadrightarrow^h \Omega_2^h$.

   By IH we have that: $\exists \Omega_1.\ \mathbb{M}[\Omega_0(P_1)] \twoheadrightarrow^h \Omega_1^h \twoheadrightarrow^{n-h} \Omega_1'$.

   Let $\Omega_1^h = (p_1, \ldots)$ and $\Omega_2^h = (p_2, \ldots)$.

   There are two cases based on p_1 and p_2: both p_1 and p_2 are in the protected partition (Item 1a) or in the unprotected one (Item 1b).

(a) $s_1 \vdash \mathsf{protected}(p_1)$ and $s_2 \vdash \mathsf{protected}(p_2)$.
This case relies on the trace semantics rules to say that either P_1 and P_2 produce the same label, or they diverge; in both cases there is a corresponding reduction in the operational semantics. There are two cases: either both programs will perform another action $\mathfrak{d}!$ (Item 1(a)i), or not (Item 1(a)ii).

i. $\exists \mathfrak{d}!.\ \lfloor \Omega_1^h \rfloor \xRightarrow{\mathfrak{d}!}\!\!\!\!\to \lfloor \Omega_1^{h'} \rfloor$.
By hypothesis $P_1 \overset{\mathrm{T}}{=} P_2$, $\lfloor \Omega_2^h \rfloor \xRightarrow{\mathfrak{d}!}\!\!\!\!\to \lfloor \Omega_2^{h'} \rfloor$.
This, in conjunction with IH, implies the thesis $\Omega_2 \twoheadrightarrow^{h+1} \Omega_2'$.
Note that $\mathfrak{d}!$ cannot be a $\surd$, as this violates the hypothesis $\forall n \in \mathbb{N}.\mathbb{M}[\Omega_0](P_1)] \rightarrow^n \Omega_1'$.

ii. $\nexists \mathfrak{d}!.\ \lfloor \Omega_1^h \rfloor \xRightarrow{\mathfrak{d}!}\!\!\!\!\to \lfloor \Omega_1^{h'} \rfloor$.
Let $\Omega_1^h = (p_1, r_1, f_1, m + m_1, s_1)$ and $\Omega_2^h = (p_2, r_2, f_2, m + m_2, s_2)$.
In this case, Ω_2^h does not terminate, since it does not produce a $\surd$ label, so it computes, generating τ actions.
By inspecting rules for generating τ in traces (the only possible rule that applies in this case), we have that $m_1(p_1) = i_1 \in \mathcal{I}$ and $m_2(p_2) = i_2 \in \mathcal{I}$.
The thesis holds because Ω_2 can always make a step for instruction i_2, so $\Omega_2 \twoheadrightarrow^h \Omega_2^h \xrightarrow{i_2} \Omega_2'$.

(b) $s_1 \vdash \mathsf{unprotected}(p_1)$ and $s_2 \vdash \mathsf{unprotected}(p_2)$.
In this case we need to prove that, for whatever computation was done so far, P_1 and P_2 end up with a program counter in the same location in their unprotected memory. We rely on Lemma 3 to state that, if P_1 and P_2 have jumped inside the protected partition and then back outside, their unprotected memory is still the same.
By IH $\exists l \leq h.\ \mathbb{M}[\Omega_0(P_1)] \twoheadrightarrow^l \Omega_1^l$ and $\lfloor \Omega_0(P_1) \rfloor \xRightarrow{\overline{\mathfrak{a}}\mathfrak{a}}\!\!\!\!\to \lfloor \Omega_1^l \rfloor$.
By hypothesis $P_1 \overset{\mathrm{T}}{=} P_2$, $\exists l \leq h.\ \mathbb{M}[\Omega_0(P_2)] \twoheadrightarrow^l \Omega_2^l$.
Additionally, $\lfloor \Omega_0(P_2) \rfloor \xRightarrow{\overline{\mathfrak{a}}\mathfrak{a}}\!\!\!\!\to \lfloor \Omega_2^l \rfloor$.
By Lemma 3, $\Omega_1^l = (p^l, r^l, f^l, \mathbb{M}^l + m_1, s_1)$ and $\Omega_2^l = (p^l, r^l, f^l, \mathbb{M}^l + m_2, s_2)$ (if $\mathfrak{a}$ does not exist and $\overline{\mathfrak{a}}$ is the empty list, there is no need to apply Lemma 3).
Additionally, $\Omega_1^l \twoheadrightarrow^{h-l} \Omega_1^h$ and $\Omega_2^l \twoheadrightarrow^{h-l} \Omega_2^h$.
Since $\mathbb{M}^l$ is the same for both P_1 and P_2, the $(h - l)$-steps they perform in unprotected memory are the same for Ω_1^l and Ω_2^l.
Thus $\Omega_1^h = (p^h, r^h, f^h, \mathbb{M}^h + m_1, s_1)$ and $\Omega_2^h = (p^h, r^h, f^h, \mathbb{M}^h + m_2, s_2)$.
As stated in Section 4.2 $p \in \mathtt{dom}(\mathbb{M}^h)$ implies that $s_1 \vdash \mathsf{unprotected}(p^h)$ and $s_2 \vdash \mathsf{unprotected}(p^h)$.

By hypothesis $\forall n \in \mathbb{N}.\ \mathbb{M}[\Omega_0(P_1)] \twoheadrightarrow^n \Omega'_1$: we have that $\Omega_1^h \xrightarrow{i} \Omega'_1$ and that $\mathbb{M}^h(p^h) \cong i \in \mathcal{I}$.

This implies the thesis: $\Omega_2 \twoheadrightarrow^{h+1} \Omega'_2$ since $\Omega_2 \twoheadrightarrow^h \Omega_2^h \xrightarrow{i} \Omega'_2$.

2. $\Leftarrow$ As in case Item 1, *mutatis mutandis.*

$\square$

Theorem 2 (Soundness) $P_1 \overset{\mathcal{T}^{\mathsf{A+I}}}{=} P_2 \ \Rightarrow\ P_1 \simeq^{\mathsf{A+I}} P_2$ *(assuming there is no overflow of the secure stack).*

Proof of Theorem 2. The proof proceeds by induction on $\overline{\mathfrak{a}}$ that leads to a case analysis on $\mathfrak{a}$. We omit the inductive cases and proceed directly to the case analysis considered for the base case.

$\surd$. Straightforward: the thesis is $\Omega_1 \mathrel{\mathring{=}} \Omega_2$, which is among the hypotheses.

?-decorated action. This action can either be a call of the form $\texttt{call}p(r;f)$ or a return of the form $\texttt{ret}p(r;f)$, only the case for the call is presented since the one for the return is analogous.

So: $\Theta_1 \xrightarrow{\texttt{call}\ p(r;f)?}\!\!\!\twoheadrightarrow \Theta'_1$ and $\Theta_2 \xrightarrow{\texttt{call}\ p(r;f)?}\!\!\!\twoheadrightarrow \Theta'_2$.

By definition, $\Theta'_1 = \Omega'_1 = (p, r, f, m'_1, s_1)$ and $\Theta'_2 = \Omega'_2 = (p, r, f, m'_2, s_2)$.

The thesis is $\Omega'_1 \mathrel{\mathring{=}} \Omega'_2$, so $(p, r, f, m'_1, s_1) \mathrel{\mathring{=}} (p, r, f, m'_2, s_2)$. Both states need to have equal registers, flags and unprotected memory. The first two points are clear, as registers and flags are set to be the same by the label. What needs to be proven is that $\mathtt{m}_{\mathtt{ext}}(m'_1, s_1) = \mathtt{m}_{\mathtt{ext}}(m'_2, s_2)$.

From hypothesis $\Omega_1 \mathrel{\mathring{=}} \Omega_2$, we have that $\mathtt{m}_{\mathtt{ext}}(m_1, s_1) = \mathtt{m}_{\mathtt{ext}}(m_2, s_2)$.

Since the action $\texttt{call}\ p(r;f)?$ does not touch the unprotected memory, we have that $\mathtt{m}_{\mathtt{ext}}(m_1, s_1) = \mathtt{m}_{\mathtt{ext}}(m'_1, s_1)$ and $\mathtt{m}_{\mathtt{ext}}(m_2, s_2) = \mathtt{m}_{\mathtt{ext}}(m'_2, s_2)$.

By transitivity we obtain that $\mathtt{m}_{\mathtt{ext}}(m'_1, s_1) = \mathtt{m}_{\mathtt{ext}}(m'_2, s_2)$ holds, so this case holds as well.

!-decorated action. Here, $\delta!$ is in the form $\delta_1 \cdots \delta_n \mathfrak{g}!$. The action $\mathfrak{g}!$ can either be a call of the form $\texttt{call}p(r;f)$ or a return of the form $\texttt{ret}p(r;f)$; only the case for the call is presented since the one for the return is analogous.

So: $\Theta_1 \xrightarrow{\delta_1 \cdots \delta_n \texttt{call}\ p(r;f)!}\!\!\!\twoheadrightarrow \Theta'_1$ and $\Theta_2 \xrightarrow{\delta_1 \cdots \delta_n \texttt{call}\ p(r;f)!}\!\!\!\twoheadrightarrow \Theta'_2$.

$\Theta_1 = \Omega_1 = (p, r, f, m_1, s_1)$ and $\Theta_2 = \Omega_2 = (p, r, f, m_2, s_2)$.

By definition, $\Theta'_1 = (\mathsf{unk}, m'_1, s_1)$ and $\Theta'_2 = (\mathsf{unk}, m'_2, s_2)$.

We can reconstruct Ω'_1 by applying the following hypotheses: $\Theta_1 = \lfloor \Omega_1 \rfloor$ and $\Omega_1 \twoheadrightarrow^* \Omega'_1$ and $\Theta'_1 = \lfloor \Omega'_1 \rfloor$. Analogously, we can reconstruct Ω'_2.

So, $\Omega'_1 = (p, r, f, m'_1, s_1)$ and $\Omega'_2 = (p, r, f, m'_2, s_2)$. The thesis is $\Omega'_1 \mathrel{\mathring{=}} \Omega'_2$, so $(p, r, f, m'_1, s_1) \mathrel{\mathring{=}} (p, r, f, m'_2, s_2)$. Both states need to have equal registers, flags and unprotected memory. The first two points are clear, as registers and flags are set to be the same by the label. What needs to be proven is that $\mathtt{m_{ext}}(m'_1, s_1) = \mathtt{m_{ext}}(m'_2, s_2)$.

From hypothesis $\Omega_1 \mathrel{\mathring{=}} \Omega_2$, we have that $\mathtt{m_{ext}}(m_1, s_1) = \mathtt{m_{ext}}(m_2, s_2)$.

What needs to be considered are the prefixes $\delta_1 \cdots \delta_n$, which can be either readouts or writeouts: The proof now proceeds by induction on n.

Base case, $n = 0$ Trivial, since we have that $\mathtt{m_{ext}}(m_1, s_1) = \mathtt{m_{ext}}(m'_1, s_1)$ and $\mathtt{m_{ext}}(m_2, s_2) = \mathtt{m_{ext}}(m'_2, s_2)$.

This, combined with the hypothesis $\mathtt{m_{ext}}(m_1, s_1) = \mathtt{m_{ext}}(m_2, s_2)$, fulfils this case.

Inductive case, $n = k + 1$ Consider $\delta_1 \cdots \delta_k \delta'$, the inductive hypothesis states that up to δ_k, external memories are the same. Indicate the memory up to the kth step with m_k, the inductive hypothesis states that $\mathtt{m_{ext}}(m^k_1, s_1) = \mathtt{m_{ext}}(m^k_2, s_2)$.

Two cases arise for δ', one for the readout and one for the writeout.

$\delta' = \mathtt{read}(a, v)$ Readouts do not change the external memory, so apply the inductive hypothesis and this case holds.

$\delta' = \mathtt{write}(a, v)$ Writeouts *do* change the external memory, so $\mathtt{m_{ext}}(m'_1, s_1) = \mathtt{m_{ext}}(m^k_1, s_1)[a \mapsto v]$ and $\mathtt{m_{ext}}(m'_2, s_2) = \mathtt{m_{ext}}(m^k_2, s_2)[a \mapsto v]$.
Since the initially-equal memories $\mathtt{m_{ext}}(m^k_1, s_1)$ and $\mathtt{m_{ext}}(m^k_2, s_2)$ are changed in the same way, the thesis holds in this case as well.

Having covered all the cases, the theorem holds. □

The proof of Lemma 3 for $\mathsf{Traces}^{\mathsf{S}}_{\mathsf{A+I}}$ is included in the proof for $\mathsf{Traces}^{\mathsf{L}}_{\mathsf{A+I}}$ with very small syntactical changes since the labels of $\mathsf{Traces}^{\mathsf{S}}_{\mathsf{A+I}}$ are a subset of the labels of $\mathsf{Traces}^{\mathsf{L}}_{\mathsf{A+I}}$.

Theorem 3 (Completeness) $P_1 \simeq^{\mathsf{A+I}} P_2 \Rightarrow P_1 \overset{T}{=}^{\mathsf{A+I}} P_2$ *(assuming there is no overflow of the secure stack).*

Completeness is equivalently stated as: $P_1 \overset{T}{\neq}{}^{\mathsf{A+I}} P_2 \Rightarrow P_1 \not\simeq^{\mathsf{J+E}} P_2$.

Proof Sketch. This is proven by devising an algorithm that takes as input two different traces $\overline{\alpha_1}$ and $\overline{\alpha_2}$ and the two programs P_1 and P_2 generating them and outputs a program P that interacts with P_1 and P_2 and is able to differentiate between them [10, 64, 94, 97]. The algorithm produces unprotected code that performs all ?-decorated actions in the traces and then terminates with result 1 or diverges, based on the program it is interacting with after the different !-decorated action.

The two different traces are generated as follows. Since $P_1 \not\overset{T}{=}{}^{\mathsf{A+I}} P_2$, we have that $\mathsf{Traces}(P_1) \neq \mathsf{Traces}(P_2)$, thus there exists a trace $\overline{\alpha}$ that belongs to either only $\mathsf{Traces}(P_1)$ or only $\mathsf{Traces}(P_2)$. Assume wlog that $\overline{\alpha} \in \mathsf{Traces}(P_1)$. The trace $\overline{\alpha}$ can be split in two parts $\overline{\alpha_s}$ (the common prefix) and $\overline{\alpha_d}$ such that $\overline{\alpha}{=}\overline{\alpha_s \alpha_d}$, and so that there exists a trace $\overline{\alpha'} \in \mathsf{Traces}(P_2)$ that can be split in two parts $\overline{\alpha_s}$ and $\overline{\alpha'_d}$ such that $\overline{\alpha'}{=}\overline{\alpha_s}\overline{\alpha'_d}$ and $\overline{\alpha_d} \neq \overline{\alpha'_d}$. Trace $\overline{\alpha'}$ always exists, it could be an empty trace, it could be composed by an empty $\overline{\alpha_s}$ and, possibly, by an empty $\overline{\alpha'_d}$. The traces input for the algorithm are $\overline{\alpha_1} = \overline{\alpha_s \alpha_d}$ and $\overline{\alpha_2} = \overline{\alpha_s}\overline{\alpha'_d}$.

Algorithm description Assume that there is always enough memory to store the algorithm; call the algorithm P. In P there must be four functions in order to set the flags to the all combinations. These function are of the form:

- store $\mathtt{r}_1$ and $\mathtt{r}_2$ in unprotected memory;
- set $\mathtt{r}_1$ and $\mathtt{r}_2$ to the right values that set the flag to the desired combination (e.g., for $\mathsf{SF}{=}0$; $\mathsf{ZF}{=}1$, set $\mathtt{r}_1{=}1$ and $\mathtt{r}_2{=}1$);
- execute $\mathtt{cmp}\ \mathtt{r}_1\ \mathtt{r}_2$;
- restore $\mathtt{r}_1$ and $\mathtt{r}_2$ to the corresponding previous values;
- `ret`.

The algorithm keeps track of where to write instructions in P with a stack: the *current address stack c*. Initially, the top of stack c is set to p_0 – the initial value of the program counter.

The algorithm scans the traces $\overline{\alpha_1}$ and $\overline{\alpha_2}$. By construction, each even-numbered label is !-decorated; each odd-numbered label is ?-decorated. The algorithm is split in two subroutines based on what kind of actions it is examining. Each subroutine analyses one action from each trace and then calls the other subroutine on the following actions until the differentiation is achieved; in that case the algorithm terminates.

?-decorated actions. These actions are generated by the unprotected code. The algorithm must output a P that generates those traces. Thus, at location c, the algorithm writes code depending on what action is being considered.

`call` $p\ (r, f)?$ Firstly, the algorithm writes a `call` to the function that sets the flags to f. Then the top of stack c is incremented by 1. Then, all twelve registers are set to the values of r, thus given that the values of register i in r is v_i, the following instruction is written: `movi` $\mathtt{r_i}$ v_i for all $i = 0..11$. If the value to be written in a register is larger than the constant allowed by `movi`, an `add` instruction is used. Then the top of c is incremented by 12 (or more, if `add` instructions are used). Then based on which register contains the value p that is where the call is directed, instruction `call` $\mathtt{r_p}$ is written. Then the top of c is incremented by 1.

`ret` $p\ (r, f)!$ As in the previous step, the algorithm sets flags and registers to the desired values. Then instruction `ret` is written. Then the top of c is incremented by 1.

!-decorated actions. These actions are generated by protected code.

callbacks. If both actions are of the form `call` $p\ (r, f)?$, then p is pushed on top of the stack c.

returns. If both actions are of the form `ret` $p\ (r, f)!$, then the top of the stack c is popped.

writeouts. The algorithm adds no code to P. In this case we are assured that control will jump back to the code because protected code does not write in the code section of unprotected code.

readouts. If both actions are of the form `read`(a, v), then the following instructions are written before other code at address c: `movi` $\mathtt{r_0}$ a; `movi` $\mathtt{r_1}$ v; `movs` $\mathtt{r_1}$ $\mathtt{r_0}$. These instructions ensure that address a contains value v.

To avoid clashes with writeouts and readouts, assume traces are inspected beforehand and the location where P is stored does not clash with the addresses of those operations.

If the labels are different, then the algorithm writes the differentiating code at address c in P. Differences in the labels can be of these types:

different length. Thus one label is $\surd$ and the other one is $\mathfrak{a} \neq \surd$. In this case, given that $\mathfrak{a}$ is generated by program P_i, the algorithm writes diverging code at the address indicated by c.

different actions. Assume that $\mathfrak{a}_1 = \texttt{ret}\ p\ (r, f)?$ and $\mathfrak{a}_2 = \texttt{call}\ 10\ (r, f)!$. Then the algorithm writes instructions $\texttt{movi r}_0\ \texttt{1}$; $\texttt{halt}$ at c and diverging code at address 10.

Assume that $\mathfrak{a}_1 = \texttt{write}(a, v).\mathfrak{d}!$ and $\mathfrak{a}_2 = \mathfrak{d}!$. In this case, before executing the protected code that generates that trace, the algorithm writes value u, different from v, at address a. Then, after the protected code has performed $\mathfrak{d}!$, the value in a is read and compared to u. If they are the same, then instructions $\texttt{movi r}_0\ \texttt{1}$; $\texttt{halt}$ are written at c, otherwise diverging code is written there.

Other cases are similar.

different values in the same action. Assume that $\mathfrak{a}_1 = \texttt{ret}\ p\ (r; 0, 1)?$ and $\mathfrak{a}_2 = \texttt{ret}\ p\ (r; 0, 0)?$. Then the differentiating code is the following: perform a jump (via $\texttt{jl}$ in this case since flag SF bears a different value in the two traces) to an address a in case the flag is 1. At address a, instructions $\texttt{movi r}_0\ \texttt{1}$; $\texttt{halt}$ are written. Right after the jump, diverging code written.

Assume that $\mathfrak{a}_1 = \texttt{ret}\ p\ (1, \ldots; f)?$ and $\mathfrak{a}_2 = \texttt{ret}\ p\ (2, \ldots; f)?$. Then the differentiating code is the following: $\texttt{movi r}_1\ 1$; $\texttt{sub r}_0\ \texttt{r}_1$. Now the problem is reduced to different values in flags, so the previous approach can be used.

Assume that $\mathfrak{a}_1 = \texttt{call}\ 10\ (r; f)!$ and $\mathfrak{a}_2 = \texttt{call}\ 20\ (r; f)!$. Then the algorithm writes instructions $\texttt{movi r}_0\ \texttt{1}$; $\texttt{halt}$ at address 10 and diverging code at address 20.

Assume that $\mathfrak{a}_1 = \texttt{write}(a_1, v_1).\mathfrak{d}!$ and $\mathfrak{a}_2 = \texttt{write}(a_2, v_2).\mathfrak{d}!$. The same procedure stated in the last paragraph for the previous point is applied.

Concerning readouts, they are included in the traces only if they are followed by different actions. Readouts that are not followed by different actions satisfy the non-interference judgment $\mathsf{NI}(\cdot)$, they are non-interfering. On the other hand, readouts that are followed by different actions do not satisfy that judgment, they are interfering. Function $\mathsf{StripNI}(\cdot)$ (Figure 5.7), which is used to accumulate labels in Rule Trace-l-action, ensures that all non-interfering readout labels are eliminated from traces. So, readouts that appear in traces are interfering and thus followed by different actions. It is that action that determines what the code generated by the algorithm is, no action is undertaken for readouts.

□

Theorem 4 (Fully abstract trace semantics for A+I) $P_1 \simeq^{\mathsf{A+I}} P_2 \iff P_1 \overset{T}{=}{}^{\mathsf{A+I}} P_2$ *(assuming there is no overflow of the secure stack).*

Proof of Theorem 4. Apply Theorem 2 and Theorem 3. □

These proofs are presented for $\mathsf{Traces}^{\mathsf{L}}_{\mathsf{A+I}}$, but they scale easily to $\mathsf{Traces}^{\mathsf{S}}_{\mathsf{A+I}}$. In fact, the proofs perform a case analysis on the labels and the labels of the $\mathsf{Traces}^{\mathsf{S}}_{\mathsf{A+I}}$ semantics are a subset of those of the $\mathsf{Traces}^{\mathsf{L}}_{\mathsf{A+I}}$ one. The proof of Lemma 3 induce on the operational semantics related to a trace semantics, but the changes to the operational semantics considered for $\mathsf{Traces}^{\mathsf{S}}_{\mathsf{A+I}}$ are minimal w.r.t. the one of $\mathsf{Traces}^{\mathsf{L}}_{\mathsf{A+I}}$.

Chapter 6

A Secure Compiler from J+E to A+I

The essence of a role-playing game is that it is a group, cooperative experience.

Gary Gygax

This chapter presents the secure compiler from J+E to A+I. Firstly, this chapter describes security violations that can arise due to a naïve compiler implementation (Section 6.1). Then it presents the secure compiler and how to securely compile language features such as outcalls, dynamic memory allocation and exceptions (Section 6.2). Finally, it discusses how to securely compile additional language features that have not been incorporated in J+E (Section 6.3).

6.1 Security Violations

This section presents a series of security violations that arise due to a naïve compiler implementation (Problems 2 to 8) that follows standard conventions about how objects are compiled [38]. These problems are each presented as two source-level snippets that are equivalent at the source level but are inequivalent to the target level. The compilation scheme used to compile them is thus not fully abstract. This full abstraction failure manifests itself in the proof in the form of two different target-level traces for which it is not possible to devise a distinguishing source-level component (the witness mentioned in Section 2.7). These cases (often) lead to security violations. Consider the code of Figure 2.3. The behaviour of the two code snippets can be represented with the following traces even when a supposedly compiled version of those snippets is considered:

$$\begin{aligned}\text{First case} &= \texttt{call}\ o.\texttt{proxy}?\ \texttt{call}\ cb.\texttt{callback}!\ \texttt{ret}\ 0?\ \texttt{ret}\ 1!\\ \text{Second case} &= \texttt{call}\ o.\texttt{proxy}?\ \texttt{call}\ cb.\texttt{callback}!\ \texttt{ret}\ 0?\ \texttt{ret}\ 0!\end{aligned}$$

The final return of the first trace is 1 because the local variable `secret` has been tampers with (this tampering is not captured by the traces). Generating a source-level component from these traces is not possible, since at the source level, unprotected code cannot access the local variable `secret`. For the compilation scheme to be secure, securely-compiled programs must not generate this kind of traces.

The failure of full abstraction for each problem is presented by the target-level trace that differentiates the two snippets at the target level, and for which a source-level witness cannot be created. As for Problem 1, the two snippets provide different implementations for the same classes or methods. Recall that these implementations often interact with an external object of type `External` which presents a method `callback()`. For example, each snippet implements a class that is implemented by an object `o`, which is called $\texttt{o}_L$ in the left-hand side snippet and $\texttt{o}_R$ in the right-hand side one.

Problem 2 (Stack security) *Consider two classes that define a secret field called* `secret` *with different values and the same method* `doCallback` *that calls method* `callback` *on object* `cb`*.*

```
package pL;
class CL {
  private secret : Int = 0;

  public doCallback( ) : Int {
    var x : Int = secret;
    cb.callback( );
    if ( x == secret )
      return 0;
    return 1;
  }
}
object oL : CL
```

```
package pR;
class CR {
  private secret : Int = 1;

  public doCallback( ) : Int {
    var x : Int = secret;
    cb.callback( );
    if ( x == secret )
      return 0;
    return 1;
  }
}
object oR : CR
```

Objects $\texttt{o}_L$ *and* $\texttt{o}_R$ *are equivalent at the source level, but their compiled counterparts are not. Since local variables are placed on the call stack (in unprotected memory) and an* A+I*-level attacker can read unprotected memory,* *she can read and write the value of* `x` *during the callback* `cb.callback()`*. Variable* `x` *contains the value of* `secret`*, which is different for both objects. The following trace is generated by the right-hand side snippet but not by the left-hand side one. No source-level component can be created to replicate the behaviour of this trace since local variables are not accessible in* J+E*.*

$$\texttt{call}\ o.\texttt{doCallback()}?\ \texttt{call}\ cb.\texttt{callback()}!\ \texttt{ret}\ 0?\ \texttt{ret}\ 1!$$

This full abstraction violation happens because there is no J+E *program that can read the stack contents, so the violation is not replicable at the source level. An attacker can use this vulnerability to read secrets from the stack, similarly to a buffer-overread attack [114]. Alternatively she can even tamper with the control flow by overwriting a return address on the stack, similarly to a return address clobbering attack [41].* ■

Problem 3 (Information leakage) *Consider two classes that define the same method* `testVariable` *which assigns different values to a local variable* `x`*. Both methods tests whether that value is 0 and the left-hand side one returns 0 if it is and 1 otherwise while the right-hand side ones returns 1 if it is and 0 otherwise.*

```
package pL;
class CL {
  public testVariable( ) : Int {
    var x : Int = 0;
    if ( x == 0 ) {
      return 0;
    } else {
      return 1;
    }
  }
}
object oL : CL
```

```
package pR;
class CR {
  public testVariable( ) : Int {
    var x : Int = 1;
    if ( x == 0 ) {
      return 1;
    } else {
      return 0;
    }
  }
}
object oR : CR
```

Objects $\mathtt{o}_L$ *and* $\mathtt{o}_R$ *are equivalent in* J+E*, as method* `testVariable` *always returns* 0 *for both. However, an* A+I*-level attacker can differentiate their translations, due to the equality test in the condition of the if-statement. This test sets the* ZF *flag in* $\mathtt{o}_L$ *and clears it in* $\mathtt{o}_R$*. This example illustrates that the flags register can leak information. Information can also be leaked through the general purpose registers* $\mathtt{r}_0$ *to* $\mathtt{r}_{11}$ *or through the stack pointer register* SP*. This full abstraction violation happens because flags are not observable in* J+E*. Whether a conditional statement in some function evaluated its "then" or "else" branch is not discernible by external code that called that function. No source-level component can be created to replicate the behaviour of this trace since flags are* *not accessible in* J+E.

$$\texttt{call } o.\texttt{testVariable()}?\ \texttt{ret } 1!$$

■

Problem 4 (Value of booleans) *Consider two classes that provide a method* `identBool` *which inputs a* **`Bool`***ean value and returns* `true` *if that value is* `true` *or* `false` *otherwise.*

```
package pL;
class CL {
  public identBool( x : Bool ) :
      Bool {
    if( x == true )
      return true;
    return false;
  }
}
object oL : CL
```

```
package pR;
class CR {
  public identBool( x : Bool ) :
      Bool {

    return x;

  }
}
object oR : CR
```

The two classes implement this method differently, yet the behaviour is the same, so objects $\texttt{o}_L$ *and* $\texttt{o}_R$ *are equivalent in* J+E*. Their* A+I *translations however are not, because an* A+I*-level attacker can use any* A+I*-level value for parameter* `x`*. The following trace is generated by the right-hand side snippet but not by the left-hand side one. No source-level component can be created to replicate the behaviour of this trace since only boolean values can be passed as arguments of* `identBool` *in* J+E*.*

$$\texttt{call } o.\texttt{identBool(7)}?\ \texttt{ret } 7!$$

This problem arises whenever the source language is high-level and it has primitive types inhabited by a reduced number of values, such as **`Bool`***eans and* **`Unit`** *[10, 45]. This is similar to a full abstraction failure for the .NET C# compiler reported by Kennedy [67], where the boolean type is two valued in C# but is byte valued in the .NET virtual machine.* ■

Problem 5 (Type of the receiver) *Consider two packages that provide a class with an unaccessible secret and a class that implements* `Pair`*s of* `Obj`*ects with a method to get the first element of the pair.*

```
package pL;
class PairL {
  private fst, snd : Obj = null;
  public getFirst(): Obj {
    return this.fst;
  }
}
class SecretL {
  private secret : Int = 0;
}
object oL : SecretL
```

```
package pR;
class PairR {
  private fst, snd : Obj = null;
  public getFirst(): Obj {
    return this.fst;
  }
}
class SecretR {
  private secret : Int = 1;
}
object oR : SecretR
```

The value of `secret` *cannot be leaked at the* J+E *level, however, the compiled counterparts of those packages can leak its value. An* A+I*-level attacker can perform a call to method* `getFirst()` *with current object* o_L *or* o_R*. This will return the* `secret` *field, since fields are accessed by offset (this is an assumption deriving from the usage of standard conventions for compiling objects). As* A+I *code is untyped, nothing prevents this attack from happening. The following trace is then generated by the right-hand side snippet but not by the left-hand side one. No source-level component can be created to replicate the behaviour of this trace since* J+E *programs are well typed, so the aforementioned* A+I *behaviour cannot be replicated.*

$$\texttt{call } o.\texttt{getFirst( )}?\texttt{ret } 1!$$

■

Problem 6 (Type of the arguments) *Similarly to Problem 5, arguments of methods can be exploited in order to mount an* A+I*-level attack. Extend both packages from Problem 5 with the class* `ProxyPair`*, that has a method* `takeFirst` *that inputs a* `Pair` *and returns the output of the method* `getFirst` *called on the input.*

```
...
class ProxyPair {
  public takeFirst( v : Pair ): Obj {
    return v.getFirst();
  }
}
```

In J+E*, both packages are still equivalent. However, an* A+I*-level attacker can pass an object of type* `Secret` *as argument to method* `takeFirst( )` *and the code will leak the contents of field* `secret`*. Assuming there exists an object* `p` *of type* `ProxyPair`*, the differentiating trace in this case is the following:*

$$\texttt{call } p.\texttt{takeFirst(} o \texttt{)}? \texttt{ ret } 1!$$

■

Problem 7 (Guessable object references) *Consider two packages with a class* `Secret` *implemented by two objects and with different implementations of method* `createSecret`*. While that method in* $\texttt{Secret}_L$ *returns a new object, in* $\texttt{Secret}_R$ *it allocates two objects and returns one of them.*

```
package pL;
class SecretL {
  private secret : Int = 0;
  public createSecret( ) : Secret {

    return new Secret( );
  }
}
object oL1 : SecretL
object oL2 : SecretL
```

```
package pR;
class SecretR {
  private secret : Int = 0;
  public createSecret( ) : Secret {
    var x : Secret = new Secret();
    return new Secret( );
  }
}
object oR1 : SecretR
object oR2 : SecretR
```

Object references at the A+I*-level are the address where objects are allocated. Once* $\texttt{p}_L$ *and* $\texttt{p}_R$ *are compiled an attacker can discover that, for example, the identities of* $\texttt{o}_L1$ *and* $\texttt{o}_R1$ *are* 100*, while those of* $\texttt{o}_L2$ *and* $\texttt{o}_R2$ *are* 104*. After method* `createSecret` *is executed, an attacker can see that, for example,* $\texttt{createSecret()}_L$ *returns* 108 *and* $\texttt{createSecret()}_R$ *returns* 112*. With this knowledge, the attacker can guess that* p_R *created an additional object at address* 108*.*

The attacker can thus call methods on objects it does not know of by guessing the address where an object is allocated. This trace, that only belongs to the right-hand side snippet, captures exactly this, as is contains a call $x.\texttt{createSecret( )}?$*. That label intuitively models the calling of method* `createSecret()` *on object* `x`*, which is only possible in the right-hand side snippet.*

$$\texttt{call}\ o.\texttt{createSecret( )}?\ \texttt{ret}\ o'!\ \texttt{call}\ x.\texttt{createSecret( )}?\ \texttt{ret}\ 1!$$

Passing object addresses from a secure program to an external one can give away the allocation strategy of the compiler as well as the size of allocated objects. An attacker that learns this information can then use it to mount attacks such as those presented in Problem 5 and 6. From a technical point of view this means that leaking object addresses and accepting guessed addresses breaks full abstraction of the compilation scheme.

This full abstraction violation happens because object ids in J+E *cannot be guessed or forged, unlike addresses in* A+I*.* ■

Problem 8 (Excessive exception catching) *Consider two implementations of method* `safeCallback` *that both invoke method* `callback` *on object* `cb`*. Even though* `callback` *does not specify that it will throw exceptions, one* *implementation of* `safeCallback` *wraps the call to* `callback` *in a* **`try/catch`** *block.*

```
package p_L;
class C_L {
  public safeCallback( ) : Int {
    try{
      cb.callback( );
    }catch ( v : Throwable ){
      return 1;
    }
    return 0;
  }
}
object o_L : C_L
```

```
package p_R;
class C_R {
  public safeCallback( ) : Int {

    cb.callback( );



    return 0;
  }
}
object o_R : C_R
```

In J+E*, calling* `safeCallback` *on either* $\mathtt{o}_L$ *or* $\mathtt{o}_R$ *always returns 0. However, when they are compiled an* A+I*-level attacker could throw an exception during the execution of* `callback`*. This will cause differentiation between the two implementations, as* `safeCallback` *called on* $\mathtt{o}_L$ *returns 1, while called on* $\mathtt{o}_R$ *it returns 0. This is captured by the following trace for the right-hand side snippet.*

$$\mathtt{call}\ o.\mathtt{safeCallback(\ )}?\ \mathtt{call}\ cb.\mathtt{callback(\ )}!\ \mathtt{call}\ \mathtt{throw(\ v\ )}?\ \mathtt{ret}\ 1!$$

This full abstraction violation happens because the type system of J+E *ensures that exception-throwing methods always declare it in their signatures.* ■

6.2 The Secure Compiler

For the sake of simplicity, we start by developing a secure compiler for a simple fragment of J+E. In the following, assume no dynamic memory allocation (i.e., no **`new`** expressions) and no presence of exceptions (i.e., no **`try/catch`** blocks and no **`throw`** e expressions). Since no new objects are created at runtime, components use static objects and externs for now. Dynamic memory allocation and exceptions will be added in subsequent sections (Section 6.2.2 and Section 6.2.3 respectively).

We follow some standard conventions about how objects are compiled [38]. The compilation of a J+E component C outputs a *protected module* $[\![C]\!]^{\mathsf{J+E}}_{\mathsf{A+I}}$, written in A+I, consisting of a partial memory space and a memory descriptor. The program interacting with the compilation output $[\![C]\!]^{\mathsf{J+E}}_{\mathsf{A+I}}$ should not be able to distinguish modules just by their size. Therefore, a constant amount of memory is reserved for each protected module, independent of the actual memory space required. The protected module is placed in protected memory and the memory descriptor divides the reserved space over the code and the data section. 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 package, class, object, interface and method of the input component. To prevent a target-level module from being distinguished by the order of its methods in memory, packages, interfaces, methods in interfaces, classes, objects and externs are sorted alphabetically. Methods that do not appear in interfaces are compiled based on the order of occurrence in the class.

When an object is compiled, a word is reserved to indicate its class, which is used to dynamically dispatch methods. Methods are dispatched based on offsets through the v-tables, which associate class and method offsets with the corresponding method body. For each object, fields are then given a unique index number starting at 1, based on their order of occurrence. For a field f_i, one word of memory is reserved at the i-th memory address of the memory section of a given object. **Int**eger-typed constants are translated to their corresponding numeric value, `unit` is translated to 0, `true` and `false` are translated to 1 and 0 respectively.

To translate a method body, the compiler processes each expression in turn, translating it into a list of behaviourally-equivalent instructions. Therefore, the compiler is assumed to be *correct* (as formally denoted by Property 1 in Section 7.3). Method bodies cannot contain `exit` expressions, as explained later in Section 7.2. Registers $\mathtt{r}_0$ to $\mathtt{r}_3$ are used as general working registers, return values are passed through $\mathtt{r}_0$. In a method call at the target level, register $\mathtt{r}_4$ identifies the current object (`this`). Method calls are limited to seven parameters, which are passed through registers $\mathtt{r}_5$ to $\mathtt{r}_{11}$. These choices are not critical, the compiler may use registers in a different way and still be fully abstract. A calling convention is set so that register $\mathtt{r}_0$ contains the address to return to after a call at the target level (i.e., any jump instruction between the protected and unprotected sections, located at address x is preceded by an instruction $\mathtt{movi}\ \mathtt{r}_0\ \mathtt{x} + 1$). This choice simplifies the proof of full abstraction of the compilation scheme, we envisage it can be lifted at the price of complicating the proof, without making it unprovable.

Parameters and local variables are given a method-local index number. For each translated method body, a prologue and an epilogue are prepended and appended to it. The prologue allocates and initialises a new activation record on the call stack, the record contains local variables and parameters for the method body. The epilogue deallocates the activation record when the method is done. The code of prologues, epilogues and method bodies is placed in free space in the code section.

To support programming to an interface, and since protected memory can be entered only through entry points, a *method entry point* is generated for each interface-defined method. The entry point for the i-th method is placed at address $i * 128$ 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 that are described in Section 6.2.1. Each entry point forwards the call to the actual method body, so code at entry points consists of two parts: (1) a call to the method's body and (2) a return instruction. When the call to the body returns, the return instruction returns control to the location from which the entry point was called, so the second part of each method entry point is named *exit point*.

In order to specify how the component interacts with external code, assume the component being compiled provides one import package whose interfaces and externs are not implemented in the export package. Refer to this package as *the distinguished import package* (DIP). The DIP contains interface and extern definitions; calls to methods defined in the DIP, on externs defined in the DIP, are called *outcalls*.

Component code is not supposed to implement interfaces defined in the DIP, as the DIP provides functionality that the component required of external programs. External code which provides an implementation for the DIP can implement interfaces defined in the component. This can lead to the dynamic dispatch procedure being called on objects that are not in the protected memory partition. When this case is detected, the compiled component must behave as in the case of an outcall. The address where protected code must jump to when performing an outcall is assumed to be known based on a calling convention set up between protected and unprotected code.

To support returning from an outcall, a specific entry point is created; it is called *the return entry point*. To perform an outcall, the actual return address is first pushed on the call stack, then the address of the return entry point is pushed on the call stack. Control is then transferred to unprotected memory. When the code in unprotected memory 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. The compilation scheme as described above ensures that a module is exited either through an outcall, or through the return statement at the end of an entry point.

To provide a better understanding of the compilation scheme thus far, Figure 6.1 presents the memory layout of the compiled counterpart of the code of Listing 2.1.

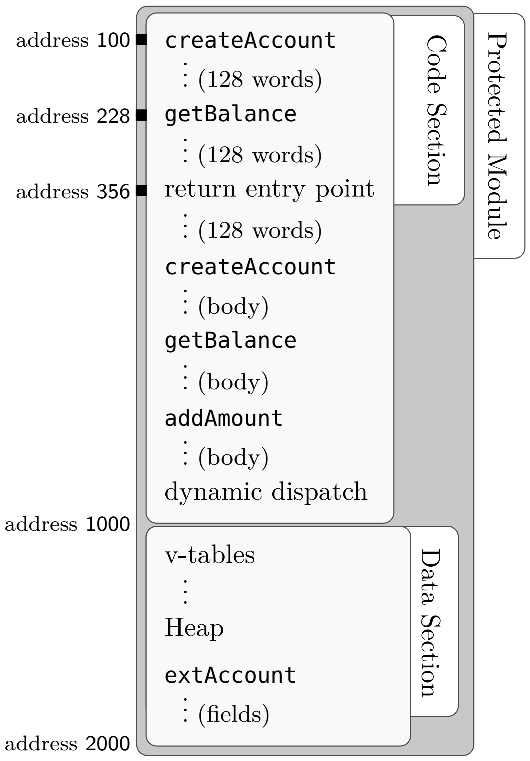

Figure 6.1: Memory layout of the compilation of code from Listing 2.1. Entry points are indicated with ■. The protected memory partition spans from address 100 to address 2000, the code section spans 900 addresses.

6.2.1 Outcalls

This section describes how to securely compile outcalls, i.e., calls to methods of objects defined outside the component being compiled. To address the vulnerabilities of Problems 1 to 4, the compilation scheme is enhanced in the following ways.

Stack security The compiler must ensure the confidentiality and integrity of variables and control structures on the call 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 the protected memory section. The protected module output by the compiler places its activation records exclusively on the secure stack. The secure stack is given a large enough upper bound so that most programs can run without overflowing it. If the stack is overflowed, all registers and flags are cleared and execution halts.

At the start of each entry point, the stack is switched to the secure stack. When leaving the protected module, the stack pointer is restored to its previous address, which is checked to be in unprotected memory. When returning from protected to unprotected code, the return address is also checked to be in unprotected memory. An outcall is performed by first pushing the actual return address onto the secure stack. Then, $\mathtt{r}_4$ is stored in the secure stack so as to be able to restore `this` to the right value once the outcall returns. Next, the stack is switched to the unprotected stack and the address of the return entry point is pushed onto it. Control is then transferred to the context. When the outcall returns, control will first be transferred to the return entry point, which switches back to the secure stack. Then `this` is restored to the expected value and control is transferred back to the actual return address. Because data is written to the unprotected stack during this process, the compiler must ensure that the location of the unprotected stack lies outside of protected memory before it writes to it. Without this check, parts of protected memory might get overwritten.

One problem that can occur related to the control flow of program is that the context could jump to the return entry point when there is no outcall to return from. To prevent this, the compiler initialises the first location of the secure stack to the address of a procedure clears all registers and flags and then halts. The return entry point will jump to this address if it is called when there is no outcall to return from.

Information leakage In J+E, the only way for two objects to communicate, is through well-typed method calls and returns. The compiler must ensure that a target-level attacker cannot use any other communication channels, as this might leak information that should be kept private to the protected module.

The model of A+I inherently provides three ways to exchange information: (1) through reads and writes to unprotected memory, (2) through the flags register and (3) through the general purpose registers $\mathtt{r}_0$ to $\mathtt{r}_{11}$ and SP. Method (1) is precluded because compiled programs never write in unprotected memory. The only J+E constructs that are compiled as a read and a write to memory is field lookup and update; since fields are private, no lookups and updates are possible on fields of object allocated in the unprotected memory. The SP register does not convey private information, because it is restored to the location of the unprotected stack whenever control leaves the protected module. The compiler constrains methods (2) and (3) as follows:

- Flags are cleared (i.e., set to 0) at each outcall and exit point.
- Every general purpose register except $\mathtt{r}_0$ is cleared at each exit point.
- Every general purpose register is cleared at each outcall, except if it is used for passing a parameter.

The compiler generates code at each entry point to enforce these constraints.

Value of primitives The compiler must ensure that all memory locations corresponding to J+E fields and variables contain only values for which there is a corresponding J+E value. The only values of type **`Bool`** are `true` and `false` and the corresponding A+I values are 1 and 0. The compiler enforces this constraint by adding a run-time check at each entry point, to verify that the value of any **`Bool`**-typed parameter is either 0 or 1. An analogous check is added at each outcall to a method with return type **`Bool`**. The same checks are introduced for type **`Unit`**, inhabited by `unit` in J+E which is compiled to 0 in A+I. If any check fails, all registers and flags are cleared and the execution halts.

These checks are needed for all ground types inhabited by a number of values that do not fit a A+I word representation. This is why **`Int`**eger-typed values are not checked, because any A+I word can map to a J+E integer.

In the light of these additions to the compilation scheme, Table 6.1 presents pseudo-code of the routine that is executed at the various kinds of entry points.

Table 6.1: Pseudo-code of entry point routines.

Method p entry point		Preamble to returnback entry point	
1	Switch stack to protected one	a	Switch stack to unprotected one
2	Check primitive-typed parameters	b	Clear flags and unused registers
	(*run method p code*)		(*run outcall code*)
Exit point		Returnback entry point	
3	Clear flags and registers $\mathtt{r}_1$ - $\mathtt{r}_{11}$	c	Switch stack to protected one
4	Switch stack to unprotected one	d	Check primitive-typed parameters

6.2.2 Dynamic Memory Allocation

The vulnerabilities of Problems 5 to 7 are related to dynamic memory allocation, which is now considered to be part of the J+E fragment of components to be securely compiled. To address these vulnerabilities, the compilation scheme is enhanced in a number of ways, as presented below. Since the countermeasure to Problem 7 affects the others, it is presented first.

Object identity The concern of Problem 7 is that an attacker can just guess the object id of a securely-compiled object in order to call methods on it. To address this concern, target-level object identities must be masked as explained below.

To mask target-level object identities, a data structure $\mathcal{O}$ is added to the protected code section. It is a map between natural numbers and target-level object identities that have been passed to external code. Such object identities that are passed to external code are added to $\mathcal{O}$ right before they are passed outside. The index in the data structure is then passed in place of the object identity, the same index must be passed whenever an already recorded object is passed. Indixes in $\mathcal{O}$ are thus passed in a deterministic order, based on the interaction between external and internal code. Code at entry points is responsible for retrieving object identities from $\mathcal{O}$ before the actual method call. Access and retrieval of entries in $\mathcal{O}$ is very fast and can be implemented in $O(1)$. As the only objects in the data structures are the ones the attacker knows, she cannot guess object identities. This does not hamper the functionality of external code as it can only call methods on objects.

Consider for example the right code snippet in Problem 7. There, $\mathtt{o}_R1$ and $\mathtt{o}_R2$ are given indexes 0 and 1 respectively and they are added in $\mathcal{O}$ at compile time (since they are static objects). Once method `createSecret` is called, two objects are created. However, the object saved in variable `x` is not returned, so it is not added to $\mathcal{O}$. The other object is, so it is added to $\mathcal{O}$. Thus, at the A+I level, the compiled counterpart of `createSecret` will return 2 the first time it is called in both $\mathtt{p}_L$ and $\mathtt{p}_R$.

Entry points Table 6.2 describes the code executed at entry points. Both method entry points and the return entry point are logically divided in two parts; they maintain the functionality introduced in Section 6.2.1 and expand them as follows. The first part performs the checks described below and then jumps either to the code that performs the dynamic dispatch or to the outcall. The second part returns control to the location from which the entry point was called.

Table 6.2: Extension to the pseudocode executed at entry points presented in Table 6.1. Loading means that a value is retrieved from the memory, push and pop are operations on the secure stack.

Method p entry point		Preamble to returnback entry point	
1	Load received $v = \mathcal{O}(\mathtt{r}_4)$	a	Push current object $v = \mathtt{r}_4$, return address a and return type m
2	Check that v's class defines method p	b	Reset flags and unused registers
3	Load parameters $\overline{v}$ from $\mathcal{O}$	c	Update $\mathcal{O}$ with leaked object ids and replace object ids to be leaked with indexes from $\mathcal{O}$
4	Dynamic typecheck that $\overline{v}$ have the right types	d	Jump to outcall address
5	Perform dynamic dispatch (*run method p code*)		(*run external code*)
Exit point		**Returnback entry point**	
6	Reset flags and unused registers	e	Pop return type m and check it
7	Update $\mathcal{O}$ with leaked object ids and replace object ids to be leaked with indexes from $\mathcal{O}$	f	Dynamic typecheck that the returned value has the right type
		g	Pop return address a, current object v and resume execution

For type constraints to be respected, the code at entry points performs dynamic typechecks. This checks that a method is invoked on objects of the right type (line 2), with parameters of the right type (line 4), addressing Problem 5 and 6. Similar checks are executed when returning from an outcall, in the returnback entry point (line f). These checks are performed only on objects whose class is defined in the compiled component, as they are allocated in protected memory; no control over externally allocated objects can be assumed. Dynamic typechecks are performed based on the word that indicates the class of a compiled object, that value is checked to be the type or a subtype of the value the method expects.

Resetting flags and registers are as in Section 6.2.1. If any check fails, all registers and flags are cleared and the execution `halts`.

A convention between protected and unprotected code is needed in order to distinguish between indexes in $\mathcal{O}$ from unprotected addresses. For this, assume that the leftmost bit of a word is 1 if it denotes an index in $\mathcal{O}$.

6.2.3 Exceptions

To present secure compilation of exceptions, from this section onward, source-level components can contain **try/catch** blocks and **throw** *e* expressions. Method signatures can specify if the method throws a single, particular exception. Method bodies can include throw statements whose semantics is to "throw" the objects they have as parameters; thrown objects are referred to as exceptions. Finally, method bodies can include try blocks, where some code that can potentially throw an exception is run. These blocks are followed by catch blocks, which intercept thrown exceptions based on the class type of the thrown object. To declare a catch block, the class name of the exception to be caught must be known, thus a component catches only exceptions that are objects whose classes it defines. Overcoming this restriction, thus catching objects based on interface types, is discussed in Section 6.3.4.

Secure compilation of languages supporting exceptions must handle the difficulties that result from the modification of the flow of execution of a program. This flow of execution can be modified when a part of a program **throw**s an exception and another part **catch**es it. Exception handling can be implemented by modifying the runtime of the language so that it knows where to dispatch a thrown exception. Activation records are responsible for pointing to the exception handlers in order to propagate a thrown exception to the right handler.

```
package P-Exc;
  class AccountTest {
    public withdraw() : Unit {
      try{
        new P-Exc.EmptyAccount().getBalance();
      } catch ( e : P-Exc.NoMoneyException ) {
        // handle e
      }
    }
  }
  class EmptyAccount {
    public getBalance() : Unit throws P-Exc.NoMoneyException {
      throw new P-Exc.NoMoneyException();
    }
  }
  class NoMoneyException implements Throwable {···}
```

Listing 6.1: Example of exceptions usage.

In Listing 6.1, the `catch` block of method `withdraw()` in class `AccountTest` defines a handler for exceptions of type `NoMoneyException`. When the activation record for `withdraw()` is allocated, the handler is registered in that activation record. When an exception of type `NoMoneyException` is thrown, the stack is traversed to find the closest handler for exceptions of type `NoMoneyException`. As the stack is traversed and a handler is not found an activation record, that record is popped from the stack and the next record is inspected.

In order to implement throwing an exception in secure code that is caught in insecure code (or vice versa), throwing is securely compiled as outcalls (or calls). Two additional entry points must be created for securely-compiled code: the *throw entry point* and the *throwback entry point*. These entry points forward calls to the secure and insecure exception dispatchers, respectively. The secure exception dispatcher traverses the secure stack looking for handlers for the thrown exception. After an activation record has been inspected and deallocated, if the 'next' allocation record to be inspected is in unprotected memory, the exception is forwarded to the external code through the throwback entry point.

Since exceptions are objects, in order to prevent exploits as in Problem 5, the throwback entry point must remember internally allocated exceptions that are thrown to external code. Data structure $\mathcal{O}$ is used to register such exceptions. Dually, this prevents external code from passing a non-existing object identity to the secure exception handler in place of the object identity of an exception, effectively throwing a non existent exception.

Figure 6.2 presents a graphical overview of how exceptions are handled normally (on the left) and in the presented compilation scheme (on the right). Lower case

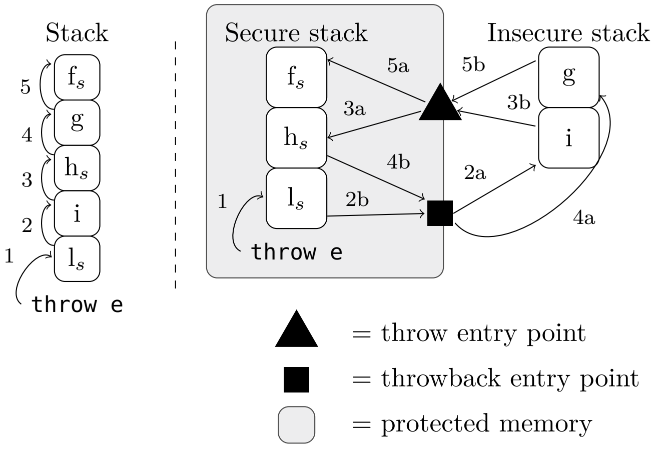

Figure 6.2: Comparison of ways to handle exceptions.

letters indicate the allocation record for the corresponding function. A subscript

s indicates a secure function; the stack grows downward thus exceptions are propagated upwards. The order in which exception handlers are searched is indicated on arrows. The throw and throwback entry point split the same arrow in two parts labelled a and b respectively.

The introduction of two additional entry points may seem to introduce functionality at the target level that the source-level lacks; however this is not the case. Only exceptions of existing types can be thrown and handling exceptions follows the normal course of the stack. The external code could replace an exception, but this is equivalent to the source-level language functionality of **`catch`**ing an exception and **`throw`**ing another one. Jumping to the throwback entry point is exactly like returning after an exception has been thrown, which is also a functionality that the source level has. As for the return entry point, the throwback entry point must not be abused. When code jumps to the throwback entry point, securely compiled code not only checks that an outcall was made, but also that an exception had been trown. This last bit of information is kept track of in the securely compiled code. We can thus conclude that the target level is not granted additional functionality.

To counter the vulnerability of Problem 8, and thus provide support for secure compilation of exceptions, the compilation scheme is enhanced as follows.

Excessive Exceptions Catching The code responsible for compiling outcalls needs to be modified. Information about the possible exceptions thrown by outcalls is known at compile times it appear in the method signature in the DIP. Allocation records must contain a $\texttt{can} - \texttt{throw} - \texttt{exception}$ flag; when they are created, this flag is set to 0. If no exceptions can be thrown by an outcall, that flag is set to 1 in the topmost allocation record of the secure stack before calling the outcall. If an exception is thrown, this flag in the topmost allocation record is checked. If it is set, then no exceptions could be thrown, so a fault is detected so all registers and flags are reset and the execution halts.

When compiling an outcall to method $\texttt{m(}\overline{x}\texttt{)}$ **`throws`** u, type u must also be saved on the secure stack. Code at the throwback entry point must then check that an exception thrown by unprotected code is an exception that could be thrown according to the corresponding unprotected method signature. This is done by performing a dynamic typecheck on the type of the thrown exception. As for method parameters, the dynamic typecheck is performed only on exceptions that are allocated in protected memory. If the typecheck succeeds, then the exception is treated normally, otherwise all registers and flags are cleared and the execution halts. If the exception to be caught comes from unprotected memory (i.e., it is an object id in unprotected memory), it is immediately thrown back to the unprotected code. In fact, as exceptions are caught based on class type, such an exception cannot be caught in the protected code, since the class type of that exception is unknown.

Since exception handling can be securely compiled, the compilation scheme is complete, as it addresses the totality of J+E. This thesis then discusses how to securely compile additional language features before moving to the formalisation of the languages and the proof that this compilation scheme is fully abstract.

6.3 Additional Features

This section describes how to securely compile first-class method references (Section 6.3.1), cross-package inheritance (Section 6.3.2), inner classes (Section 6.3.3) and catching exceptions based on interface types (Section 6.3.4). These features are presented separately because they are not part of J+E, so the correctness of their implementation is only argued and not proven like for the previous features.

6.3.1 First-Class Methods

With first-class methods, method names can be supplied as parameters of other methods in order to be called. Right now the address of an outcall cannot be supplied by external code, that can only supply primitive-typed arguments and objects. With first-class methods, A+I-level attackers can supply arbitrary addresses instead of these parameters, raising the following problem.

Problem 9 (Illegal addresses) *Consider two classes that define a method `doCallback` that inputs a reference to a method `cb`, calls that method and then continues by performing the same computation. These two classes are implemented by two objects:* $\mathtt{o}_L$ *and* $\mathtt{o}_R$*.*

```
1  package p_L;
2  class C_L {
3    private f : Int = 1;
4
5    public doCallback( cb : Unit →
        Unit ) : Int {
6      cb();
7      f += 1;
8      f −= 1;
9      return f;
10   }
11 }
12 object o_L : C_L
```

```
1  package p_R;
2  class C_R {
3    private f : Int = 1;
4
5    public doCallback( cb : Unit →
        Unit ) : Int {
6      cb();
7      f −= 1;
8      f += 1;
9      return f;
10   }
11 }
12 object o_R : C_R
```

Objects $\mathtt{o}_L$ *and* $\mathtt{o}_R$ *are equivalent in* J+E*, as in both objects the method* `doCallback` *always returns 1. Once compiled an* A+I *attacker can differentiate between them by giving the address of the instructions corresponding to line 8 as the outcall* `cb`*. In this case,* $\mathtt{o}_L$ *will decrement* `f` *without first incrementing it, while* $\mathtt{o}_R$ *will increment* `f` *without first decrementing it. This will result in* `f` *having a value of* 0 *in* $\mathtt{o}_L$ *and* 2 *in* $\mathtt{o}_R$*. This is similar to a return-oriented programming attack [103].* ■

To counter this attack, the compiler must ensure the integrity of control flow when jumping from a protected module to an externally supplied address. For a call to a method whose address was externally supplied, a valid destination address is (1) an address outside of the memory bounds of the module, or (2) the address of one of the method entry points. The compiler must add run-time checks for these conditions at each indirect call. In case of a jump to the entry point, the additional checks provided at the entry point and the signature information ensure that the supplied address has a signature that matches that which is specified in the source-level component.

6.3.2 Cross-Package Inheritance

With cross-package inheritance, we indicate a class from an export package extending a class from a *different* export package. This feature is not provided by J+E, as it breaks the encapsulation property of the language that only allows interfaces to be exported to other components and not classes. With cross-package inheritance, both classes and interfaces can be exported to other components. Apart from violating the component-oriented language paradigm, exporting classes can create a breach in the security of the language, which is a second reason why it is not included in J+E. Nevertheless, as there can be cases in which this feature is desirable, and many programming languages allow this, so this paper now discusses how to securely implement it.

Listing 6.2 provides an example of code using cross-package inheritance. Consider a **`public`** class `Animal` defined by protected code (lines 3 to 15) that is extended by class `Feline` in unprotected code (lines 30 to 40). Class `Feline` is ultimately extended by another class of protected code, called `Cat` (lines 16 to 26). In the following, a class extending another class is referred to as the sub-class (as in the case of `Cat`), while the extended class is the super-class (as in the case of `Animal`). Class `Feline` is the sub-class of `Animal` and the super-class of `Cat`. Sub-classes can optionally override methods of the super-class, as is

```
 1 // in protected code
 2 package animal {
 3   public class Animal {
 4     public Animal() {
 5       this.legs = 4;
 6       return this;
 7     }
 8     public sound() : Int {
 9       return 0;
10     }
11     public doSound() : Int {
12       return this.sound();
13     }
14     private legs : Int;
15   }
16   public class Cat extends feline.Feline {
17     public Cat() {
18       super();
19       this.tail = 1;
20       return this;
21     }
22     public sound() : Int {
23       return super.sound();
24     }
25     private tail : Int;
26   }
27 }
28 // in unprotected code
29 package feline {
30   public class Feline extends animal.Animal {
31     public Feline() {
32       super();
33       this.whiskers = 10;
34       return this;
35     }
36     public sound() : Int {
37       return 1;
38     }
39     private whiskers : Int;
40   }
41 }
```

Listing 6.2: Example code to explain cross-package inheritance.

the case of method `sound()`. Within those methods, calls to **`super`** can be used in order to call method `sound()` of the super-class, as in the case of class `Cat`. Alternatively, if a method is not overridden (e.g., as in the case of method `doSound`), calling `doSound()` on an object of type `Feline` executes method body defined in the super-class `Animal` (as in standard object-oriented languages).

Let us firstly describe what changes to a compiler are needed to compile (not necessarily securely) components exporting classes. To include the cross-package inheritance feature and preserve as many benefits as possible from the encapsulation property of J+E, classes that can be extended must appear in import packages. Given an import package, entry points are created not only for interface-defined methods, but also for **`public`** class-defined methods and for constructors. This ensures that whatever is declared as **`public`** in the source code is publicly available in the target code as well.

If the normal compilation scheme were followed, at the target-level an object of type `Cat` would be allocated to a single memory area where fields from classes `Cat`, `Feline` and `Animal` are all allocated. When secure classes can extend insecure ones and vice-versa, some complications arise, as presented in Problems 10 to 12.

Problem 10 (Allocation of a sub-object) *When an object `felix` of type `Cat` is allocated, its fields are `tail`, `whiskers` and `legs`, each coming from a different class. Since all fields are **`private`**, sub-classes do not have access to fields of super-classes (i.e., code in class `Feline` cannot access field `legs` in class `Animal`).*

If `felix` is allocated outside the protected memory partition, private fields of the `Cat` sub-object become accessible to external code, thus breaking the security of the compilation scheme. If `felix` is allocated inside the protected memory partition, two options arise. The first one is placing methods of `Feline` in the protected memory partition, violating the security of the compilation scheme. The second is to place methods of `Feline` in the unprotected memory partition, but at this point these methods cannot access `Cat`'s fields via offset. Getters and setters for fields of `Cat` could be exposed through entry points, but this violates full abstraction, as those methods are not in general available at the source level [1].

These problems also arise when an object of type `Feline` is allocated, but from the dual perspective. ■

To allocate an object whose class extends classes that are not defined in the same component, the object is split into sub-objects, one per class [116]. Given a class c that has n super-classes (a number that is known statically), an object of type c occupies: 1 word for its type, 1 word for a possible bottom-most object id, n words, one per super-object id. If the object has not been extended by other classes, it is the *bottom-most object* of its class hierarchy. The second word of its memory representation will be 0. The role of the bottom-most object is described now.

Given a class hierarchy such as that of Listing 6.2, the sub-object approach to object implementation ensures that there is a sub-object per class in the hierarchy. This effectively creates a hierarchy of sub-objects; in the following, we use the term super-objects w.r.t. another object to refer to the sub-objects implementing the super-classes of that object. The bottom-most object in this sub-object hierarchy must regulate the creation of the super-objects. In Listing 6.2, code inside the `Cat` constructor is responsible to call the `Animal` constructor and the `Feline` one. The target-level representation of an object must include fields that indicate the sub-objects of which a `Cat` object is made of. In Listing 6.2, a `Cat` object must have a field that indicates the `Cat` type, a field pointing to its `Animal` sub-object id, and another field pointing to its `Feline` sub-object id. These fields are used to implement specific functionalities as described below. For example, method `doSound` needs to be always dispatched to the `Animal` sub-object.

Two entry points are generated for constructors, one for a normal call and one for a **`super`** call. Intuitively, the former should be called only when a `Feline` object is created. The latter should be called when a sub-object of type `Feline` is created. So, for allocating a `Feline` object, a call to the normal constructor should be made. When `Cat` calls the `Feline` constructor, it calls it via **`super`**, so it should jump to the super-constructor entry point for `Feline`.

Code at the super-constructor entry point does not allocate an object of all extended super-types. The constructor of the bottom-most object must call the constructors of all super-classes. When allocating a `Cat`, code at its constructor is responsible to create a `Felix` sub-object and an `Animal` sub-object.

An alternative could be that each constructor just calls the direct super constructor. This is a problem because intermediate classes in the inheritance hierarchy (such as `Feline` here) could violate this chain of super-calls. This violation could go undetected and cause security violations.

Following is a pictographic representation of the target-level memory layout of a `Cat` object `felix` and of a `Feline` object `richard` (Figure 6.3). To allocate object `felix`, an `Animal` object `felix-a` is created in protected memory, with field `legs`. Then a `Feline` object `felix-f` is created in unprotected memory, with field `whiskers` and finally a `Cat` object `felix-c` is created in protected memory, with field `tail`. The object id of the `Cat` object `felix` is `felix-c`.

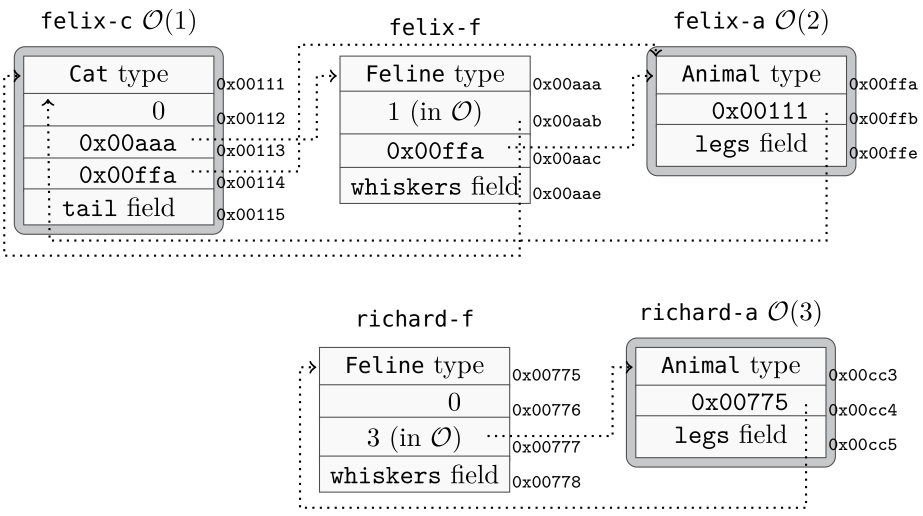

Figure 6.3: Memory layout of object `felix:Cat` and of object `richard:Feline`. For objects located in the protected memory partition, the notation $\mathcal{O}(n)$ indicates that they have masking index n. The protected memory is indicated with a dark grey background; fields whose contents are addresses are accompanied by a dashed line denoting where the address points to.

The creation of sub-objects may seem to break full abstraction of the compilation scheme in a way similar to what Abadi discovered for inner classes [1] in the early JVM. In fact, target level external code is given the functionality to call `felix-a.sound()`, which is not explicitly possible in the source-level language. However, `felix-f.`**`super`**`.sound()` is an implicit call to the `sound()` method of `Animal`, functionality that the source-level language already has. This way of handling cross-package inheritance does not add functionality at the target level, so it does not break full abstraction of the compilation scheme.

Problem 11 (Correct method dispatch) *Another concern with the code of Listing 6.2 is the dispatch of method* `sound` *in method* `doSound` *in class* `Animal`*. If that method is called on an object of type* `Cat`*, it returns* 1*, while when it is called on an object of type* `Animal` *is returns* 0*. However, when dispatching method calls, the sub-object* `Animal` *does not know if it is part of a larger object or not, so it cannot dispatch to its* `Cat` *sub-object because it does not know its id.* ■

To address this concern, the super-constructor entry points take these additional parameters: the object id of the bottom-most object and the object id of the sub-object of its extended super-class. The super-constructor entry point for `Animal` could save the object id of a `Cat` sub-object or of a `Feline` sub-object depending of which class constructor calls it. At a super-constructor entry point there is an additional check that the supplied ids are addresses in unprotected memory.

Unless it is 0, the id of the bottom-most object is used as the current object id to resolve all method dispatches. If nothing is stored in the bottom-most field of the target-level representation of an object, the object is not a sub-object of a larger object, so its object id is used as current object. For example, when a `Feline` object is created as part of a `Cat` object, it stores the id of its `Cat` sub-object and of its `Animal` sub-object (to dispatch eventual super-calls). If a `Feline` object is created as a new object, its bottom-most field will not be initialised to another object id.

The last concern is more related to a correct implementation of sub-objects rather than to security.

Problem 12 (Super method invocation) *Another concern of the presented sub-object splitting approach and of using the bottom-most sub-object id for dispatch is the following. Consider an object `felix` of type `Cat`, which is composed of sub-objects `felix-c`, a `felix-f` and a `felix-a` corresponding to the `Cat`, `Feline` and `Animal` classes. When `felix.doSound()` is executed (technically it is `felix-c.doSound()`), it is dispatched to the `felix-a` identifier. Then, that method calls `felix-c.sound()` which is dispatched to `felix-c`. Then, the `Cat` implementation of `sound` calls to the **`super`** one, so a call is dispatched to the `sound` entry point of `felix-f` (let us assume unprotected code also has entry points). There, the `felix-f` sub-object knows that its current object to be used is `felix-c`, so a loop is entered where the dispatch bounces between the two `sound` functions of `Cat` and `Feline`.* ■

In order to address this concern, **`super`** entry points are created for all **`public`** methods, as for constructors (see Problem 10). When a super-call is identified, the current sub-object id is used as the current object id in place of the bottom-most sub-object id stored in the target-level object representation. In these super-call entry points, an additional check is made that the current object has indeed a bottom-most non-zero sub-object id. If that field is 0, then the **`super`** should not be called on that object because it has not been extended. If that check does not succeed, 0 is placed in $\mathtt{r}_0$ and the execution is halted.

6.3.3 Inner Classes

An inner classe is one that is defined inside another class; one such example is presented in Listing 6.3. An inner class has access to private fields of the class it is defined within. Inner classes have not been included in the formalisation of J+E as to keep it as simple as possible.

```
class AccountClass implements P-Import.Account {
 AccountClass() { counter = 0; }
 private counter : Int;

 class Inner { // Inner has access to counter }
}
```

Listing 6.3: Example of an inner class.

To securely compile inner classes, we follow an approach inspired by Abadi [1]. That approach is shown to break full abstraction of compilation in an early version of the JVM, however it does not violate full abstraction in our setting.

For inner classes to be securely compiled they are compiled as normal classes in the protected memory partition, in the usual fashion. To implement access from the inner class to the private fields of the surrounding class in a JVM style [42], a getter and a setter for each private field are created. In the case of Listing 6.3, class `AccountClass` is extended with getters and setters for the `counter` field when compiled. Access from `Inner` to `counter` is compiled as method calls via the getter and setter.

In the JVM setting of the work of Abadi, the additional target-level methods are not available at the high level. Thus other target-level code besides the inner classes can call those methods, achieving something that was not possible at the high level. In our secure compilation scheme, the additional methods are available in the surrounding class. However the additional methods are not made available through entry points, thus the external code cannot invoke them. This means that the addition of inner classes to the secure compilation scheme preserves the full abstraction property.

6.3.4 Catching Exceptions Based on Interface Type

This section discusses how to lift the restriction of catching exceptions only based on class types (Section 6.2.3) to catching exceptions based on interface type.

In the secure compilation scheme of Chapter 6, catching exceptions is done only based on class types. Because of the strong encapsulation property of J+E, this means that a component can only catch exceptions that are objects whose class is defined in that component. However, this can be undesirable in certain software systems (Listing 6.4). For example, consider a programmer

```
1 public writePDF( pdf : PDF ){ // protected code
2   File f = new TextFile( );
3   try{
4     f.write( );
5   }catch( x : FileNotFoundException){
6     // do something
7   }
8 }
```

Listing 6.4: Example of code where catching exceptions based on interfaces is desirable.

who is writing a PDF viewing program that relies on a `File` library for creating, reading and writing files. When writing to a file (with method `write`), the file may be inexistent, and this generally results in the read operation throwing a FileNotFound exception (line 5). To be able to handle the file manager's `FileNotFound` exception, the PDF application programmer must implement a class `FileNotFound`.

To implement catching exceptions based on interface types, type information must be known at the target level. For catching an exception based on an interface type, the type of the thrown object must be accessible to all who can catch an exception: both protected and unprotected code. Then, as both code bases agree on a bit-representation for interface types, they must check that the type of the exception to be caught is implemented by a throws object.

A first concern for supporting catching exceptions based on interface types is how to add the type information at the target level. In fact, adding this information is necessary for code to function properly; without it there is no way of knowing the type of an object and thus to catch an object when it is thrown. This affects the target-level object representation of both unprotected objects and protected ones.

For unprotected objects (i.e., those residing in unprotected memory), a convention is set up: every object id must be followed by a pointer to a list of implemented interfaces. Concerning list encodings, they must firstly define their length (in a word), then all interfaces. So, when checking whether an interface is in a list, a bound is provided to prevent scanning the whole memory looking for an interface type. Any time the securely compiled component receives an unprotected object (via a method call or a return), it must check that this convention is respected. There are three cases in which it cannot be respected: (i) if the pointer of the list is not an address in unprotected memory; (ii) if the length of the list is greater than the maximum number of interfaces it can implement or (iii) if an interface is not a valid encoding. If any of these cases are detected, all registers and flags are reset and the execution is halted.

Remark 1 (Dynamic typechecks on parameters) *As the protected code has access to the interface type information of unprotected objects, the dynamic typechecks performed at entry points can be done on unprotected objects as well. Table 6.2 can be expanded with dynamic typechecks on unprotected objects.*

For securely-compiled objects, the masking associated to them is changed since the masking must capture the interface types as well. Masked object ids then become two words long: the first word is the masking index in $\mathcal{O}$ while the second word is a pointer to a list of interface types. As already mentioned, a convention between protected and unprotected code is set up to encode interfaces as bit values so. This list of interface types contains all bit-representation of interface types in alphabetical order.

For a component that defines n interfaces in its import packages, $P(n)$ lists are made, one for each combination of possible implementable interfaces. Call this list of list simply the list of implemented interfaces. For example, if a component declares interfaces **`Account`** and **`Bank`**, the following lists are created.

```
Account
Bank
Account,Bank
```

A second concern is how to represent the lists of interfaces at the target level. The lists of interfaces is written in unprotected memory for it to be readable from unprotected code This makes the output of the compilation scheme be not just a protected module but a protected module and some unprotected memory. Only the unprotected code uses the lists of interfaces for catching exceptions so, when unprotected code is not a malicious attacker, this list will not be tampered with. If the unprotected code tampers with this list, only the unprotected code will be affected by tampering with the list.

An alternative to the list is to encode each possible combination of interfaces as a bit mask: sort the interfaces alphabetically, if an object implements interface j and i, then only the jth and the ith bits are set to 1. This is only possible if there are a total of less than ℓ interfaces defined in the J+E component. This alternative representation, when less than ℓ interfaces are present in a system, allows for the object id and its interfaces to be expressed all in the same word for optimisation purposes.

Catching exceptions based on interface types grants additional power to an attacker *already at the source level*. In fact, since interfaces are all known, at the source level one can write code that tests whether an object implements any interface (Listing 6.5). That same power is added at the target level by

```
// unprotected code
public receiveAccount ( account : Account ){
  try{
    throw account;
  }catch ( x : Account ){
    try{
      throw x;
    }catch ( y : Bank ){
      // account has type Account and Bank
    }catch (y : Account ){
      // account only has type Account
    }
  }catch (x : Bank){
    // account only has type Bank
  }
}
```

Listing 6.5: Example code that discerns all interfaces implemented by an object.

augmenting the representation of a target-level object id. Since the knowledge of interface types is already present at the source level, we argue that adding it at the target level does not violate full abstraction.

Chapter 7

Proof of Full Abstraction for $[\![\cdot]\!]^{\mathsf{J+E}}_{\mathsf{A+I}}$

I have a cunning plan!

Baldrick, Black Adder

This chapter presents the proof that the compiler of Section 6.2 is fully abstract. This proof revolves around an algorithm, which this chapter defines (Section 7.1). This chapter then discusses why is the algorithm applicable (Section 7.2) and then presents the actual proofs (Section 7.3).

7.1 The Algorithm

This section presents the algorithm which takes as input two different, target-level traces $\overline{\alpha}_1$ and $\overline{\alpha}_2$ and two components C_1 and C_2 and outputs a source-level component C that differentiates between C_1 and C_2. Traces $\overline{\alpha}_1$ and $\overline{\alpha}_2$ were generated by $[\![C_1]\!]^{\mathsf{J+E}}_{\mathsf{A+I}}$ and $[\![C_2]\!]^{\mathsf{J+E}}_{\mathsf{A+I}}$ when interacting with the same, unknown external memory.

This section firstly describes the general idea of the algorithm (Section 7.1.1). Then, it presents the starting point of the codebase of the algorithm (Section 7.1.2) and it presents examples of the expected output of the algorithm when different traces and components are input (Section 7.1.3). The examples illustrate crucial cases the algorithm needs to considers when creating the output component.

In the following, the adjective *internal* denotes objects (classes) that are allocated (defined) by components C_1 and C_2. The adjective *external* denotes objects (classes) that are allocated (defined) by the output component.

7.1.1 General Idea

The algorithm analyses actions in the target-level traces $\overline{\alpha}_1$ and $\overline{\alpha}_2$. Those actions can be: call, return, callback, returnback and termination ($\surd$). Actions that appear at even-numbered positions in a trace are calls or returnbacks, generated from the external memory. Actions that appear at odd-numbered positions are returns or callbacks, generated by C_1 or C_2. This partitioning is because execution starts in unprotected memory.

For the algorithm to be correct, it must detect when two different actions are encountered at the same odd-numbered position in the traces. Assuming the first different actions are at index i, the algorithm produces code that replicates the first $i-1$ actions. Then, it produces code that, based on the difference in the i-th action, either diverges or terminates based on which component it is interacting with.

The algorithm has been implemented in Scala, and it outputs Java components that adhere to the J+E formalisation.[1] For implementation purposes, instead of diverging in a case and terminating in the other the implementation terminates with value 1 or 2. This formulation of contextual equivalence is equivalent to ours [31], yet more amenable to an implementation, as divergence can be hard to tell from a long-lasting computation. Nevertheless, in the examples presented below, differentiation will be achieved by terminating in a case and diverging in the other.

7.1.2 Starting Point

The algorithm starts by creating a knowledge base about C_1 and C_2. The knowledge base contains all signatures of internally- and externally-defined methods, as well as source- and target-level identities of static objects and externs. This is because the algorithm needs to be able to differentiate, for example, whether a type is internally or externally defined or what the identities

[1] Available at http://people.cs.kuleuven.be/∼marco.patrignani/Publications.html.

of static objects are. Then, a code skeleton for the output component is created, based on the structure of the distinguished import package (DIP) of C_1 and C_2.

A single class `Main` is created, it implements all interfaces i defined in the DIP. An object `main` of type `Main` is then created. Class `Main` contains dummy implementations of all methods defined in all interfaces i. These method implementations return a value whose type matches the expected return type: 0 for type `Int`, `unit` for type `Unit` and **`null`** otherwise. Moreover, class `Main` contains a `main` method and a method called `defaultCreate()` that returns a new `Main` object (used for allocation of external objects).

7.1.3 Examples of Algorithm Code for Differentiation

Examples 22 to 29 present different implementations of C_1, on the left, and of C_2, on the right. Components C_1 and C_2 are modifications of the code in Listing 2.1, whose DIP is defined in Listing 7.1 below. Omitted code is the

```
package PIMP;
interface Transaction extends Atomic {
  public createTrans() : Transaction;
  public callback( arg : Transaction ) : Unit;
}
interface Atomic {
  public lock() : Int;
}
extern extTrans : Transaction;
```

Listing 7.1: Example of a distinguished import package (DIP).

same in both C_1 and C_2 and can be found in Listing 2.1. Each code fragment is followed by the target-level trace it generates: $\overline{\alpha_1}$ and $\overline{\alpha_2}$ for C_1 and C_2 respectively. The examples also describe what the algorithm must do in order to create the correct output before presenting the output produced for each case.

The target-level traces will be massaged to aid understanding. For example, given that object `extAccount` is compiled to identity 0x123 and that the entry point of method `createAccount` is located at address 0x456, the target-level label `call` 0x456$(r[\mathtt{r}_4 = \text{0x123}])?$ is written as `extAccount.createAccount()`. This abstraction is safe, as it does not introduce additional information, it merely massages the present one into a more human-readable form. Numbers in italic font, e.g., *1*, refer to indexes from the masking table $\mathcal{O}$, while identities of externally allocated objects are numbers in hexadecimal form.

Example 22 (Different returned values) *Consider the following implementations for C_1 and C_2.*

```
object extAccount: AccountClass{
  counter = 1
}
public getBalance(): Int{

  return counter;
}
```

```
object extAccount: AccountClass{
  counter = 1
}
public getBalance(): Int{
  counter += 1;
  return counter;
}
```

Trace $\overline{\alpha_1}$ for C_1 is `extAccount.getBalance()`?·`ret 1`!·`extAccount.getBalance()`?·`ret 1`!, while trace $\overline{\alpha_2}$ for C_2 is `extAccount.getBalance()`?·`ret 1`! ·`extAccount.getBalance()`?·`ret 2`!. In this example, the code produced needs to differentiate between C_1 and C_2 based on the type of expected returned values. These types can be either: primitive, internal, external. With primitive-typed values the differentiation is based on the different values returned by C_1 or C_2, in this example C_1 returns `1` and C_2 returns `2`.

This example highlights how both the algorithm and the code produced need to keep track of the index of the action they replicate. To that end, the algorithm maintains a global variable that keeps track of the index of the action being replicated. The code produced is extended with a class `Helper` and a static object `oc` implementing it. `Helper` contains a field `step` with methods `getStep()` and `incrementStep()`, the latter increases the value of `step` by one. Additionally, it contains a method `diverge()` that recursively calls itself, which is used to achieve divergence. As `oc` is static, its fields are global variables for the output component.

The algorithm outputs the code of Listing 7.2. The first actions generate the code in lines 2 to 6, thus it is wrapped in an if-statement that makes the generated code take place only when the considered action is the first: i.e., `step` is 0. The second actions are responsible for incrementing `step` in line 5. The third actions generate the code in lines 7 to 11, while the fourth actions, the different ones, generate the code in line 10.

The approach of this example is similar to what the algorithm does in case the difference in the traces is in primitive-typed parameters of a callback. In that case, instead of creating fresh variable `varb`, the code produced performs the differentiation by using the name of the parameter which has the different value. ⊡

Example 23 (Different internally-typed returned objects)

```
public main ( args : String[] ) : Unit {
  if ( oc.getStep() == 0 ) {
    oc.incrementStep();
    var vara : Int = extAccount.getBalance();
    oc.incrementStep();
  }
  if ( oc.getStep() == 2 ) {
    oc.incrementStep();
    var varb : Int = extAccount.getBalance();
    if ( varb == 1 ) {
      exit( 1 );
    } else {
      oc.diverge();
    }
  }
}
```

Listing 7.2: Output of the algorithm for Example 22.

```
public createAccount() : Account {
  return this;
}
```

```
public createAccount() : Account {
  return new AccountClass();
}
```

Trace $\overline{\alpha_1}$ is `extAccount.createAccount()`$?\cdot$ `ret extAccount`$!$*, while trace $\overline{\alpha_2}$ is* `extAccount.createAccount()`$?\cdot$ `ret` *1*$!$*. In this case the code produced must be able to differentiate between two return values that are internal objects. They are given different indexes in $\mathcal{O}$. Here, C_1 returns a known object:* `extAccount`*, while C_2 returns a new object: index 1 in $\mathcal{O}$.*

To achieve differentiation in this case, the code produced needs to keep track of internally allocated objects. For this it relies on a list `internals` *provided by* `oc`*. In order for internal objects to be accessible, they are wrapped with a new class:* `Internal` *that has two fields. The first, of type* `Obj`*, contains a reference to an internal object. The second,* `name`*, can be used to filter the search for objects, that field contains the target-level id as found in the traces. No two objects with the same* `name` *can be added to* `internals`*. Elements of this list can be accessed via method* `getNameByObject( o )`*, which returns the* `name` *of object* `o` *or* `null` *if* `o` *is not in* `internals`*. The algorithm has a table with target-level identities of all dynamically-allocated objects in order to generate correct code when retrieving* `internals` *as in line 6 in the code below. Table* `internals` *is initialised with entries for all known static objects.*

The algorithm outputs the code of Listing 7.3. Line 5 has no effect, since `internals` *already has an entry for* `extAccount`*. In case C_1 and C_2 were*

```
 1 public main ( args : String[] ) : Unit {
 2   if ( oc.getStep() == 0 ) {
 3     oc.incrementStep();
 4     var f : Account = extAccount.createAccount();
 5     oc.addInternal( new Intern ( f, "extAccount" ) );
 6     if ( "extAccount" == oc.getNameByObject( f ) ) {
 7       exit( 1 );
 8     }else {
 9       oc.diverge();
10     }
11   }
12 }
```

Listing 7.3: Output of the algorithm for Example 23.

swapped, line 5 would bind `f` *to name 1, ensuring the execution of the else-branch in the if statement in line 6.*

This example scales to different internally-typed parameters in a callback. ⊡

Example 24 (Different method of callbacks)

```
public createAccount() : Account {
  extTrans.lock();

}
```

```
public createAccount() : Account {
  var b : Transaction = extTrans.
        createTransaction();
}
```

Trace $\overline{\alpha_1}$ is `extAccount.createAccount()`$?\cdot$`extTrans.lock()`$!$*, while trace $\overline{\alpha_2}$ is* `extAccount.createAccount()`$?\cdot$`extTrans.createTransaction()`$!$*. In this example, C_1 performs a callback to method* `lock`*, while C_2 performs it to method* `createTransaction`*.*

To achieve differentiation in this case, the algorithm needs to keep track of the current method*, since it indicates where the differentiating code will be placed. The current method is recorded in a stack which is initially set to method* `main`*. Callbacks indicate that the current method is changed to a new entry, returnbacks indicate that the current method is restored to a previous one. Thus, whenever a callback to method* `m` *(implemented in class* `Main`*) is performed, an entry of the form* `Main.m` *is pushed on the stack. A returnback pops the head of the current method stack.*

The algorithm outputs the code of Listing 7.4. Notice that the if-statements of lines 8 and 12, whose addition was discussed in Example 22, help the code produced differentiate between C_1 and C_2 in this case as well. Should methods `createTransaction()` *or* `lock()` *be called multiple times the if-guard ensures* *that the differentiation only happens at the right time.* ⊡

```
public main ( args : String[] ) : Unit {
  if ( oc.getStep() == 0 ) {
    oc.incrementStep();
    var f : Account = extAccount.createAccount();
  }
}
public createTransaction() : Transaction {
  if ( oc.getStep() == 1 ) {
    oc.diverge();
  }
  return null;
}
public lock() : Int {
  if ( oc.getStep() == 1 ) {
    exit( 1 );
  }
  return 0;
}
```

Listing 7.4: Output of the algorithm for Example 24.

Example 25 (Different callees of callbacks)

```
public createAccount() : Account {
  var b : Transaction = extTrans1.
      createTransaction();
}
```

```
public createAccount() : Account {
  var b : Transaction = extTrans2.
      createTransaction();
}
```

Trace $\overline{\alpha_1}$ *is* `extAccount.createAccount()`$?\cdot$ `extTrans1.createTransaction()`$!$, *while trace* $\overline{\alpha_2}$ *is* `extAccount.createAccount()`$?\cdot$`extTrans2.createTransaction()`$!$. *In this case the difference is the external object on which the second callback is performed. Here,* C_1 *calls* `createTransaction()` *on* `extTrans1`*, while* C_2 *calls it on* `extTrans2`*.*

In order to achieve this differentiation, the code produced needs to keep track of external objects similarly to how it needed to keep track of internal objects in Example 23. All external objects must be bound to a name, just as the internally allocated ones are. For this purpose, a class `Listable` *is created, class* `Main` *extends it.* `Listable` *contains a* `name` *and a* `type` *field, with getters and setters. It also contains a method* `setAndRegister( n , t )` *that sets* `name = n`*,* `type = t` *and adds the object to a list of* `Listable` *called* `externals` *that is kept in object* `oc`*. Object* `oc` *contains method* `getExternal( n , t )` *to retrieve these objects based on* `name` *and* `type`*.*

The algorithm outputs the code of Listing 7.5. Fields `name` *and* `type` *for external*

```
// same main as in Example 24
public createTransaction() : Transaction {
  if ( oc.getStep() == 1 ) {
    if ( this.getName() == "extTrans" ) {
      exit( 1 );
    } else {
      oc.diverge();
    }
  }
  return null;
}
```

Listing 7.5: Output of the algorithm for Example 25.

static objects are initialised in the first instructions of the `main`*. That code is omitted for brevity.* ⊡

Example 26 (Different callees of callbacks #2)

```
public createAccount() : Account {
  var b : Transaction = extTrans.
      createTransaction();
  b = b.createTransaction();
}
```

```
public createAccount() : Account {
  var b : Transaction = extTrans.
      createTransaction();
  b = extTrans.createTransaction();
}
```

Trace $\overline{\alpha_1}$ is `extAccount.createAccount()`$?\cdot$`extTrans.createTransaction()`$!\cdot$ `ret 0x6`$?\cdot$`0x6.createTransaction()`$!$ *while $\overline{\alpha_2}$ is* `extAccount.createAccount()`$?\cdot$ `extTrans.createTransaction()`$!\cdot$`ret 0x6`$?\cdot$`extTrans.createTransaction()`$!$*. In this case the difference is the external object on which a callback is performed. Here, C_1 calls* `createTransaction()` *on* `0x6`*, while C_2 calls the same method on* `extTrans`*.*

The algorithm outputs the code of Listing 7.6. Lines 14 to 17 ensure that if an external object is not found in the list `externals`*, it is allocated by calling to the default factory method and then added to* `externals`*. Fields* `name` *and* `type` *for external static objects are initialised to the right value in the first instructions of the* `main`*. That code is omitted for brevity.* ⊡

Example 27 (Different externally-typed callback parameters)

```
// same main as in Example 24
public createTransaction() : Transaction {
 if ( oc.getStep() == 1 ) {
   oc.incrementStep();
 }
 if ( oc.getStep() == 2 ) {
   oc.incrementStep();
   var h :Transaction = oc.getExternal( "0x6", "Transaction" );
   if ( h == null ) {
     h = staticForTransaction.defaultCreate();
     ( ( Listable ) h ).setAndRegister( "0x6", "Transaction" );
   }
   return h;
 }
 if ( oc.getStep() == 3 ) {
   if ( this.getName() == "0x6" ) {
     exit( 1 );
   } else {
     oc.diverge();
   }
 }
 return null;
}
```

Listing 7.6: Output of the algorithm for Example 26.

```
public createAccount() : Account {
  var b : Transaction = extTrans.
        createTransaction();
  b.callback( b );
}
```

```
public createAccount() : Account {
  var b : Transaction = extTrans.
        createTransaction();
  b.callback( extTrans );
}
```

Trace $\overline{\alpha_1}$ is `extAccount.createAccount()`$?\cdot$ `extTrans.createTransaction()`$!\cdot$ `ret 0x6`$?\cdot$`0x6.callback( 0x6 )`$!$, *while trace $\overline{\alpha_2}$ is* `extAccount.createAccount()`$?\cdot$ `extTrans.createTransaction()`$!\cdot$ `ret 0x6`$?\cdot$ `0x6.callback( extTrans )`$!$. *This example presents the expected output in case the difference is in a parameter of a callback. The code produced relies on the notions defined in Example 25, using the field* `name` *of external objects to achieve differentiation.*

The algorithm outputs the code of Listing 7.7. Casting `arg` *to* `Listable` *is needed in order to make sure the call to* `getName()` *succeeds. In fact,* `arg` *is known to implement interface* `Transaction`*, which has no connection with class* `Listable` *that defines method* `getName()`*.*

Casts are not present in J+E*, yet casting an object o to a type t can be modelled by throwing o and catching an exception of type t. The cast is presented for the sake of simplicity.* ⊡

```
// same main and createTransaction from Example 25,
// except that lines 20 - 22 are removed
public callback( arg : Transaction ) : Unit {
  if ( oc.getStep() == 3 ) {
    if ( ( ( Listable ) arg ).getName() == "0x6" ) {
      exit(1);
    } else {
      oc.diverge();
    }
  }
}
```

Listing 7.7: Output of the algorithm for Example 27.

Example 28 (Traces with different actions)

```
public createAccount() : Account {
  return extAccount;

}
```

```
public createAccount() : Account {
  var b : Transaction = extTrans.
        createTransaction();
}
```

Trace α_1 is `extAccount.createAccount()`$?\cdot$ `ret` $1!$, while trace α_2 is `extAccount.createAccount()`$?\cdot$`0x6.createTransaction()`$!$. In this case the algorithm needs to identify the two different locations where execution will be after the different actions are executed. The concept of current method introduced in Example 24 can be used in this case as well in order to determine where to place the code that performs the differentiation.

The algorithm outputs the code of Listing 7.8. ⊡

```
public main ( args : String[] ) : Unit {
  if ( oc.getStep() == 0 ) {
    oc.incrementStep();
    var f : Account = extAccount.createAccount();
    exit( 1 );
  }
}
public createTransaction() : Transaction {
  oc.diverge();
  return null;
}
```

Listing 7.8: Output of the algorithm for Example 28.

Example 29 (Traces of different length)

```
public createAccount() : Account {
  while ( 1 == 1 ) { skip; };
  return null;
}
```

```
public createAccount() : Account {

  return new AccountClass();
}
```

Trace α_1 is `extAccount.createAccount()`?, while trace α_2 is `extAccount.createAccount()`? $\cdot$ ret 1!.

The algorithm outputs the code of Listing 7.9. When control is returned to

```
public main ( args : String[] ) : Unit {
  if ( oc.getStep() == 0 ) {
    oc.incrementStep();
    var f : Account = extAccount.createAccount();
    exit( 2 );
  }
}
```

Listing 7.9: Output of the algorithm for Example 29.

`main` after a call to `createAccount()`, it means that the output component is interacting with C_2. In this case the code produced terminates via the expression of line 5. Divergence is accomplished by C_1. ⊡

As previously mentioned, exceptions are objects. Throwing and receiving exceptions are compiled as call and callbacks to the throw and the throwback entry points. These calls have a single parameter: the object id of what is being thrown. Whenever a call to the throw entry point is detected in the traces, a **`throw`** expression is written in place of a normal method call. The signature of the method must declare that it will throw an exception, otherwise the last action in the trace will be this thrown exception. This means that after this trace the execution will halt, as no exception to be thrown is registered, but this contradicts the presence of a difference in $\overline{\alpha_1}$ and $\overline{\alpha_2}$. Calls to methods that can throw exceptions are wrapped in a **`try/catch`** block. If a jump to the external exception handler is detected, then the code is written in the **`catch`** block. Otherwise if a normal action is detected, the code is written in the **`try`** block, below the method call.

As further clarification, the pseudo-code of the algorithm is presented in Listing 7.10 below.

```
//inputs two traces and two components and returns a J+E component that
     differentiates between the two components
Algorithm(trace1, trace2, component1, component2)
  knowledge_base = ... // get interfaces, methods, etc. from components
  index = 0
  while( index <= min(length(trace1), length(trace2)))
    elem1 = trace1[index]
    elem2 = trace2[index]

    if( elem1 is a ? action ) // so is elem2 and elem1 == elem2
      WriteCode( " if (oc.getStep() == "+ index +")" )
      case elem1 of
      | "call p v1 ... vn" →
          meth = lookup "p" in methods_Table of knowledge_base
          curr_obj = Get A+I value corresponding to( "v4" )
          w1 = Get A+I value corresponding to( "v5" )
          ...
          wn = Get A+I value corresponding to( "vn" )
          WriteCode( "curr_obj.meth("+ w1 +", ...,"+ wn +");" )
      | "ret p v" →
          w = Get A+I value corresponding to( "v" )
          WriteCode( "return "+ w +";" )
    else // both elem1 and elem2 are ! actions

      if( elem1 ≠ elem2 )
        // write differentiating code as in Examples 22 to 29
        case elem1, elem2 of
        | "call P v1 ... vn", "call Q v1 ... vn" →... // different methods
        | "call p v1 ... V4 ... vn", "call p v1 ... W4 ... vn" →...
          // different callee
        | "call p V5 ... vn", "call p W5 ... vn" →... // different parameter
             #1
        | ... // all other cases for different parameters
        | "call p v1 ... vn", "ret p v" →... // different actions
        | "ret p V", "ret p W" →... // different returned value
        | "ret p v", " " →... // different actions
        | "call p v1 ... vn", " " →... // different actions
      else // elem1 and elem2 are the same ! action
        case elem1 of
        | "call p v1 ... vn" →
          knowledge base. add ( "v1", Type for "v1", argument for "v1")
          ...
          knowledge base. add ( "vn", Type for "vn", argument for "vn")
        | "ret p v" →
          knowledge base. add ( "v", Type for "v", argument for "v")
    index++;
```

Listing 7.10: Pseudo-code representation of the algorithm.

7.2 Algorithm Applicability

All ?-decorated actions are either followed by a !-decorated one, by a $\surd$ or by nothing in case of divergence of protected code. For all A+I traces of securely compiled J+E components, if a ?-decorated action $\gamma?$ is followed by a !-decorated one, then there must exists a source-level component that can replicate $\gamma?$. Otherwise, if a ?-decorated action is followed by a $\surd$, then a check inserted by the compiler has been triggered and the execution has been stopped. A source-level component that can replicate $\gamma?$ does not exist in this case. This is the only case when securely-compiled components stop (i.e., they execute `halt`), thus they cannot contain `exit` statement. If they could, ambiguity would arise for when a $\surd$ is encountered in traces, as $\surd$ could be generated from a triggered check or from the execution of `exit`. Since we need to be able to distinguish all cases when a check has been triggered, no `exit` statement can be found in method bodies.

For the algorithm to be applicable, all ?-decorated actions considered in the distinguishing traces must correspond to actions that a source-level J+E component can replicate. Replicating a target-level action at the source level has been described as part of the algorithm; this means generating J+E code that performs a method call or a return (based on the encountered label), with the source-level parameters corresponding to those found in the trace. The inability to replicate a target-level action at the source level corresponds to creating a source-level component that is ill-typed. If all ?-decorated actions can be replicated at the source level, the algorithm is sure to create a witness that can replicate all actions in the considered traces.

Theorem 5 (Applicability of the Algorithm) *Consider a* J+E *component C. For any trace $\overline{\alpha} \in \mathsf{Traces}^{\mathsf{S}}_{\mathsf{A+I}}(\llbracket C \rrbracket^{\mathsf{J+E}}_{\mathsf{A+I}})$, $\forall i \in \mathbb{N}.\overline{\alpha}(i) = \gamma?$, if $\gamma?$ cannot be replicated in* J+E*, then $\overline{\alpha}(i+1) = \surd$.*

Proof of Theorem 5. The proof proceeds by induction on i, then by case analysis on $\gamma?$. For each case of $\gamma?$, a number of cases arise when considering which actions A+I code can do that are not replicable in J+E. Only the case analysis for the base case is presented. In fact, the only difference between the base and the inductive case is that in the latter, dynamically created objects can be communicated in the trace via the actions up to index $i-1$.

Case analysis on $\gamma?$:

`call` $a\ \overline{v}?$ In this case, a can be either the throw entry point or the entry point for a method m.

Throw entry point. By analysing $\overline{v}$, the following cases arise for when the action could not be replicated in J+E:

- An exception could not be thrown;
- An internally-typed, ill-typed exception was thrown;
- A non-existent internally-typed object was thrown;
- Something that was not an object was thrown.

The secure compiler inserts checks that capture all of the aforementioned faults and termination is triggered when they arise.

Method m entry point. The following cases arise by analysing $\overline{v}$:

- Ill-typed current object;
- Non-existent current object;
- Ill-typed, internally-typed parameter;
- Non-existent, internally-typed parameter;
- Ill-typed, primitively-typed parameter.

The secure compiler inserts checks that capture all of the aforementioned faults and termination is triggered when they arise.

One could think that there is a fault missing from this list, namely that A+I code could pass a *non-existent, externally-typed parameter*. However, this is not a fault because given an externally-typed parameter, there always exists a context that has an object allocated for that type. This, together with the fact that interfaces are compatible in J+E programs, allows us to conclude that this is not a fault, as an externally-typed parameters can always be created.

`ret` a v? In this case a is the return entry point.

Return entry point . The following cases arise:

Wrong returned value In this case, v could be: (i) an ill-typed, internally-typed parameter; (ii) a non-existent, internally-typed parameter, (iii) a non-existent, externally-typed parameter or (iv) an ill-typed, primitively-typed parameter. All these cases have been debated before.

Non-existent return In this case, no return is possible because no outcall was made. The compiler checks that an outcall was made when jumps to the returnback entry point are made, and if it is not the case, the execution is terminated.

□

7.3 Theorem Statements and Proofs

This section presents the theorem statements that the compiler of Section 6.2 is secure and their proofs.

Property 1 (Correct compilation) *To be correct, a compiler can be proven to preserve and reflect contextual equivalence at the source level with well-behaved contextual equivalence at the target level. Formally, for a correct compiler, the following holds:* $C_1 \simeq^{\mathcal{S}} C_2 \iff [\![C_1]\!]^{\mathcal{S}}_{\mathcal{T}} \overset{w}{\simeq}{}^{\mathcal{T}}_{\mathcal{S}} [\![C_2]\!]^{\mathcal{S}}_{\mathcal{T}}$.

Theorem 6 (Compiler preserves behaviour) *Given that the secure compiler is a correct compiler extended with checks, the secure compiler outputs target-level programs that behave as the corresponding input ones. Formally:* $[\![C_1]\!]^{\mathsf{J+E}}_{\mathsf{A+I}} \simeq^{\mathsf{A+I}} [\![C_2]\!]^{\mathsf{J+E}}_{\mathsf{A+I}} \Rightarrow C_1 \simeq^{\mathsf{J+E}} C_2$.

Proof of Theorem 6. As stated in Section 6.2, for the development of the secure compiler we start from a correct one and we extend it with checks. The additional checks inserted by the compiler do not modify the behaviour of compiled code, so the secure compiler $[\![\cdot]\!]^{\mathsf{J+E}}_{\mathsf{A+I}}$ satisfies Property 1, so the following holds: $C_1 \simeq^{\mathsf{J+E}} C_2 \iff [\![C_1]\!]^{\mathsf{J+E}}_{\mathsf{A+I}} \overset{\mathrm{w}}{\simeq}{}^{\mathsf{A+I}}_{\mathsf{J+E}} [\![C_2]\!]^{\mathsf{J+E}}_{\mathsf{A+I}}$. From this equivalence, we can consider only the $\Leftarrow$ direction: $[\![C_1]\!]^{\mathsf{J+E}}_{\mathsf{A+I}} \overset{\mathrm{w}}{\simeq}{}^{\mathsf{A+I}}_{\mathsf{J+E}} [\![C_2]\!]^{\mathsf{J+E}}_{\mathsf{A+I}} \Rightarrow C_1 \simeq^{\mathsf{J+E}} C_2$. Since the set of well-behaved contexts is a subset of the set of contexts, we have that $[\![C_1]\!]^{\mathsf{J+E}}_{\mathsf{A+I}} \simeq^{\mathsf{A+I}} [\![C_2]\!]^{\mathsf{J+E}}_{\mathsf{A+I}} \Rightarrow [\![C_1]\!]^{\mathsf{J+E}}_{\mathsf{A+I}} \simeq^{\mathsf{A+I}} [\![C_2]\!]^{\mathsf{J+E}}_{\mathsf{A+I}}$. Thus the proof of this theorem follows as a corollary of compiler correctness. □

Theorem 7 (Algorithm correctness) *For any two source-level components* C_1 *and* C_2 *that, once compiled exhibit a different trace semantics, the algorithm of Section 7.1 outputs a component* C *that differentiates between* C_1 *and* C_2 *(assuming there is no overflow of the secure stack and of the secure heap). Formally:* $[\![C_1]\!]^{\mathsf{J+E}}_{\mathsf{A+I}} \overset{T}{\not\simeq}{}^{\mathsf{A+I}} [\![C_2]\!]^{\mathsf{J+E}}_{\mathsf{A+I}} \Rightarrow C_1 \not\simeq^{\mathsf{J+E}} C_2$.

In the following, indicate the i-th action of a trace $\overline{\alpha}$ as $\alpha^{(i)}$.

Proof of Theorem 7. Trace semantics deals with *sets* of traces, while the algorithm inputs *single* traces. Moreover, these single traces must be the same up to a !-decorated action. The two different single traces are obtained as follows. Since $[\![C_1]\!]^{\mathsf{J+E}}_{\mathsf{A+I}} \overset{T}{\not\simeq}{}^{\mathsf{A+I}} [\![C_2]\!]^{\mathsf{J+E}}_{\mathsf{A+I}}$, we have that $\mathsf{Traces}_{\mathsf{A+I}}([\![C_1]\!]^{\mathsf{J+E}}_{\mathsf{A+I}}) \neq \mathsf{Traces}_{\mathsf{A+I}}([\![C_2]\!]^{\mathsf{J+E}}_{\mathsf{A+I}})$, thus there exists a trace $\overline{\alpha}$ that belongs to either only $\mathsf{Traces}_{\mathsf{A+I}}([\![C_1]\!]^{\mathsf{J+E}}_{\mathsf{A+I}})$ or only $\mathsf{Traces}_{\mathsf{A+I}}([\![C_2]\!]^{\mathsf{J+E}}_{\mathsf{A+I}})$ but not to both. Assume wlog that $\overline{\alpha} \in \mathsf{Traces}_{\mathsf{A+I}}([\![C_1]\!]^{\mathsf{J+E}}_{\mathsf{A+I}})$.

The trace $\overline{\alpha}$ can be split in two parts $\overline{\alpha_s}$ and $\overline{\alpha_d}$ such that $\overline{\alpha}$=$\overline{\alpha_s\alpha_d}$ and such that $\overline{\alpha_s}$ is the longest prefix of all traces of $\overline{\alpha_2}$. So, there exists a trace $\overline{\alpha'} \in \mathsf{Traces}_{\mathsf{A+I}}([\![C_2]\!]^{\mathsf{J+E}}_{\mathsf{A+I}})$ that can be split in two parts $\overline{\alpha_s}$ and $\overline{\alpha'_d}$ such that $\overline{\alpha'}$=$\overline{\alpha_s\alpha'_d}$ and $\overline{\alpha_d} \neq \overline{\alpha'_d}$. Additionally, $\overline{\alpha_2}$ must have even length, so that the different action in the traces is !-decorated and thus generated by either C_1 or C_2. Trace $\overline{\alpha'}$ always exists, it could be an empty trace, it could be composed by an empty $\overline{\alpha_s}$ and, possibly, by an empty $\overline{\alpha'_d}$. The traces input for the algorithm are $\overline{\alpha_1} = \overline{\alpha_s\alpha_d}$ and $\overline{\alpha_2} = \overline{\alpha_s\alpha'_d}$.

Due to Theorem 5, the algorithm can be applied. There may be a case when the differentiating traces contain a $\surd$ in a case but not in the other. In this case, the witness can be created, but it will not typecheck against the component whose traces generate the $\surd$. The differentiation in this case is thus achieved by detecting this case as the terminating one, while the algorithm must diverge in the other case.

The proof analyses all possible differences in $[\![C_1]\!]^{\mathsf{J+E}}_{\mathsf{A+I}} \not\overset{T}{=}{}^{\mathsf{A+I}} [\![C_2]\!]^{\mathsf{J+E}}_{\mathsf{A+I}}$ and proves that the output of the algorithm differentiates between C_1 and C_2, so $C_1 \not\simeq^{\mathsf{J+E}} C_2$. For each possible difference, the proof refers to an example from Section 7.1 that generate the context capable of performing the differentiation.

- traces of different length (Example 29);
- different kind of actions (Example 28);
- same kind of actions with differences in their structure:
  - return action
    - different primitive-typed value (Example 22);
    - different internally-typed value (Example 23);
    - different externally-typed value (Example 27);
  - call action
    - different callee (Example 25 and 26);
    - different primitive-typed parameter (Example 22);
    - different internally-typed parameter (Example 23);
    - different externally-typed parameter (Example 27);
    - different method (Example 24).

As those listed above are the only differences that can appear in two different traces, and since they all present a counterexample that reaches the contradiction, the theorem holds. □

Theorem 8 (Full abstraction of the compilation scheme) *For any two source-level components C_1 and C_2, we have: $C_1 \simeq^{\mathsf{J+E}} C_2 \iff [\![C_1]\!]^{\mathsf{J+E}}_{\mathsf{A+I}} \simeq^{\mathsf{A+I}} [\![C_2]\!]^{\mathsf{J+E}}_{\mathsf{A+I}}$.*

Proof of Theorem 8. The equivalence is split into two subgoals. The direction $\Leftarrow$ holds due to Theorem 6. The direction $\Rightarrow$ is reversed to the equivalent statement: $[\![C_1]\!]^{\mathsf{J+E}}_{\mathsf{A+I}} \not\simeq^{\mathsf{A+I}} [\![C_2]\!]^{\mathsf{J+E}}_{\mathsf{A+I}} \Rightarrow C_1 \not\simeq^{\mathsf{J+E}} C_2$. Apply Theorem 4 to restate the statement as $[\![C_1]\!]^{\mathsf{J+E}}_{\mathsf{A+I}} \not\overset{T}{=}^{\mathsf{A+I}} [\![C_2]\!]^{\mathsf{J+E}}_{\mathsf{A+I}} \Rightarrow C_1 \not\simeq^{\mathsf{J+E}} C_2$. Apply Theorem 7 to prove the statement. □

Chapter 8

A Secure Compiler for Multi-Principal Languages

"I dislike him." - Why? - "I'm no match for him." - Has a human being ever answered in this way?

Friedrich Nietzsche - Beyond Good and Evil Part IV - Aphorism 185

The secure compiler of Section 6.2 considered code written by a single programmer, a single principal who runs his code on a certain machine that is susceptible to attacks. However, it is often the case that code of different programmers are executed alongside on the same machine. Many high-level languages in fact have module systems (à la ML), package systems (à la Java) or mixins (called traits in Scala) that allow source-level programmers to define interaction between their different programs.

Consider for example two clients (called Alice and Eve) of the same network library (called Network), operating on the same machine. The clients are likely not to trust each other, yet they do trust the network library. How should the whole code base be compartmentalised in a low-level target language? Placing the library code in the same compartment of one client code will either generate redundancy of code or violate the other client's trust assumption.

To address this concern, each code base needs to be separated and the interoperation between each code base must be secured. As previously seen, a technique to achieve this result is secure compilation. However, applying known secure (fully abstract) compilation schemes to this setting results in insecure, exploitable code (Problem 13).

Problem 13 (Unknown object guessing) *Consider an implementation of a class* `Alice` *(lines 1 to 6) with a single method* `create` *that calls method* `send` *on object* `net` *(line 4), which is implemented by the trusted library. Function* `send` *is called with a newly allocated* `Alice` *object* `n` *as parameter (line 3).*

```
1 class Alice implements Client {   // code of a client
2  public create( ) : Int {
3    var n = new Alice( );          // allocate a new Client object
4    return net.send( n );          // call the code of the trusted library
5  }
6 }
```

The code snippet above only calls the trusted library. With the following implementation of `send` *that just returns 0 without ever leaking* `arg` *to other code, the communicated object* `n` *is* confidential.

```
1 public send( arg : Client ) : Int { return 0; }  // trusted code
```

However, by securely compiling the `Alice` *class with known techniques, the confidentiality of* `n` *can be violated. A malicious attacker operating at the target-language level can call method* `create` *on an object of type* `Alice`*, and it will create object* `n` *and call method* `send`*. Once both methods return, the control is given back to the attacker who can just "guess" the target-level object id of* `n` *and violate the confidentiality property.*[1]

When proving a compiler to be fully abstract, these security violations must be detected. Since the communicated argument is confidential, the `Alice` *class above must be indistinguishable (once compiled) from the* `Alice` *class below, whose only difference is the argument of* `send` *(*`this` *in place of* `n`*, line 4).*

```
1 class Alice implements Client {  // code of a retailer
2  public create( ) : Int {
3                                  // this version does not create a new
                                        object
4    return net.send( this );      // communicate this
5  }
6 }
```

[1]This violation does not happen if the target language uses address space layout randomisation (ASLR) [6, 62]. Nevertheless, ASLR-compiled code is subject to other multi-principal-related violations described in the remainder of this thesis.

Informally, the current definition of fully abstract compilation is concerned with two parties: the code to be secured and the attacker to that code. If trusted code is considered part of the former, the trust assumption of code of other principals is violated. If trusted code is considered part of the latter, the trust assumption of the code to be secured is violated. Thus the definition of fully abstract compilation must be changed and it must account for trusted code explicitly. ■

The goal of this chapter is to provide a secure (fully abstract) compilation scheme for multi-principal, object oriented languages that allows a programmer to write programs like that of Problem 13 that are secure.

Firstly, this chapter informally presents the changes to *PMA* and A+I to support multiple protected modules and changes to J+E to support principal annotations (Section 8.1). Then, it defines the security violations that arise when multi-principal code is considered (Section 8.2). The core of this chapter is the secure compilation scheme for multi-principal code (Section 8.3). The target and the source language are then formalised (Section 8.4 and Section 8.5, respectively) before arguing that the compilation scheme of Section 8.3 is secure (Section 8.6).

8.1 Language Changes

This section firstly presents *PMAS*, which is an extension of *PMA* from Section 2.1 with multiple protected modules (Section 8.1.3). Then it informally describes AIM, the extension of A+I with multiple protected modules (Section 8.1.1) and JEM, the extension of J+E with principal annotations (Section 8.1.2). Finally, it presents the threat model considered for this chapter (Section 8.1.4) and the definition of multi-principal full abstraction, which is used to prove the compilation scheme secure (Section 8.1.5).

8.1.1 *PMAS*: *PMA* with Multiple Protected Modules

PMAS adds multiple protected modules to *PMA*, where a single protected module was considered. Figure 8.1 contains a graphical representation of how is the memory affected by the adoption of *PMAS*.

All previously presented *PMA* instances (e.g., Fides, the Intel SGX etc) support multiple modules but, for the sake of simplicity, they were not considered. The *PMAS* architecture introduces no new concepts, it simply extends the access

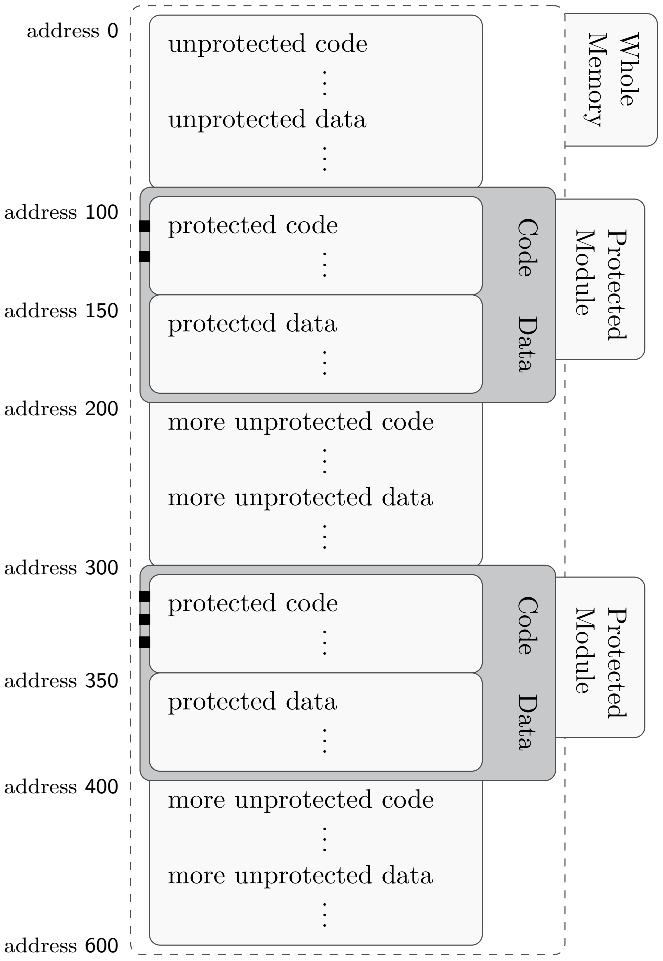

Figure 8.1: Graphical representation of the *PMAS* architecture.

control policy of *PMA* to consider inter-module access. Table 8.1 summarises the *PMAS* access control model.

8.1.2 A Target Language with *PMAS*: Aim

This chapter adopts Aim (acronym of Assembly plus Isolation and Modules) as target language for the secure compiler. Aim extends the A+I language of Chapter 4 with support for *PMAS*.

Table 8.1: Access control policy of *PMAS*.

From\ To	Protected (same Id)			Protected (different id)			Unprotected
	E. Point	Code	Data	E. Point	Code	Data	
Protected	r x	r x	r w	x			r w x
Unprotected	x			x			r w x

Examples 30 to 31 provide code examples of how Aim works.

Example 30 (Aim code) *Consider a protected module spanning from address 100 to 200 with two entry points at address 100 and 106 and another protected module spanning from address 300 to 400 with a single entry point at address 300. The code snippet below shows some* Aim *instructions (preceded by the address where they are located) and how code from different modules interacts with each other.*

```
0    movi r0 12     // unprotected code. Load value 12 in register r0
1    movi r1 10     // load value 10 in register r1
2    movi r5 100    // load the address of the entry point of protected
     code
3    call r5        // call the first protected module by jumping there
...
100  sub  r0 r1     // protected module. Calculate the subtraction r0 − r1
101  movi r3 104
102  jl   r3        // check if the 'less-than-zero' flag was set (by sub)
103  ret            // if not, return the value of the subtraction (in r0)
104  movi r0 300    // otherwise the flag was set. In this case
105  call r0        // first call another protected module
106  ret            // and then return what that code returns
...
300  movi r1 350    // another protected module
301  movs r1 r0     // writes in its own protected memory (address 350)
302  movi r0 0
303  ret            // and returns 0
```

⊡

Example 31 (Violations of the *PMAS* access control policy) *Consider the same memory partition of Example 30.*

```
1 0    movi r0 101    // unprotected code. Load value 101 in r0
2 1    jmp  r0        // jump to address 101
```

Since address 101 is not an entry point of the protected module, the jump of protected code (line 2) does not succeed.

```
1 0    movi r0 101    // unprotected code. Load value 101 in r0
2 1    movi r1 20     // load value 20 in r1
3 2    movs r0 r1     // write 20 at address 101
```

Since unprotected code cannot write (nor read) inside protected modules, the write at address 2 (line 3) fails. Analogously, if the instruction of line 2 were replaced with `movl` r_0 r_1 *(i.e., a read to address 101), that would fail as well since unprotected code does not have that privilege. If these code snippets were placed in a different protected module (e.g., at address 300) instead of in unprotected memory, the same violations would be detected. Any violation of the access control policy of PMAS is trapped by the architecture and the execution is halted.* ⊡

8.1.3 A Source Language with Principal Annotations: Jem

Jem extends the J+E language of Chapter 3 with explicit principal annotations to study how principals affect secure compilation. In most component-based languages, component information is implicitly used to identify the principal that wrote that component. In Jem, principal annotations and trust statement are explicit in order to reason about them. Principal annotations are used to identify the entity that provides a software component. Trust statements define code whose behaviour is not supposed to deviate from its Jem specification once compiled.

To provide a better understanding of the language, Listing 8.1 presents the complete definition of Alice's code from Problem 13.

8.1.4 The New Threat Model

The threat model of this chapter consists of: the system under attack, the security properties that the system must uphold, the attacker to the system and how are security properties enforced with a secure compiler. This threat model extends the one of Section 2.4 to multi-principal systems.

The system under attack is a von Neumann machine with a flat address space and *multiple PMA*-enforced protected modules. The system contains *protected*, *trusted* and *unprotected* code. Protected and trusted code reside within two different protected modules, unprotected code resides both in unprotected memory and within other protected modules. To refer to trusted code or unprotected code alike, the term *external code* is used.

```
package library of principal Network {  // package defining library code
  interface Network {
    public send( arg : client.Client ) : Int;
  }
  extern net : Network;      // static object implementing Network methods
}

package client of principal Alice { // interface exposed by Alice's code
  interface Client {
    public create( ) : Int;
  }
  extern alice : Client;            // static object provided by Alice
}

package alice of principal Alice {   // Alice's code implementation
  trust principal Network;           // explicit trust statment
  class Alice implements client.Client {
    ··· // omitted, see Problem 13
  }
  object alice : Alice;     // implementation of the homonymous extern
}
```

Listing 8.1: Complete definition of Alice's code.

The security property that the system must uphold is that protected and trusted code behave like Jem components and in no other way. As in Section 2.4, this property entails: (i) confidentiality and integrity of fields, of object names and of method bodies; (ii) no control flow alteration apart from method calls, returns and exception usage; (iii) non reachability of stuck (error) program states.

The attacker does not change from the one defined in Section 2.4.

To relate these concepts to Problem 13, `Alice`'s code is the protected code, the `Network` implementation is the trusted code and code from other clients is the unprotected code and the omitted `Eve` implementation is the attacker. The security property entails that the newly created object `n` is confidential, inaccessible from Eve.

Since the definition of fully abstract compilation does not permit the differentiation between protected and trusted code, this chapter defines multi-principal fully abstract compilation (Section 8.1.5). If a compilation scheme is proven to enjoy that property, the output it produces behaves *only* like its source level counterpart, satisfying the property that the system must uphold.

8.1.5 Multi-Principal Full Abstraction

Definition 13 presented what a fully abstract compilation scheme is. When multi-principal program are considered, this notion is no longer sufficient, since it has no way of specifying what trusted code is. In fact, when relating this definition to the elements of the threat model, we can see that C_1 and C_2 are the protected code and contexts are the unprotected code [6,62,94], but trusted code is missing. If trusted code is part of the context, it can behave arbitrarily once compiled and this violates the security property that the system must uphold. Trusted code is assumed to behave as its source-level counterpart and the definition of fully abstract compilation scheme must reflect this. Trusted code can also not be considered as part of the protected code, as this could violate possible trust assumptions from other principals. If Alice and Eve mistrust each other but trust the Network, incorporating the Network with either principal will violate the trust assumption of the other.

We could change Definition 13 to reason about lists of components $\overline{C_1}$ and $\overline{C_2}$, so that a list of component contains the trusted ones as well. However, consider $\overline{C_1}$ and $\overline{C_2}$ to both contain the code of Alice and of the Network but $\overline{C_2}$ also contains C^C, a component with no public methods that no code can interact with. Intuitively, these two lists of components are equivalent at the source level since C^C cannot be interacted with. However, its presence at the target level (C^C occupies its own module) highlights a difference between $\overline{C_1}$ and $\overline{C_2}$ that causes a failure of full abstraction.

Following is a definition of secure compilation scheme that overcomes these limitations (Definition 38). It states that a compilation scheme is secure if and only if it preserves and reflects contextual equivalence of two programs, for all possible trusted program C^T they link against (denoted as $C + C^T$).

Definition 38 (Multi-principal full abstraction of a compilation scheme) *A compilation scheme* $[\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}} : \mathcal{S} \to \mathcal{T}$ *is multi-principal fully abstract if:* $\forall C_1,\ C_2,\ C^T \in \mathcal{S}$. *if* C_1 *trusts* C^T *and* C_2 *trusts* C^T *then* $(C_1 + C^T) \simeq^{\mathcal{S}} (C_2 + C^T) \iff [\![(C_1 + C^T)]\!]^{\mathcal{S}}_{\mathcal{T}} \simeq^{\mathcal{T}} [\![(C_2 + C^T)]\!]^{\mathcal{S}}_{\mathcal{T}}$.

C^T cannot be existentially quantified because if it were, existing secure compilation schemes could be proven to be secure while they are subject to security violations (as Section 8.2 discusses).

This definition scales to lists of trusted components $\overline{C^T}$, but this is not considered for the sake of simplicity. With lists of trusted components, more trusted components can be considered. In this scenario, assume the trust relation is transitive, i.e., if a trusted component C^T trusts another component C^S, C^S

is securely compiled with C_1, C_2 and C^T. Future work will investigate the relationship between intransitive trust statements and secure compilation.

Proving Multi-Principal Full Abstraction

To prove a compilation scheme secure, the same strategy of Chapter 7 can be followed. The co-implication of Definition 38 is split in two cases: $\Leftarrow$ and $\Rightarrow$, one for each of its directions.

Direction $[\![C_1 + C^T]\!]^{\mathcal{S}}_{\mathcal{T}} \simeq^{\mathcal{T}} [\![C_2 + C^T]\!]^{\mathcal{S}}_{\mathcal{T}} \Rightarrow (C_1 + C^T) \simeq^{\mathcal{S}} (C_2 + C^T)$ states that the compiler outputs target-level programs that behave as their source-level counterparts.

Direction $(C_1 + C^T) \simeq^{\mathcal{S}} (C_2 + C^T) \Rightarrow [\![C_1 + C^T]\!]^{\mathcal{S}}_{\mathcal{T}} \simeq^{\mathcal{T}} [\![C_2 + C^T]\!]^{\mathcal{S}}_{\mathcal{T}}$ states that source-level abstractions are preserved through compilation to the target level. To avoid working with target-level contexts, we replace the notion of contextual equivalence at the target level ($\simeq^{\mathcal{T}}$), with that of trace equivalence ($\overset{\mathrm{T}}{=}^{\mathcal{T}}$). This notion is used to prove the contrapositive of the current direction: $[\![C_1 + C^T]\!]^{\mathcal{S}}_{\mathcal{T}} \overset{\mathrm{T}}{\neq}^{\mathcal{T}} [\![C_2 + C^T]\!]^{\mathcal{S}}_{\mathcal{T}} \Rightarrow (C_1 + C^T) \not\simeq^{\mathcal{S}} (C_2 + C^T)$. To prove that $C_1 + C^T$ and $C_2 + C^T$ are not contextually equivalent, it suffices to show that *there exists* a source-level context that behaves differently depending on whether its hole is filled with $C_1 + C^T$ or $C_2 + C^T$. Such a context is said to *differentiate* $C_1 + C^T$ from $C_2 + C^T$.

As before, when the witness cannot be generated, failures of multi-principal full abstraction arise, and these failures (often) lead to security violations. Consider the code of Problem 13; let n denote the target-level object id of the confidential object `n` and c denote the offset of method `create`. The behaviour of the two code snippets can be represented with the following traces:

$$\begin{aligned} \textsf{First case} &= \texttt{call}\ c\ \textsf{on}\ n?\ \texttt{return}\ 0! \\ \textsf{Second case} &= \texttt{call}\ c\ \textsf{on}\ n?\ \surd \end{aligned}$$

In fact, while a call on `n` succeeds in the first case, it does not in the latter, since `n` does not exist there (assume in the latter case the execution stops as indicated by $\surd$). However, generating a source-level component from these traces is not possible, since at the source level, unprotected code does not know name `n` and thus cannot call methods on it. For the compilation scheme to be secure, securely-compiled programs must not generate this kind of traces. The next section will highlight other similar security violations that arise due to the presence of multiple principals.

8.2 Multi-Principal Related Security Violations

This section describes the security pitfalls that a naïve implementation of a compiler for multi-principal languages must avoid (Problems 14 to 19). Each problem highlights a security vulnerability and how it makes multi-principal full abstraction not provable for a compilation scheme. In the following, code snippets are extensions of the ones presented in Section 8.1.3; when new methods are defined assume their signature appears in the `Client` interface. Each problem is followed by a brief, high-level overview of the solution that addresses it. The secure compilation scheme of Section 8.3 will provide more detailed, implementable solutions to these problems.

Problem 14 (Principal information at the target level) *Consider method `allocate` (lines 1-3) that returns new objects of type `Client` (line 2). Method `proxyAlloc` (lines 4-11) calls method `allocate` (line 6) and forwards the returned value in case it is not **`null`**, otherwise it returns a new `Bob` object.*

```
 1 class Alice implements Client{                   // protected code
 2   public allocate( ) : Client { return new Alice( ); }
 3 }
 4 class Bob implements Client{                    // unprotected code
 5   public proxyAlloc( ) : Client {
 6     var nl = alice.allocate( );
 7     if ( nl == null )
 8       return new Bob( );
 9     return nl;
10   }
11 }
```

When `proxyAlloc` returns at the target level, it returns a target-level object id. Since the code interacting with `proxyAlloc` does not know the principal related to the returned object, dispatching of methods called on that object can be ambiguous. In fact, the known type of the returned object is `Client`, but both principals provide code that implements that interface. ■

To address this concern, target-level object ids must indicate their principal.

Problem 15 (Unknown structure guessing) *As mentioned in Problem 13, target-level object ids can be guessed by unprotected code.*

A naïve solution to this problem is keeping track of which principal has received a certain object id. In Problem 13, protected code could keep track that trusted code has received the object id of `n`. With this solution a guess from unprotected code would be detected and stopped. However, this solution also prevents *unprotected code from using that id altogether. Consider the following, different implementation of* `send` *(renamed* `sendLeak` *to avoid confusion) that forwards its argument to Eve's unprotected code. Eve is implementing a logging functionality that the trusted code uses.*

```
// trusted code
public sendLeak ( arg : Client ) : Int { return eve.log( arg ); }
```

With the aforementioned solution, unprotected code would not be able to call methods on object `n`*. In fact, protected code would have registered that trusted code has received* `n` *and it would not accept it once Eve's code uses it.* ■

To address this concern, target-level object id must be unforgeable: if a principal supplies an object id it is because it has received it, not guessed it.

Problem 16 (Different target-level object id) *Consider two implementations of the method* `allocate`*. The left-hand side one allocates a new object* `new1` *and sends it to trusted code using method* `send`*, then it allocates a new object* `new2` *and sends it to unprotected code over method* `log`*. The right-hand side implementation does the same but in the reverse order.*

```
// protected code
public allocate( ) : Client {
  var new1 = new Alice( );
  net.send( new1 );
  var new2 = new Alice( );
  eve.log( new2 );
  return new1;
}
```

```
// protected code
public allocate( ) : Client {
  var new2 = new Alice( );
  eve.log( new2 );
  var new1 = new Alice( );
  net.send( new1 );
  return new1;
}
```

These code snippets are equivalent at the source level (when the `send` *implementation of Problem 13 is considered). However, at the target level, object* `new1` *will have a certain id in the left-hand side snippet and a different one in the right-hand side one. Assuming the allocation strategy is deterministic, in the former case* `new1` *will be placed at a certain location while in the latter case,* `new2` *will be placed there. This would generate two target-level traces with different values for the same target-level object id that the source-level differentiating component could not tell apart.*

The same security violations of Problem 13 happen (i.e., confidentiality violation), though the violation is revealed by a different failure of the multi-principal full abstraction proof. ■

To address this concern, the compiler must ensure that objects communicated to different principals do not affect each others' target-level id.

Problem 17 (Call stack shortcutting) *Consider two implementations of method `add` (lines 2 to 7) that call function `send` (line 3). If `send` returns the same value as the supplied argument (`elem` on the left-hand side or `this` on the right-hand side,* 1 *is returned (line 5), otherwise,* 0 *is returned (line 6).*

```
// protected code
public add( elem : Client ) : Int {
  var tmp = net.send( elem );
  if ( tmp == elem )
    return 1;
  return 0;
}
```

```
// protected code
public add( elem : Client ) : Int {
  var tmp = net.send( this );
  if ( tmp == this )
    return 1;
  return 0;
}
```

Method `send` calls `log`, which is in unprotected code, then returns `arg`.

```
// trusted code
public send( arg : Int ) : Int { eve.log( null ); return arg; }
```

These code snippets are equivalent at the source level, the else branch of line 6 is never taken, so both `add` implementations return 1. However, once their compiled counterparts are considered, they are not equivalent. At the target level, the address where to return can be read from the registers file and this causes the security violation. Once control is returned to unprotected code (in method `log`), that code can use a previously-read address to perform a call shortcut.

Figure 8.2 provides a pictographic representation of call stack shortcutting. The

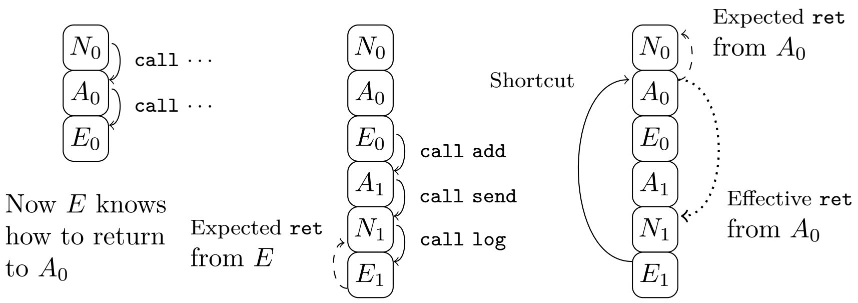

Figure 8.2: A shortcuttable call stack. The stack grows down.

figure presents the view of the stack during a particular execution of a program, where code from the `Network` called code from `Alice` that called code from `Eve` (left-hand side of the fig.). Then, `Eve` calls to function `add`, presented in the snippets above, and the stack grows accordingly: `Alice`'s `add` calls `Network`'s `send` that calls `Eve`'s `log` (central part of the fig.). Allocation records are marked with the principal they belong to: E *for unprotected code (`Eve`),* N *for trusted* *code (`Network`), A for protected code (`Alice`). Records N_0, A_0 and E_0 belong to omitted code whose implementation is not relevant. In the function corresponding to allocation record E_0, unprotected code can read the address where to return to A_0 from the registers file. Then, in the function corresponding to allocation record E_1 (`Eve`'s `log`), unprotected code can use that address and jump to the previously-read address (right-hand side of the fig.). Protected code will then execute the return code related to A_1, assuming that it will return to E_0 but when popping the return value from the stack, execution will jump to the function related to N_1.*

Applying call-stack shortcutting to the code snippets above, can make the left-hand side return 0 and the right-hand side return 1 at the target level as follows. External code can call method **`add`** *on an object o with an argument different from o, then it can shortcut the stack and supply the id of o as the return value. These differences can be captured by target-level traces, though no source-level component can cause the same behaviour.* Jem *components cannot in fact cause an unnatural traversal of the stack; even when using exceptions, the stack is traversed from top to bottom. Note that in fact also exceptions can be used to trigger stack shortcutting.*

Call stack shortcutting can violate confidentiality properties as protected code can be made return to unprotected code in place of trusted code, leaking confidential information as in Problem 13. ■

To address this concern, unnatural traversals of the stack must not happen.

Problem 18 (Types of other parties' parameters) *Consider two functions `fw` that call a function `send` on an parameter `arg` of type `Network` (line 3) that is defined in trusted code. The two implementations always return 0 (line 5), though right-hand implementation checks that the value returned by `send` is equal to* 0*.*

```
1 // protected code
2 public fw( arg : Network ) : Int {
3   var n = arg.send( this );
4
5   return n;
6
7 }
```

```
1 // protected code
2 public fw( arg : Network ) : Int {
3   var n = arg.send( this );
4    if (n == 0)
5     return n;
6   return 0;
7 }
```

In addition to the implementation of Problem 13, trusted code provides a class `Connection` implementing method `test` that always returns 1.

```
1 // trusted code
2 class Connection { public test( arg : Client ) : Int { return 1; } }
```

The two protected code snippets are equivalent at the source level but they are not at the target level. Unprotected code can call `fw` *supplying the index of an object of type* `Connection` *in place of an object of type* `Network`*. Protected code has no way to tell the difference between the two at the target level because there is no type information there, so it will call function* `test`*, since functions are accessed by offset. This causes two different target-level traces for the code snippets above. In fact, the left-hand side snippet returns* 1*, (the value returned by* `test`*) while the right-hand side always returns* 0*. This differentiation is not possible at the source level, where only well-typed code is executed since passing an object of type* `Connection` *in place of an object of type* `Network` *is not possible.*

This vulnerability can lead to violation of invariant properties, causing protected code to call undesired functionality of trusted code. ■

To address this concern, protected code must check that the type of a received trusted object complies to the expected type.

Problem 19 (Existence of other parties' objects) *Analogously, to Problem 18, in place of an object of the wrong type, unprotected code can call* `fw` *by passing a non-existent trusted object id.*

Consider a different `fw` *implementation that firstly checks whether the passed argument is* **`null`***, in which case it returns 1 (line 3). Then it calls method* `send` *of the trusted code (line 5) and it returns 1 (line 9). In case an exception is raised, the right-hand side snippet returns 0 (line 7).*

```
 1 // protected code
 2 public fw( arg : Network ) : Int {
 3   if ( arg == null ) return 1;
 4   try{
 5     arg.send( this );
 6   } catch e : Obj {
 7
 8   }
 9   return 1;
10 }
```

```
 1 // protected code
 2 public fw( arg : Network ) : Int {
 3   if ( arg == null ) return 1;
 4   try{
 5     arg.send( this );
 6   } catch e : Obj {
 7     return 0;
 8   }
 9   return 1;
10 }
```

These snippets are equivalent at the source level: they always return 1. At the target level, unprotected code can call `fw` *supplying a fake object id of a trusted object that does not exist. This generates two different traces, since the left-hand side snippet returns 1 anyway while the right-hand side one returns 0. Unfortunately, this cannot be expressed at the source level, because the differentiating component cannot invent fake object ids.* ■

To address this concern, protected code must assess whether a trusted object exists or not before calling methods on it.

Finally, let us consider a different concern of the secure compiler: attestation. When linking different code pieces coming from different principals, the secure compiler must attest that the code has not been tampered with and that the expected principal is providing the expected code. To that extent, existing techniques such as proof carrying-code can be used [90]. Thus, target-level protection mechanism must provide code attestation functionalities, as some *PMA* implementations do [15, 112]. Since attestation falls beyond the scope of language translation (which is the subject of this thesis), this is not considered further.

8.3 The Secure Compiler

The code of each principal is compiled in a separate protected module as described in Section 6.2 with the following adjustment regarding how object ids are encoded (Section 8.3.1), how to address call stack shortcutting (Section 8.3.2) and how to perform checks on trusted objects (Section 8.3.3).

8.3.1 Encoding of Object Ids

To address Problem 13, 14, 15 and 16, the secure compiler must use a different format for target-level object ids. Currently, target-level object ids are either indexes in the masking table (for objects of the protected code), or addresses in memory (for objects of external code). With multiple securely-compiled modules, using the index in the masking table is not enough, as different objects in different modules can have the same id (Problem 14). Moreover, object ids must not be guessable (Problem 13). Target-level object ids now consist of three words: a module id, a masking index and a random number. They take three registers to be communicated. This can be optimised in architectures with sufficiently-long word sizes but we omit such optimisations for the sake of simplicity.

The module id is required so that object ids can uniquely identify an object in memory (Problem 14). For objects in unprotected memory, this value is 0.

To prevent unknown object id guessing (Problem 13), the object id includes a randomised nonce (also stored in $\mathcal{O}$). When an index is looked up in $\mathcal{O}$, the nonce must be supplied as well, if the wrong nonce is supplied, the check fails.

To ensure the same object always has the same masking index (Problem 16), a masking table for each principal interacting with the protected code is required. In the case of Problem 16, objects `new2` is placed in the masking table for principal `Eve`, who implements `log`). Object `new1` is then placed in the masking table for principal `Network`, who implements `send`. To know in which table to insert an object, protected code must know the module id of the code where it is jumping next. Thus, protected code must use *caller-callee authentication*, a feature provided by most *PMA* implementation that support multiple modules [15, 82, 91, 112]. With caller-callee authentication, the *PMA* implementation tags function calls with the id of the module that is doing the call (or who is returning from a call) by storing the module id in a predefined register. When protected code is returning from a function call, it uses the module id supplied at call time to know the masking table of which module to use. Otherwise if it is jumping to a function, protected code statically knows which module implements that function and that tells protected code the masking table to use. Should this static information not be known, most *PMA* implementations provide functionality that determine a module id given an address in the memory range of that module.

8.3.2 Preventing Call Stack Shortcutting

When additional entry points are created (e.g., the returnback entry point, the throw entry point, and so on) they must not allow call stack shortcutting (Problem 17). *PMA* is subject to call stack shortcutting because the stack is split among different modules. Method-related entry point are not subject to call stack shortcutting, since a global method can always be called. An entry point for a specific functionality must lead to executable code only if the specific functionality can be called. For example, an exception can only be thrown if the call stack is traversed in the right order, and the secure compiler must ensure this is fulfilled by adding extra checks.

Caller-callee authentication can also be used to address call stack shortcutting. The information regarding the caller module id must be stored in the secure stack any time a method of protected code is called. The compiler must insert a check at the returnback entry point to ensure that code returning there comes from the module whose id was stored in the secure stack.

Remark 2 (Alternative solution) *An alternative solution is to replace caller-callee module id with random number. When protected code calls methods of the external code, it supplies a random number, which is then expected at the returnback entry point. Only by guessing the random number can the unprotected code impersonate trusted code when retuning to protected code.*

8.3.3 Checks on Trusted Objects

To typecheck other objects in trusted code (Problem 18), the compiler creates an additional entry point in securely-compiled code. This entry point contains a function that takes two parameters: and object id and the bit representation of an interface type implemented by the component (so, a convention is set up to express interface types as the md5 has of their names). The function returns 1 if the object id is of the specified type and 0 otherwise (in most object-oriented languages this is the `instanceof` operator). This approach also addresses Problem 19, as that function returns 0 when called on non-existing objects.

To avoid violating full abstraction with the introduction of `instanceof`, (i) the source language must have the same functionality but, at the same time (ii) securely-compiled programs must not use it. Jem is extended with the `instanceof` operator but an additional check is performed on securely-compiled programs, to ensure that they do not use it. Point (i) is analogous to what Abadi discovered in an early JVM implementation [1] or what Kennedy found in compilers for the .NET platform [67]. In fact, target-level languages must not offer functionality that are not present in the source-level language. Concerning point (ii), if securely-compiled programs can use `instanceof`, the following source-level equivalent programs would be inequivalent at the target level. Consider two objects `eve` and `eve2` of type `Log`.

```
// protected code
public create( ) : Item {
  return instanceof( eve, Log );
}
```

```
// protected code
public create() ) : Item {
  return instanceof( eve2, Log );
}
```

These snippets always return `true` at the source level, thus they are equivalent. However, at the target level, external code implementing `Log` can detect a difference in the two snippets. Because the `Log` implementation of `instanceof` is called with two different arguments: `eve` and `eve2`. This difference, however, cannot be replicated at the source level, since `instanceof` is not a principal-implemented function but it is implemented by the runtime. To avoid this violation of full abstraction, securely compiled code cannot use `instanceof`.

Remark 3 (Types of trusted objects and catching exceptions) *The concerns of Problem 18 do not arise when catching exceptions is done based on interface types. In fact, as explained in Section 6.3.4, to support that, interface type information of all objects is required at the target level. However, that does not address Problem 19, i.e., the need to check that a supplied trusted object id* *exists and is not* **`null`***, for which a specific solution such as the one presented here is required.*

Remark 4 (Function parameters and register file size) *One last remark concerns the amount of function parameters and its relation to the registers file size. As a register file has finite size, parameters that exceed that size are spilled on the unprotected stack. However, when multiple modules are taken into account, this spilling is no longer possible. Spilling in unprotected memory would in fact leak the spilled data to unprotected code. Spilling on the stack of other modules is also not possible, since modules do not have access to each other's data section.*

Some PMA implementations [112] use chunks of unprotected memory for bulk-data communication between modules. The PMA implementation ensures that no other module can access the communicated data. Unfortunately, not all PMA implementations support this functionality, so Aim *does not have it.*

For function calls to have an unbounded number of parameters, source-level calls are split in a series of target-level ones, each with a reduced number of parameters. The first call jumps to the related entry point and passes only the first chunk of parameters. The callee stores these information and immediately returns in order to allow the caller perform other calls with the successive four parameters through the registers. The protocol continues until all parameters are passed; when the callee has all the parameters, it can perform the required checks. Any violation of the protocol, such as jumping to another entry point, is treated as a failed check. The only downside of this solution is its efficiency, as crossing module boundaries can be costly in certain PMA implementations.

Heving defined the compilation scheme, this chapter formalises both its target and source languages and argues that it is secure.

8.4 The Target Language Aim, Formally

As mentioned in Section 8.1.2, Aim is an extension of A+I with multiple protected modules. This section presents the syntax (Section 8.4.1) and the semantics (Section 8.4.2) of Aim. Then, it describes how to account for randomisation in the definition of contextual equivalence for Aim (Section 8.4.3). Finally, it presents trace semantics for Aim programs (Section 8.4.4).

$$\begin{array}{rll}
\textit{Memory descriptors} & s ::= (a_b, n_c, n_d, n, id) & \\
\textit{Programs} & P ::= (m, \overline{s}, O) & \\
\textit{Oracles} & O ::= w; O & \\
\textit{Principals} & id \in \mathcal{PR} \subset \textit{Words} & \\
\textit{Instructions} & \mathcal{I} + \texttt{rand}\ \texttt{r}_\texttt{d} & \text{Initialise } \texttt{r}_\texttt{d} \text{ with a random value.}
\end{array}$$

Figure 8.3: Elements of the Aim language. The instruction set $\mathcal{I}$ is defined in Figure 4.2

8.4.1 Syntax of Aim

Figure 8.3 presents additional elements of the language formalisation; missing elements can be found in Figure 4.1. Memory descriptors s are quintuples: (a_b, n_c, n_d, n, id) that formalise the access control policy of Section 8.1.2. In this case we do not partition unprotected code in a code and a data section since the trace semantics for Aim does not consider reading or writing to it. Programs P are pairs of a memory m and a sequence of memory descriptors $\overline{s}$, one for each protected module. Principals are taken from the set $\mathcal{PR}$, which is a subset of all possible words. Instructions are the same considered for A+I plus the instruction to randomly initialise the value of a register. The semantics of `rand r`$_\texttt{d}$ is to initialise register $\texttt{r}_\texttt{d}$ with a random value (Rule Eval-rand). To model the semantics of `rand`, we use an *oracle* [5], which provides random values for the randomisation function. An oracle O is an infinite stream values. Oracles affects Aim contexts $\mathbb{P}$, whose holes can now be filled by a pair consisting of a program and an oracle.

8.4.2 Semantics of Aim

The dynamic semantics of Aim relies on auxiliary functions to encode the access control policy of Section 8.1.2 (Figure 8.4). In the rules, p indicates program counter values (i.e., addresses in memory), a indicates an address in memory and s is a memory descriptor. Rules Aux-cross-un to Aux-cross-modules are the addition to the rules of the single module case. They tell when a jump across different domains is performed, where different domains can be different protected modules or unprotected memory.

The dynamic semantics of Aim is analogous to the semantics of A+I. The semantics considers states $\Omega ::= (p, r, f, m, \overline{s}, O)$ where p is the program counter,

$$\frac{s \equiv (a_b, n_c, n_d, n, id) \quad a_b \leq a \leq a_b + n + c + n_d}{s \vdash \mathsf{range}(a)} \text{(Aux-range)}$$

$$\frac{s = (a_b, n_c, n_d, n, id) \in \overline{s} \quad s \vdash \mathsf{range}(p)}{\overline{s} \vdash \mathsf{currentModule}(s, p, id)} \text{(Aux-current-module)}$$

$$\frac{\nexists s \in \overline{s}. s \vdash \mathsf{range}(p)}{\overline{s} \vdash \mathsf{unprotected}(p, un)} \text{(Aux-unprotected)}$$

$$\frac{s \vdash \mathsf{range}(p) \quad s \vdash \mathsf{range}(p') \quad s \vdash \mathsf{validJmp}(p, p')}{s \vdash \mathsf{intJmp}(p, p')} \text{(Aux-int-jmp)}$$

$$\frac{s \vdash \mathsf{unprotected}(p) \quad s \vdash \mathsf{unprotected}(p')}{s \vdash \mathsf{intJmp}(p, p')} \text{(Aux-ext-jmp)}$$

$$\frac{\overline{s} \vdash \mathsf{unprotected}(p, un) \quad \exists s \in \overline{s}. s \vdash \mathsf{range}(p')}{\overline{s} \vdash \mathsf{crossJmp}(p, p')} \text{(Aux-cross-un)}$$

$$\frac{\exists s \in \overline{s}. s \vdash \mathsf{range}(p) \quad \exists s' \in \overline{s}. s' \vdash \mathsf{range}(p') \quad s \neq s'}{\overline{s} \vdash \mathsf{crossJmp}(p, p')} \text{(Aux-cross-modules)}$$

Figure 8.4: Access control enforcement rules for Aim.

r is the registers file f is the flags register, m is the memory, $\overline{s}$ is the list of memory descriptors and O is the oracle. The dynamic semantics is given via relation $\xrightarrow{id} \subseteq \Omega \times id \times \Omega$ for reductions within the same module with id id (Figure 8.5). This relation extends the $\xrightarrow{i}$ relation of A+I by adding an assumption of the form $\overline{s} \vdash \mathsf{currentModule}(s, p, id)$ to all rules; this is used to indicate the id of the current module. Instead of re-stating all rules, Figure 8.5 presents how two example rules (Rule Eval-movl and Rule Eval-jump) together with the rule for generation of a random value (Rule Eval-rand). Reductions

$$\frac{\overline{s} \vdash \mathsf{currentModule}(s, p, id) \quad s \vdash \mathsf{intJmp}(p, p+1) \quad s \vdash \mathsf{readAllowed}(p, r(\mathtt{r_s})) \quad m(p) = \mathtt{movl}\ \mathtt{r_d}\ \mathtt{r_s} \quad r' = r[\mathtt{r_d} \mapsto m(r(\mathtt{r_s}))]}{(p, r, f, m, \overline{s}, O) \xrightarrow{id} (p+1, r', f, m, \overline{s}, O)} \text{(Eval-movl)}$$

$$\frac{\overline{s} \vdash \mathsf{currentModule}(s, p, id) \quad m(p) = (\mathtt{jmp}\ \mathtt{r_d}) \quad p' = r(\mathtt{r_d}) \quad s \vdash \mathsf{intJump}(p, p')}{(p, r, f, m, \overline{s}, O) \xrightarrow{id} (p', r, f, m, \overline{s}, O)} \text{(Eval-jump)}$$

$$\frac{\overline{s} \vdash \mathsf{currentModule}(s, p, id) \quad s \vdash \mathsf{intJmp}(p, p+1) \quad m(p) = \mathtt{rand}\ \mathtt{r_d} \quad r' = r[\mathtt{r_d} \mapsto w] \quad O = w; O'}{(p, r, f, m, \overline{s}, O) \xrightarrow{id} (p+1, r', f, m, \overline{s}, O')} \text{(Eval-rand)}$$

Figure 8.5: Reduction rules for Aim.

within unprotected memory (given by relation $\xrightarrow{un}$) are derived from those of within a module by changing the $\overline{s} \vdash \mathsf{currentModule} \cdots$ assumption with an $\overline{s} \vdash \mathsf{unprotected} \cdots$ one.

Reduction $\rightarrow\ \subseteq\ \Omega\times\Omega$ gives cross-module reductions for Aim programs (Figure 8.6). Cross-module calls and returns use $\overline{s}\vdash \mathsf{crossJmp}\cdots$ assumptions

$$\dfrac{\Omega \xrightarrow{id} \Omega'}{\Omega \rightarrow \Omega'}\ \text{(Eval-id)} \qquad \dfrac{\Omega \xrightarrow{un} \Omega'}{\Omega \rightarrow \Omega'}\ \text{(Eval-un)}$$

$$\text{(Eval-call-cross)}\quad \dfrac{\begin{array}{c}\overline{s}\vdash \mathsf{currentModule}(s,p,id) \quad m(p)=(\texttt{call r}_{\texttt{d}}) \quad p'=r(\texttt{r}_{\texttt{d}})\\ s\vdash \mathsf{crossJmp}(p,p') \quad p,r,m,\overline{s}\searrow^{\mathsf{SS}} p',r',m' \\ r''=r'[\mathsf{SP}\mapsto r(\mathsf{SP})+1;\texttt{r}_0\mapsto id] \quad m''=m'[r''(\mathsf{SP})\mapsto p+1]\end{array}}{(p,r,f,m,\overline{s},O)\rightarrow(p',r'',f,m'',\overline{s},O)}$$

$$\text{(Eval-ret-cross)}\quad \dfrac{\begin{array}{c}\overline{s}\vdash \mathsf{currentModule}(s,p,id) \quad m(p)=(\texttt{ret}) \quad p'=m(r(\mathsf{SP}))\\ s\vdash \mathsf{crossJmp}(p,p') \quad r'=r[\mathsf{SP}\mapsto r(\mathsf{SP})-1;\texttt{r}_0\mapsto id] \quad p,r',m,\overline{s}\searrow^{\mathsf{SS}} p',r'',m'\end{array}}{(p,r,f,m,\overline{s},O)\rightarrow(p',r'',f,m',\overline{s},O)}$$

$$\text{(Eval-call-un)}\quad \dfrac{\begin{array}{c}\overline{s}\vdash \mathsf{unprotected}(p,un) \quad m(p)=(\texttt{call r}_{\texttt{d}}) \quad p'=r(\texttt{r}_{\texttt{d}})\\ s\vdash \mathsf{crossJmp}(p,p') \quad p,r,m,\overline{s}\searrow^{\mathsf{SS}} p',r',m' \\ r''=r'[\mathsf{SP}\mapsto r(\mathsf{SP})+1;\texttt{r}_0\mapsto un] \quad m''=m'[r''(\mathsf{SP})\mapsto p+1]\end{array}}{(p,r,f,m,\overline{s},O)\rightarrow(p',r'',f,m'',\overline{s},O)}$$

$$\text{(Eval-ret-un)}\quad \dfrac{\begin{array}{c}\overline{s}\vdash \mathsf{unprotected}(p,un) \quad m(p)=(\texttt{ret}) \quad p'=m(r(\mathsf{SP})) \quad s\vdash \mathsf{crossJmp}(p,p')\\ r'=r[\mathsf{SP}\mapsto r(\mathsf{SP})-1;\texttt{r}_0\mapsto un] \quad p,r',m,\overline{s}\searrow^{\mathsf{SS}} p',r'',m'\end{array}}{(p,r,f,m,\overline{s},O)\rightarrow(p',r'',f,m',\overline{s},O)}$$

Figure 8.6: Rules for the $\rightarrow$ relation. un is the principal related to unprotected code.

while previously-defined same-module calls and returns relied on $\overline{s}\vdash \mathsf{intJmp}\cdots$ assumptions. Moreover, jumps between modules set the $\texttt{r}_0$ to the id of the principal from which the call originates.

8.4.3 Randomisation and Contextual Equivalence for Aim

To account for randomisation in Aim, a randomisation instruction is added to the language, and the odds of guessing its outcome must be considered.

Examples 32 to 33 describe the usage of oracles; Example 34 discusses how guessing random numbers affects the definition of contextual equivalence.

Example 32 (Obvious equivalences) *With an explicit oracle, obvious equivalences are respected; a program P_r that returns a random value is equivalent to itself.*

```
0  rand r0  // call this program P_r
1  ret      // return a random number in r0
```

In fact, for any oracle O, there exists an oracle (the same oracle O) that makes the behaviour of P_r independent of random values.

P_r is also equivalent to the following snippet P_{rr}, which generates two random numbers and only returns the second.

```
0  rand r0  // generate a number
1  rand r0  // then generate a second number
2  ret      // return the second random number in r0
```

The oracle that must be used with P_{rr} is one that has all elements of O interleaved with other elements. $\boxdot$

Example 33 (Obvious inequivalences) *Consider a program that returns two random numbers and a program that returns the same random number twice.*

```
100  rand r0
101   ··· // copy r0 in r1
102  ret
```

```
100  rand r0
101  rand r1
102  ret
```

For any oracle used with the left-hand side snippet, the oracle that contains the same value twice will make the right-hand side snippet equivalent to it. However, there is no oracle that will make the opposite true. $\boxdot$

By generalising these examples, we can conclude that any two Aim programs are equivalent if, for any oracle used with the first program, there exists an oracle for the second program such that they have the same behaviour *and vice versa.*

The notion of contextual equivalence for A+I programs will therefore be defined in terms of a preorder relation that must hold in both directions between two programs for them to be equivalent.

Example 34 (Guessing) *Consider two programs that store a random number (line 3) and then input from the external code on $\texttt{r}_1$ (line 6). If that input matches the random number, one returns 0, the other returns 1 (line 15), otherwise they both return 2 (line 14, the syntax is massaged for the sake of simplicity).*

```
100  rand r0
101  movi r1 120 // at address 120
102  movs r0 r1 // store r0
103  movi r0 0
// call unprotected code
104  call r0 ···
105  movi r1 120
// load the random number
106  movl r5 r1
// compare it with argument r1
107  cmp  r1 r5
108  movi r3 111
109  je   r3   // if it is not zero
110  ret 2    // return 2
111  ret 1    // otherwise return 1
```

```
100  rand r0
101  movi r1 120 // at address 120
102  movs r0 r1 // store r0
103  movi r0 0
// call unprotected code
104  call r0 ···
105  movi r1 120
// load the random number
106  movl r5 r1
// compare it with argument r1
107  cmp  r1 r5
108  movi r3 111
109  je   r3   // if it is not zero
110  ret 2    // return 2
111  ret 0    // otherwise return 0
```

If the random function is strong enough, external code has very little chance of telling these programs apart; intuitively, that chance is $\sim 1/2^{\ell}$ times the number of guesses (where ℓ is the length of a word in memory). However, the contexts considered in the definition of contextual equivalence are universally quantified. Thus there is also a context that differentiates between the two programs by guessing the number and we do not want to consider it [6, 62]. The definition of contextual equivalence needs to incorporate probability concepts to rule out these contexts that can succeed in guesses with negligible odds. ⊡

The probabilistic notion of contextual equivalence states that two programs are equivalent if they behave the same to a certain probability. Denote the probability of a certain event with $\texttt{Pr}(\cdot)$ and let σ range over the 0..1 interval.

Definition 39 (Contextual preorder for Aim) $P_1 \sqsubseteq_{\sigma}^{\mathsf{AIM}} P_2 \triangleq \texttt{Pr}(\forall \mathbb{P},\ O_1.\ \exists O_2.\ \mathbb{P}[P_1, O_1]\Uparrow \iff \mathbb{P}[P_2, O_2]\Uparrow) > \sigma.$

Contextual equivalence for Aim programs is thus defined as following.

Definition 40 (Contextual equivalence for Aim) $P_1 \simeq_{\sigma}^{\mathsf{AIM}} P_2 \triangleq P_1 \sqsubseteq_{\sigma}^{\mathsf{AIM}} P_2$ *and* $P_2 \sqsubseteq_{\sigma}^{\mathsf{AIM}} P_1$.

For securely-compiled components, if the randomisation function has a guessability of σ_g, σ must be $1 - \sigma_g$. Unprotected code can in fact only guess once, if the guess is wrong, a fault is detected and the execution is halted, therefore σ accounts only for one guess. Alternative definitions of analogous concepts can be found in work reasoning about the security of cryptographic primitives. For example, sometimes the distance between random number distributions is considered, which is analogous to what we are doing here, as we consider that distribution to range uniformly between 0 and $2^\ell - 1$. Alternatively, weaker attackers could be considered (e.g., by using an analogous definition of well-behaved contextual equivalence), but this would imply changing A+I, so this is not considered further.

Correctness of Definition 40

Consider two programs; one always returns 0, the other always returns a random value. Intuitively, we want them not to be equivalent, but with probability in place, they might be.

```
100  movi r0 0
101  ret
```

```
100  rand r0
101  ret
```

Consider for the sake of simplicity, $\ell = 1$, so words are 1 bit long, they can be either 0 or 1. The probability to guess a random number is then $1/2^1 = 1/2$, the guessability of the random number generator is assumed to be $\sigma_g = 1/2$. Contexts can therefore be split in two sets. Those that expect the output to be 0; and those that expect the output to be 1. In both cases, the two programs will be the same 1/2 the times so there is 1/2 total chance that a context differentiates these programs. However, to be declared equivalent, programs need to have the same behaviour for a number of times strictly greater than σ_g, which is 1/2. Since 1/2 is not greater than 1/2, the programs are not equivalent.

8.4.4 Trace Semantics for AIM

The trace semantics for AIM programs is an adaptation of the $\mathsf{Traces}^{\mathsf{S}}_{\mathsf{A+I}}$ semantics of Section 5.2, so its labels comprise call and returns but no readouts nor writeouts. These new states of the trace semantics must account for oracles, thus they are indicated as follows $\Theta ::= \Omega \mid (\mathsf{unk}, m, \overline{s}, O)$. Ω models that the partial program is executing, while state $(\mathsf{unk}, m, \overline{s}, O)$ models execution happening outside of the partial program. Given a program P and an oracle O, indicate its initial state (with program counter, registers and flags set to 0) as $\Theta_0(P, O)$.

Relation $\xrightarrow{\lambda}\!\!\!\!\rightarrow \subseteq \Theta \times \widehat{\lambda} \times \Theta$ captures how labels are generated. That relation is omitted as it is a small variation of the same relation defined for $\mathsf{Traces}^{\mathsf{S}}_{\mathsf{A+I}}$ with the changes for the semantics of AIM discussed in Section 8.4.2. For analogous reasons, relation $\overset{\overline{\alpha}}{\Longrightarrow}\!\!\!\!\Rightarrow \subseteq \Theta \times \widehat{\overline{\alpha}} \times \Theta$, which captures how traces are generated, is also omitted.

Define the trace semantics of a program P as:

$$\mathsf{Traces}_{\mathsf{AIM}}(P) = \{\overline{\alpha} \mid \forall O.\exists \Theta.\Theta_0(P, O) \overset{\overline{\alpha}}{\Longrightarrow} \Theta\}$$

.

Example 35 (Traces of example Aim **programs)** *Consider only the program spanning from address 100 to 200 from Example 30, consider the rest of the memory space to be undefined. The following traces are valid descriptions of the behaviour of that program (use $\cdots$ to gloss over unimportant details):*

$$\begin{aligned} T_1 &= \texttt{call}\ 100\ 12, 10, \cdots?\ \texttt{ret}\ 4\ 2! \\ T_2 &= \texttt{call}\ 100\ 5, 0, \cdots?\ \texttt{call}\ 300\ \cdots! \\ T_3 &= \texttt{ret}\ 106\ 0?\ \texttt{ret}\ 4\ 0! \end{aligned}$$

T_1 captures the behaviour of the code in case it is called with $\mathtt{r}_0$ greater than $\mathtt{r}_1$. T_2 captures the opposite. T_3 captures the behaviour of the code after it has called code in the other protected module. $\boxdot$

Two programs P_1 and P_2 are trace equivalent, denoted with $P_1 \overset{\mathrm{T}}{=}{}^{\mathsf{AIM}}_{\sigma} P_2$ if their trace semantics coincides with a certain probability.

Definition 41 (Trace equivalence for Aim**)** $P_1 \overset{T}{=}{}^{\mathsf{AIM}}_{\sigma} P_2 \triangleq \mathtt{Pr}(\mathsf{Traces}_{\mathsf{AIM}}(P_1, O_1)$ $= \mathsf{Traces}_{\mathsf{AIM}}(P_2, O_2)) > \sigma$.

For securely-compiled components, trace semantics coincides with contextual equivalence, as captured by Proposition 1.

Proposition 1 (Fully abstract trace semantics for securely-compiled Jem **programs)** $\forall C_1, C_2, C^T \in \mathsf{JEM}. [\![C_1 + C^T]\!]^{\mathsf{JEM}}_{\mathsf{AIM}} \simeq^{\mathsf{AIM}}_{\sigma} [\![C_2 + C^T]\!]^{\mathsf{JEM}}_{\mathsf{AIM}} \iff [\![C_1 + C^T]\!]^{\mathsf{JEM}}_{\mathsf{AIM}} \overset{T}{=}{}^{\mathsf{AIM}}_{\sigma} [\![C_2 + C^T]\!]^{\mathsf{JEM}}_{\mathsf{AIM}}$.

Proof Sketch. Securely-compiled components only use `call` and `ret` instructions to jump outside protected code. No other instruction is executed on external code, e.g., no reading or writing in unprotected memory. Moreover, external code also can only jump to the entry points of protected code and it cannot perform any other instruction on protected code. Since unused registers and flags are reset at each jump from and to the protected code, the information communicated on the labels captures the behaviour of protected code precisely. □

8.5 The Source Language Jem, Formally

Jem is an extension of the J+E language of Chapter 3 that includes principal definitions and trust statements. This section presents the language syntax (Section 8.5.1) and the additions to the static semantics of J+E (Section 8.5.2).

8.5.1 Syntax

The syntax of Jem is the same as that of J+E (Figure 3.1). The only addition is principals, which are taken from the denumerable set $\mathcal{PR}$, and statements to define the principal of an export package and which principals does a package trust (Figure 8.7).

$$
\begin{array}{ll}
\textit{components} & C ::= \overline{P} \\
\textit{packages} & P ::= \{\texttt{package}\ p\ \texttt{of principal}\ t; \overline{D_i}\} \\
 & \quad\mid \{\texttt{package}\ p\ \texttt{of principal}\ t; \overline{T}; \overline{D_e}\} \\
\textit{principals} & \quad t \in \mathcal{PR} \\
\textit{truststatements} & T ::= \texttt{trust principal}\ t;
\end{array}
$$

Figure 8.7: Syntax of Jem.

8.5.2 Static Semantics

The static semantics extends that of J+E with consistency check on principals, which are presented below. Function $\textit{principal}(P)$ returns the principal of the package, i.e., $\textit{principal}(\{\texttt{package}\ p\ \texttt{of principal}\ t; \overline{T}; \overline{D_i}\}) = t$.

$$
\text{(Packages)}\quad \frac{C \vdash \overline{D} : \mathsf{dec\ in}\ p \qquad C \vdash \overline{T} : \mathsf{trs}}{C \vdash \{\texttt{package}\ p\ \texttt{of principal}\ t; \overline{T}; \overline{D_i}\} : \mathsf{pkg}}
$$

$$
\text{(Trust)}\quad \frac{\exists P \in C.\textit{principal}(P) = t \ \wedge\ \textit{isExport}(P)}{C \vdash \texttt{trust principal}\ t; : \mathsf{trs}}
$$

Rule Packages checks the validity of its definitions and of its trust statements. Rule Trust states that a trust statement is valid only if it talks of a principal whose code is implemented in the component.

As required by the new definition of full abstraction (Definition 38), a notion of trust between components needs to be defined. A component C trusts another component C' if the principal of C' is included in the trust statements of the export packages of C (Rule Jem Trust).

$$\frac{\begin{array}{c}\forall P \in C.\mathit{isExport}(P) \wedge, P \equiv \{\texttt{package}\ p\ \texttt{of}\ \texttt{principal}\ t; \overline{T}; \overline{D_i}\} \\ \exists P' \in C'.\neg \mathit{isExport}(P') \wedge, P' \equiv \{\texttt{package}\ p'\ \texttt{of}\ \texttt{principal}\ t'; \overline{T'}; \overline{D'_i}\} \\ \wedge\ \texttt{trust principal}\ t' \in \overline{T}\end{array}}{C\ \mathit{trusts}\ C'}\ (\mathsf{JEM\ Trust})$$

8.6 Multi-Principal Full Abstraction for $[\![\cdot]\!]_{\mathsf{AIM}}^{\mathsf{JEM}}$

This section presents the theorem statements whose proof would imply security of the compilation scheme of Section 8.3. These theorems are analogous to the ones proven for the compilation scheme of Section 6.2.

Conjecture 1 captures the assumption that the compiler translates Jem expressions into a behaviourally-equivalent list of Aim instructions.

Conjecture 1 (Compiler preserves behaviour) *Assuming there is no overflow of the secure stack and of the secure heap, the secure compiler outputs* Aim *programs that behave as their* Jem *counterparts. Thus:* $\forall C_1, C_2, C^T \in \mathsf{JEM}.$ $[\![C_1 + C^T]\!]_{\mathsf{AIM}}^{\mathsf{JEM}} \simeq_{\sigma}^{\mathsf{AIM}} [\![C_2 + C^T]\!]_{\mathsf{AIM}}^{\mathsf{JEM}} \Rightarrow (C_1 + C^T) \simeq^{\mathsf{JEM}} (C_2 + C^T)$.

The compilation scheme of Section 8.3 considers a correct compiler as starting point and only adds checks without modifying the translation of source-level expressions into target-level instructions. Thus, we believe this conjecture holds.

Conjecture 2 states that it is always possible to construct a source-level component that distinguishes between two securely-compiled Jem components that exhibit a different target-level trace semantics.

Conjecture 2 (Compiler security) *Assuming there is no overflow of the secure stack and of the secure heap, and that the randomisation function can be guessed with a* σ_g *probability. Let* $\sigma = 1 - \sigma_g$*, then:* $\forall C_1, C_2, C^T \in \mathsf{JEM}.[\![C_1 + C^T]\!]_{\mathsf{AIM}}^{\mathsf{JEM}} \not\simeq_{\sigma}^{\mathsf{AIM}} [\![C_2 + C^T]\!]_{\mathsf{AIM}}^{\mathsf{JEM}} \Rightarrow (C_1 + C^T) \not\simeq^{\mathsf{JEM}} (C_2 + C^T)$.

We only argue that this conjecture holds without any formal proof. Based on the examples of Section 8.2, the compiler has checks that prevent any target-language context from violating the well-typedness of securely compiled Jem components. So the only way that Aim contexts have to interoperate with a compiled Jem component is to behave like a Jem context, thus without violating any of the desired security properties (as highlighted in the thread model of Section 8.1.4. If two different traces are provided, an algorithm that is analogous of that of Section 7.1 could be developed, as for the secure compilation scheme of Chapter 6. The witness must replicate all ?-decorated actions at the source level and, upon receiving the distinguishing !-decorated action, it must tell whether it is $(C_1 + C^T)$ or $(C_2 + C^T)$. The development of such an algorithm is left for future work.

The presented compilation scheme is multi-principal fully abstract (Conjecture 3).

Conjecture 3 (Multi-principal full abstraction for $[\![\cdot]\!]^{\mathsf{JEM}}_{\mathsf{AIM}}$) *Assuming that there is no overflow of the secure stack and of the secure heap and that the randomisation function can be guessed with a σ_g probability. Let $\sigma = 1-\sigma_g$, then:* $\forall C_1, C_2, C^T \in \mathsf{JEM}$. *if* C_1 *trusts* $C^T \wedge, C_2$ *trusts* C^T *then* $(C_1 + C^T) \simeq^{\mathsf{JEM}} (C_2 + C^T) \iff [\![(C_1 + C^T)]\!]^{\mathsf{JEM}}_{\mathsf{AIM}} \simeq^{\mathsf{AIM}}_{\sigma} [\![(C_2 + C^T)]\!]^{\mathsf{JEM}}_{\mathsf{AIM}}$.

This conjecture holds as soon as Conjecture 1 and Conjecture 2 do. Since multi-principal full abstraction implies security of the compilation scheme, proving this theorem implies that $[\![\cdot]\!]^{\mathsf{JEM}}_{\mathsf{AIM}}$ is secure.

Chapter 9

Evaluation and Discussion

The aim of argument, or of discussion, should not be victory, but progress.

Joseph Joubert

This chapter firstly presents benchmarks that evaluate the computational overhead introduced by the secure compiler (Section 9.1). Then, it briefly considers how to port the results developed to the Intel SGX architecture (Section 9.2). Finally, it presents the limitations of the secure compilation scheme (Section 9.3) and discusses different definitions of a secure compilation scheme (Section 9.4).

9.1 Benchmarking

This section details the architecture adopted to develop the secure compiler of Chapter 6 (Section 9.1.1). Then it presents benchmarking of the overhead introduced by the secure compiler (Section 9.1.2).

9.1.1 The Fides Protected Module Architecture

The secure compilation scheme of Section 6.2 relies on the target language having a protected module architecture for it to be secure. In order to time the overhead of the secure compiler we implemented it for the Fides architecture [112].

The Fides architecture implements precisely the protection mechanism described in Section 2.1 in a very small TCB: ∼7000 lines of code. Fides consists of a hypervisor that runs two virtual machines: the secure VM handles the protected memory section and the Legacy VM handles the unprotected one. Switching between the two virtual machines of Fides (i.e., when performing calls and outcalls) is more costly than in a hardware-based implementation. However, we are not interested in this overhead: the overhead that we are interested in timing is the one provided by the additional checks introduced by the secure compiler.

This section gives a brief description of the Fides architecture, followed by an informal presentation of the implementation of the secure compiler.

Legacy VM The Legacy VM executes all legacy applications and other code in unprotected memory. Using virtualisation 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, 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 accessing the protected module and it prevents the Secure VM from accessing the hypervisor.

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

Secure VM The Secure VM can access all memory, with the exception of memory containing the hypervisor. The fine-grained memory access control mechanism is implemented by a security kernel running in this VM, as follows. First, when a request is received from the hypervisor to execute a method in the protected module, the requested entry point is checked against a list of valid entry points provided in the module's memory descriptor. When this check passes, the hardware memory management unit is set up to allow memory accesses to the module's memory region and execution proceeds from the entry point that was called. When execution tries to jump back out of the protected module, a page fault is generated, which causes the security kernel to ultimately return execution control to the Legacy VM.

Secure Build Tools

To simplify the development and benchmarking of modules, a fully abstract compilation tool chain has been developed using the LLVM compiler and the ELF Tool Chain library.[1]

Compilation of modules follows the guidelines of Section 6.2 and is done in two steps. First, for every function that is annotated as an entry point, an entry in the module's entry point table is created. Each entry checks whether an initialisation function needs to be called, sets up the stack pointer and stores the return address in unprotected memory on the stack. After the correct function is called, registers not carrying a result value are cleared. Wrapper functions for each entry point are also generated, in order to simplify the calling of the module. Secondly, the source code is analysed and modified so that every call site that results in an outcall to unprotected memory flows through the callback entry point. Registers not carrying a function parameter are reset, so that no information is leaked.

After compilation of the module – possibly resulting in multiple ELF files if the source code was split over multiple files – a secure linker lays out the module in memory according to Fides' requirements. Protected modules must start with the entry table, followed by all compiled code and read-only data such as strings (i.e., the Code section) and the runtime stack and security sensitive variables (i.e., the Data section).

9.1.2 Measurements

To benchmark the cost of the additional checks introduced by the secure compilation scheme, we have implemented stub objects in C, a data structure that models the low-level representation of objects. Stub objects have an **Int**eger field that indicates the class of the object followed by the fields of the object. We have implemented a secure runtime containing the data structure

[1]Respectively available at `http://llvm.org/` and `http://sourceforge.net/p/elftoolchain/wiki/Home/`.

$\mathcal{O}$ and functions to mask object references through it. The secure runtime also implements the runtime checks presented in Section 6.2. These tests are simply used as an indicator that the overhead introduced by the compiler is reasonable. More detailed measurements, detailing the strengths and limitations of particular PMA implementations are left for future work.

We have then taken a simple program and, using a hardware high-frequency timestamp, we have timed its performance in three cases, as presented in Figure 9.1. The figure presents the average program execution time without any protection (in blue), with Fides (in red) and with Fides extended with the runtime checks provided by the security runtime (in beige).

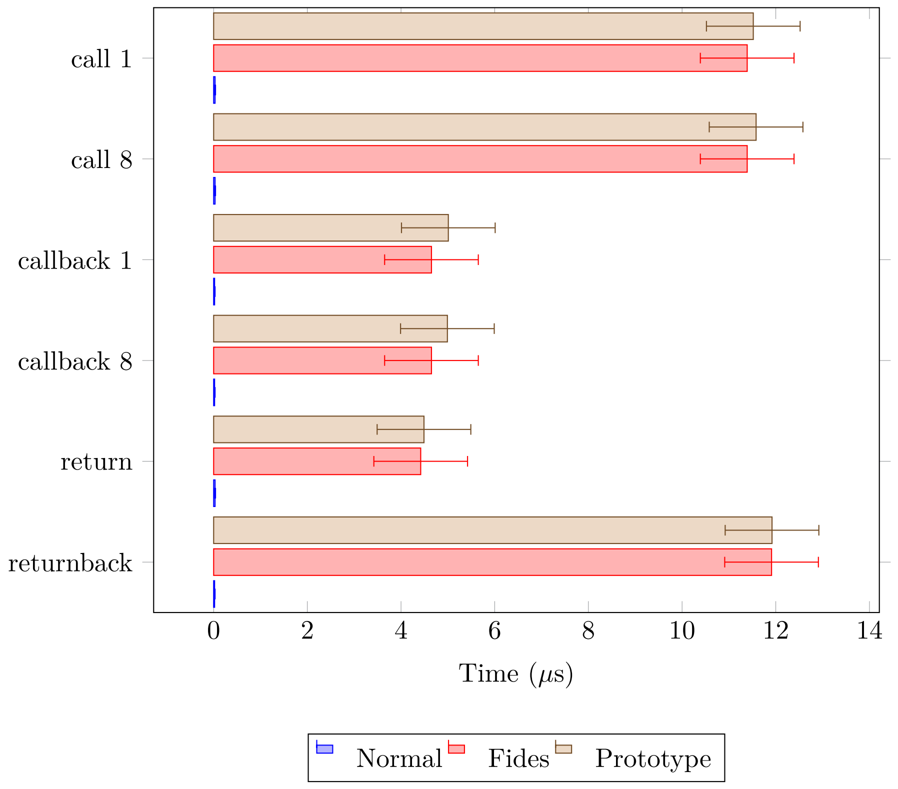

Figure 9.1: Cost of different benchmarked instructions.

In Figure 9.1, the y axis indicates which operations have been tested. The number following calls and callbacks indicates the number of arguments used, they trigger runtime checks. The security runtime adds checks to calls, callbacks, returns and returnbacks, so they are the only instructions that are considered. The “Normal” (blue) row indicates the cost of each operation without using the Fides architecture. The “Fides” (red) row indicates the cost of each operation while using the Fides architecture without the secure compilation scheme. The “Prototype” (beige) row the cost of each operation when the secure compilation scheme is used in addition to Fides. Each operation was performed 1000 times on a MacBook Pro with a 2.3 GHz Intel Core i5 processor and 4GB 1333MHz DDR3 RAM. The difference between rows “Normal" and “Fides" shows the already high overhead of adopting the Fides architecture. The difference between rows “Fides” and “Prototype” is the overhead of the security checks introduced by the secure compiler: on average, this is a ~3% overhead. Security checks are triggered only when the boundary between the protected and the unprotected memory partitions is crossed. Method calls within the same memory partition suffer no overhead. The overhead introduced by the compiler is proportional to the number of boundary crossings.

9.2 Intel Software Guard eXtensions

In June 2013 Intel publicly disclosed its work on Software Guard eXtensions (SGX) [15, 58, 82]. SGX provides a hardware-implemented isolation mechanism that is very similar to Fides [112] and related protected module architectures [91, 111, 112, 113]. *Enclaves* is the SGX terminology for what we called protected modules in this paper. Enclaves live in the same address space as unprotected parts of the application and can only be accessed through an explicitly exposed interface. Direct memory accesses from unprotected memory to enclaves memory regions are prevented. Enclaves, like protected modules, have full access to unprotected parts of the application. We believe that the presented fully abstract compilation scheme can be easily ported to SGX-enabled platforms, modulo small technical changes.

There are a few notable differences between Fides and SGX. For instance, SGX requires special entry and exit instructions to cross enclaves boundaries, while Fides does not. SGX also provides only a single entry point to enclaves [82], but additional entry points can be emulated by taking the index of the intended function as an additional function argument. The main difference between Fides and SGX is the following. To prevent denial-of-service attacks by buggy or malicious modules that never return control to code in unprotected memory, SGX supports interruption of enclaves. Unfortunately, without added security measures, this may violate the integrity of SGX enclaves, as explained below.

Problem 20 (SGX Interrupts) *Consider two classes that define the same methods* `plusTwo` *and* `reset` *that, respectively, increment a variable* `even` *by two and reset that variable to 0. These two classes are implemented by two objects:* $\mathtt{o}_L$ *and* $\mathtt{o}_R$*. Assume these objects are compiled to SGX enclaves.*

```
 1 package p;
 2 class C_L {
 3   private even : Int = 0;
 4
 5   public plusTwo() : Int {
 6     even = even + 1;
 7     even = even + 1;
 8     return 0;
 9   }
10
11   public reset() : Int {
12     even = 0;
13     return 0;
14   }
15 }
16 object o_L : C_L
```

```
 1 package p;
 2 class C_R {
 3   private even : Int = 0;
 4
 5   public plusTwo() : Int {
 6     even = even + 2;
 7
 8     return 0;
 9   }
10
11   public reset() : Int {
12     even = 0;
13     return 0;
14   }
15 }
16 object o_R : C_R
```

Objects $\mathtt{o}_L$ *and* $\mathtt{o}_R$ *are equivalent at the source-code level, but their compiled counterparts are not. The enclave with the compiled counterpart of* $\mathtt{o}_L$ *may receive an interrupt before executing the instruction located at line 7 in method* `plusTwo`*. The interrupt can call to method* `reset` *before the execution of* `plusTwo` *is resumed. This will result in* `even` *holding value* `1` *in* $\mathtt{o}_L$*, while* $\mathtt{o}_R$ *will either have value 0 or 2 in* `even`*.* ■

This integrity constraint violation of Problem 20 can be prevented by ensuring that new entry points cannot be called while an interrupt is handled. In practice, this can be accomplished by adding a boolean field `busy` to the compiled code. The `busy` field is set when the module is entered and reset before control is passed back to unprotected memory. In case a module is interrupted, it has still the `busy` variable set and the module may refuse to service the function request. To avoid race conditions, an atomic `test` − `and` − `set` instruction should be used to keep track of the `busy` variable.

Finally, SGX, in contrast to Fides also provides protection against hardware attacks such as an attacker snooping the memory bus [122] or performing a cold boot attack [55]. This enables various TPM primitives to be offloaded to the main CPU but does not impact the development of a fully abstract compilation scheme.

9.3 Limitations of the Compilation Schemes

This section presents limitations of the presented compilation schemes (both Section 6.2 and Chapter 8).

Like many model languages [6, 62], J+E lacks features that real-world programming languages have, such as multithreading, foreign-function interfaces and garbage collection. A thorough investigation of the changes needed in order to support secure compilation of languages with those features is left for future work. For now we informally present the research challenges of performing garbage collection (Section 9.3.1) in concert with securely compiled code and sketch the path future research could follow to address these challenges.

9.3.1 Garbage Collection and Secure Compilation

Garbage collection is a runtime mechanism of certain languages that (generally) manages whole programs. Since in the secure compilation scenario whole programs are split between the protected and the unprotected memory partition, the garbage collector (GC) would be also split into a protected and an unprotected part. Given the powers of an attacker to the system (Definition 7 in Section 2.4), the attacker can tamper with the unprotected GC but not with the protected one. The attacker can thus inspect all references that the unprotected GC has, introduce fake pointers and impersonate the unprotected GC when interacting with the protected one.

Since the unprotected GC can be tampered with, the implementation of a secure garbage collector is reduced to extending the securely compiled program with a secure GC in charge of the secure memory partition. The secure GC must be trusted and its code must be placed inside the protected memory partition, so it can access $\mathcal{O}$ and the object graphs of the protected objects. However to allow the secure GCs to communicate with a GC in unprotected memory (for the case when there is no attacker), additional entry points need to be set up. Unfortunately, this violates full abstraction in a similar way to that pointed out by Abadi for Java [1]: this functionality is available only at the target level but not at the source level. The first challenge is proving that the secure GC does not introduce security leaks. The possible approach to such a proof is described at the end of this section.

Additional challenges arise for the implementation of the secure GC. A common implementation for GCs is reference counting. With reference counting, the GC keeps track of how many references an object has, when this counter reaches 0, the objects can be safely deallocated, as no other object has a reference to it. Reference counting introduces a failure of full abstraction, as highlighted by the code below.

```
1 package p;
2 class C_L {
3   public doCb( Object x , Object y
        ) {
4     while ( ··· ) {   //infinite
          loop
5       x.callback( this );
6     }
7   }
8   y.callback();
9 }
```

```
1 package p;
2 class C_R {
3   public doCb( Object x , Object y
        ) {
4     while ( ··· ) {   //infinite
          loop
5       x.callback( this );
6     }
7   }
8
9 }
```

In these code snippets, both functions `doCb` receive two arguments `x` and `y` (line 3) and loop infinitely (line 4) on performing outcalls on `x` (line 5). Additionally, the left hand side snippet has unreachable code where `callback` is called on `y` as well (line 8). A garbage collector that does reference counting will behave differently in these two cases. In fact, it will keep a reference to `y` in $\texttt{C}_L$ and not in $\texttt{C}_R$, as it cannot know that `y` lies in unreachable code without solving the halting problem. This is a failure of full abstraction: $\texttt{C}_L$ and $\texttt{C}_R$ behave the same at the source level but not at the target level, when a garbage collector is considered. A simple solution to problem would be to change the way references are counted and let parameters also increase the counter for an object reference. The return would decrease the counter for all parameters.

However, a second challenge arises: once a reference to an internal object is passed to the unprotected code (such as for `this` in the `callback` above) the GC does not know when to deallocate such a reference. Here, an arguably safe methodology is to not deallocate a reference that is passed from the secure component to unprotected code. However, this creates problems when the allocated object is large or when many references are passed out, namingly that a large part of memory cannot be freed.

An analogy that can be made now is that the secure GC faces challenges similar to those faced by distributed garbage collectors [7]. Passing a reference to unprotected code is in fact analogous to passing a reference to a remote program: it is difficult to find out when such a reference can be deallocated. In the distributed setting this is due to communication problems and the impossibility for a garbage collector to inspect the object graph of a program on a remote machine. In this setting, this is because the object graph in the unprotected memory section can be tampered with by the attacker. Unfortunately, in this setting the GC needs not only to be performant in case of interaction with unprotected code, it needs also to defend from potential attackers. Research on distributed garbage collection has developed several ways to addresses this problem, for example by giving each leaked reference a lease time. An unprotected object receiving a leased reference must periodically renew the lease on the reference, because once the lease time has expired the reference is collected by the secure GC. This solution could be adopted in order to implement a secure memory manager, for it already addresses the need to provide a performant GC algorithm in case the unprotected code is well behaved. The details of the implementation, of how to make such a GC resilient against attacks and a more thorough treatment of the problems arising in various implementations are left for future work.

The proof that the garbage collector is secure remains a great challenge for the integration of secure compilation and GC. In any way such a proof is approached, GC notions (allocation, deallocation, references, the object graph etc.) need to be carried into both the target and the source language in order to prove the compilation scheme between the two to be fully abstract [87]. However, this causes the source-level programming model to become hindered by memory management – an additional way to let programmers introduce security flaws in their code. A way to overcome this challenge is to capture the behaviour of the GC in a different semantics of the source-level language, an *extended semantics*. Full abstraction of the compilation scheme should then be proven with respect to the extended semantics. This treatment would not clutter the source language with explicit GC, but it would still capture the effect of GC at the source level. The extended semantics should capture the behaviour of the GC as it is at the target level, so that problems as those highlighted above do not arise, since they are captured at the source level as well. Additionally, the extended semantics should be proven to be secure w.r.t the normal operational semantics: this would guarantee that the behaviour of the GC does not introduce security leaks.

Devising such an extended semantics and proving (i) that it is not introducing security leaks and (ii) a fully abstract translation involving it are left for future work.

9.4 Different Formulations of Secure Compilation

Fully abstract compilation is not the only way to define that a compilation scheme is secure. Another such definition is security-types preserving compilation (Section 9.4.1).

Alternatives to define a secure compilation scheme exist, but they do not have a crisp, acknowledged definition like full abstraction and security types preservation. The general idea behind these alternative definitions is to establish a property between source-level programs that captures the security abstraction one is interested in. Then, the compilation scheme must be proven to preserve that property in compiled programs in the presence of arbitrary target-level code interacting with the compiled program. To the best of the author's knowledge, no definition or official statement of such a property exists (yet).

9.4.1 Security Types Preserving Compilation

The definition of security types preserving compilation relies on two assumptions. Firstly, the security properties of interest in the source language are enforced only by means of a type system. Secondly, the target language is equipped with a type system that entails non-interference.

Concerning the first assumption, starting from the seminal work of Volpano *et al.* [117], security type systems have been developed in a variety of forms, often in order to tackle non-interference [115,126]. To prove non-interference the input of a program, as well as its variables, are tagged to be either *high* or *low* security (more complicated security lattices can be considered [33] but we will not delve into that for the sake of simplicity). Informally, a program is non-interfering if and only if any sequence of low inputs will produce the same low outputs, regardless of what the high level inputs are. By denoting secure values as *high*, if a program is non-interfering, its security policies are not violated.

Concerning the second assumption, it has been shown that languages such as assembly, though providing low-level abstractions, can enjoy the benefits of powerful type systems. The most renowned example of such a language is the Typed Assembly Language (TAL) of Morrisett *et al.* [88]. It is important to note that attackers written in these languages must also be well-typed.

When a compiler is security types preserving, it transforms non-interfering source programs into non-interfering target programs. Formally, let $\Gamma \vdash \mathsf{NI}(P)$ denote that program P is non-interfering according to environment Γ. A compiler $[\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}}$ is security types preserving if the following holds: $\forall P \in \mathcal{S}, \Gamma \vdash \mathsf{NI}(P) \Rightarrow \Gamma \vdash \mathsf{NI}([\![P]\!]^{\mathcal{S}}_{\mathcal{T}})$.

Proving a compiler to be security types preserving seems more straightforward than proving it fully abstract, as it does not require any reasoning about contextual equivalence. However, it is only applicable to a setting where the target language is typed with a type system that is powerful enough to entail non-inteference.

Chapter 10

Related Work

In questo campo [analisi dei programmi], piccolo è bello. Non dappertutto funziona così però!

In [the field of program analysis], small is good. Mind that this does not apply to all fields!

A prof., lecturing on program analysis

This chapter presents related works. Firstly, it discusses security architectures (Section 10.1) and related work in compiler research (Section 10.2). Then it surveys other secure compilation results (Section 10.3). Finally, it discusses fully abstract semantics (Section 10.4) and the usage of principals in other work (Section 10.5).

10.1 On Target-Level Protection Mechanisms

This section presents a number of target-level protection mechanisms that have been used (or could be used) to achieve secure compilation: ASLR (Section 10.1.1), *PMA* (Section 10.1.2), Sandboxing (Section 10.1.3), Capability machines (Section 10.1.4), the Crash-Safe machine (Section 10.1.5) and massive-scale differentiation techniques (Section 10.1.6).

10.1.1 Address Space Layout Randomisation (ASLR)

ASLR is a technique that randomises the memory layout of key data areas of a program such as the base of the stack, of the heap, of libraries, etc. when creating the executable. The executable is divided into segments whose order is randomised by the dynamic linker (i.e., just before running the executable). This technique is used to hinder an attacker from mounting "return to libc" attacks, and from using previously acquired knowledge of the location of certain data to access that data in subsequent runs of a program.

10.1.2 Protected Module Architectures (*PMA*)

PMA is an assembly-level isolation mechanism implemented in several research and industrial prototypes that encodes the access control policy of Section 2.1 [40, 80, 81, 82, 91, 109, 111, 112]. As mentioned in Section 9.1.1, Fides is a hypervisor implementation of such a machine [112]. Only two hardware implementations exist: Sancus [91] and the Intel SGX [15, 82]. The Salus implementation provides the same guarantees of *PMA* but in a higher position in the software stack, it is in fact an OS-based implementation [111]. Mondriaan is a *PMA* implementation with fine-grained protection schemes aimed at developing less monolithic operating systems [123]. Flicker is a software-based *PMA* implementation that can execute pieces of code in complete isolation guaranteeing secrecy of sensitive information [81]. Some disadvantages of this architecture have been mitigated in Trust visor by using a hypervisor [80]. The TCB of *PMA* can be very small [81, 109, 112, 113] or even zero-software [40, 82, 91]. A more in-depth analysis of variations of the *PMA* concept can be found in Raoul Strackx's Ph.D. thesis [110].

A formalisation of *PMA* as an untyped assembly language extended with a program counter-based access control manager was provided by Patrignani and Clarke [95]. Albeit glossing over specific implementation details, this formalisation captures the general access control policy and it can, up to due changes, scale to effectively model specific *PMA* implementations.

The isolation mechanism of *PMA* has been studied by Larmuseau *et al.* [70]. They specified an operational semantics that securely combined an abstraction-rich language with a model of an arbitrary attacker without relying on any static checks. To do so, they lifted the *PMA* memory isolation mechanism into the semantics of a multi-language system à la Matthews and Findler [78].

10.1.3 Sandboxing

A protection mechanism seemingly dual to *PMA* is sandboxing [51,127]. With sandboxing, a trustworthy environment is extended with a location (the sandbox) where non-trustworthy code is placed and monitored as to detect any malicious action it performs. Conceptually, sandboxing seems to be dual of *PMA*, thus we expect that the same insights developed in the secure compilation works for *PMA* would lead to the development of a secure compiler for sandboxed programs. However, no such secure compiler has been devised yet.

10.1.4 Capability machines

Capability machines embody the capability paradigm of Dennis and Van Horn [34]. The idea revolves around the concepts of *subjects*, *operations* and *objects*: subjects perform operations on objects. For example, Alice (a subject) can write (an operation) something to the filesystem (an object). With capabilities, Alice's write succeeds only if she is able to present a capability that allows her to *at least* write to the filesystem. In this case, we say that Alice has *permission* to write the filesystem. Were she not able to present such a capability, the write would fail. With capabilities, *only connectivity begets connectivity.* So if a subject cannot create a capability to an object, and if she does not receive a capability to an object, she cannot perform operations on the object. Capabilities are thus *not forgeable*, all Capability Machines provide a *supervisor* (to use the terminology of Dennis and Van Horn) that ensures this. The literature on the subject is very vast, but for secure compilation purposes, we are interested in implementations of this machine. Few capability machines implementation exist: the M-Machine [25], Cheri [124] and Capsicum [119]. In the M-Machine, capabilities are words whose first bit is set to 1 (no instruction can set this bit to 1 without calling the supervisor). Capsicum introduces capabilities at the OS level in a BSD-like operating system. Cheri introduces capabilities at the hardware level like the M-Machine. However, a capability co-processor is added to the CPU in order to handle capabilities; normal instructions then succeed only if an enabling capability is present in the co-processor.

Capability Machines are very expressive security architectures that would suit being targeted by a secure compiler.

10.1.5 The Crash-Safe machine

A more general-purpose security architecture is the Pump machine [35,36], which is the base of the Crash-Safe machine [17]. The architecture permits arbitrary meta-data tracking at the machine code level, which can be used to enforce security policies like information flow ones [17]. Support for the enforcement of these security properties suggests that these security architectures may also be suitable targets for secure compilation schemes.

10.1.6 Massive-Scale differentiation

Massive-scale software differentiation is a biologically-inspired software defence mechanism that aims at protecting the same software from being susceptible to the same attack. The idea is that the same software is not distributed in the same copy but each copy has small modifications that do not impact the general behaviour but that can hinder certain attacks [59, 71]. This defence mechanism has a clear property that can be formalised: all copies of the same software generated via differentiation techniques must be fully abstract w.r.t. the original one. This technique seems to provide some of the benefits of ASLR, so it could be given to the target language of a secure compiler in order to exploit it for secure compilation purpose. No work that achieves such a result exists, though.

10.2 On Compilers

This section describes related research advances concerning compilers.

Certifying compilers generate target code that is accompanied by a certificate that the code enjoys certain properties. PCC [90] is a mechanism that binds a program to a proof of its properties, so that a host can check the properties of a program before executing it. PCC can be integrated with secure compilation to allow the insecure code to prove that it is compliant to a pre-defined agreement, i.e., it is not malicious, so to speed up runtime checks. Several of the surveyed works envisage such a cooperation between PCC and secure compilation [18, 19, 20, 73, 88].

A related, yet different research field is that of verified compilation (also known as certified compilation). Verified compilation aims at providing formal guarantees that a compiler is correct [74]. This correctness is often stated as semantics preserving: the compiler is proven to output target code that behaves as its source-level counterpart. To this extent, verified compilation works make extensive use of theorem provers, the most widely used being Coq. The main difference with secure compilation is that there is no notion of security in verified compilation. This influences the notion of low-level contexts which compiled programs interact with; they are obtained by compiling high-level contexts, so low-level attackers have the same power as high-level ones. Results from this field can be employed in a secure compilation scenario, as they provide one direction of the proof of fully abstract compilation, namely the one we assumed with Theorem 6.

The CompCert project is the most well-known effort to provide a verified compiler; CompCert is a Coq-verified multi-pass compiler for a (growing) subset of C to PowerPC, ARM and x86 assembly [74, 76]. Other works have followed the CompCert approach and provided verified compiler for multithreaded languages [77], just-in-time compilation [89] and C with relaxed memory concurrency [107].

Chlipala [28] also provided a Coq-verified compiler for the simply-typed λ-calculus to assembly language that is proven to be type-preserving. Type preservation is also an often proven result about compilers [27, 54, 73]. Type-preservation, however, does not entail security of the compiler. In certain cases a type-preserving compiler can also be secure [73], but that follows from additional statements besides type preservation.

All these results apply to whole programs but recently, verified compilers have appeared for partial programs as well. Benton and Hur [22] provided a verified compiler from a call-by-value λ-calculus to a SECD machine as well as for System F with recursion to the same target language [23]. Hur and Dreyer [60] devised a verified compiler between an idealised ML to assembly. These works rely on a logical relation between the source and the target language to prove semantics preservation. Claiming that the results above do not scale to multi-pass compilers (i.e., real-world compilers), Perconti and Ahmed devised a two-steps verified compiler for System F with existential and recursive types to TAL. Their works relies on logical relations that are devised in a multi-language setting. The formal developments of these works hint at possible alternatives to prove full abstraction for compilers for partial programs. Moreover, these techniques seem more scalable than the approach proposed in this thesis (i.e., the algorithm), however, they have not been applied to secure compilation (yet).

10.3 On Secure Compilation

This section surveys the existing research on secure compilation. This section is organised based on the target language feature exploited: type systems (Section 10.3.1), cryptographic primitives (Section 10.3.2), memory protection mechanisms (Section 10.3.3) and the insertion of dynamic checks (Section 10.3.4).

10.3.1 Type Systems-Based

Following are the works that achieve secure compilation by exploiting the type system provided by the target language. In most cases, secure compilation is achieved by means of type-preserving compilation.

Typed Assembly Language is a RISC assembly language extended with a type system based on primitive types (e.g., integers, records, arrays), higher order, recursive types and more [88]. This language has been targeted by Morrisett *et al.* to securely compile a variant of System F augmented with integers, products and term-level recursion [88]. The compilation phase consists of five type-preserving steps: continuation-passing style (CPS) conversion, closure conversion, hoisting, allocation and code generation. The target languages of these steps are relatively similar to each other, starting from System F and gradually adding features such as closures and explicit memory allocation. Each translation step is type-directed, i.e., it is guided by the type of the expression that is being translated, and generates well-typed target-level code. In the original work, these translation steps are not formally proven to be correct, in fact, subsequent work proves that certain of these steps cannot be made with a fully abstract translation [13].

As a first step, Ahmed and Blume proved that typed closure conversion from (and to) System F is fully abstract [12]. This translation exploits additional typed wrappers for source terms in the target language. Typed closure conversion turns each function into a pair consisting of a function pointer and a closure environment that provides bindings from free variables to values. The conversion is type-directed and generates typed pairs, which are given the type of their closure environment. Typed wrappers are terms that translate source values v of type t to target values of type $[\![t]\!]^{\mathcal{S}}_{\mathcal{T}}$ (and vice-versa) based on the syntactic structure of v and of type t. As a proof technique the authors adopt step-indexed logical relations [11] instead of contextual equivalence to prove the translation fully abstract. Moreover, the proofs exploit several key properties of typed wrappers: wrapper termination (i.e., wrapper functions are total), cancellation (i.e., a translation from t to $[\![t]\!]^{\mathcal{S}}_{\mathcal{T}}$ and one from $[\![t]\!]^{\mathcal{S}}_{\mathcal{T}}$ to t cancel each other) and parametricity (enabling the usage of wrappers for abstract types).

In subsequent work, Ahmed and Blume proved that a typed CPS translation from the simply-typed λ-calculus to System F is also fully abstract [13]. Instead of using global "answer types" (i.e., the type of the continuation), the typed CPS translation given each continuation its own individually abstract answer type. Consequently, a well-typed function in typed CPS form can only use its continuation, and this prevents "bad" target terms from being well-typed. To prove the translation fully abstract, the authors combine source and target language in a Matthews and Findler-style multilanguage system [79] so that both languages have access to each other's values. Preservation and reflection of contextual equivalence is proven by using step-indexed logical relations.

Barthe *et al.* devised a secure compilation scheme from a WHILE language to a typed, stack-based assembly language [19]. Both languages enjoy information flow type systems, which is the mechanism exploited by the compilation scheme to be secure. Information-flow type systems assign a security label l or h to insecure and secure values respectively; they enforce a non-interference property: h-level values do not influence l-level output [117]. The secure compilation scheme produces target code that is well typed and therefore enjoys the non-interference properties of the source code, thus making the translation security types-preserving. Since the security properties of the source language stem only from the type system, the compilation is secure. Subsequently, the authors extended their secure compilation results to a Java-like concurrent setting, extending both source and target languages with thread creation [20]. The compilation scheme exploits the typing information to label its output code as being either high or low security. Then, this information is fed into a secure scheduler, which uses it to ensure that the interleaving of observable events may not depend on sensitive data. Together, the compiler and the scheduler prevent internal timing leaks performed by an attacker with access to low security variables.

League *et al.* developed a secure compilation scheme from Featherweight Java (FJ, [98]) to F_ω that exploits the latter's higher order type system (extended with ordered records, fixed-point functionality, recursive types, existential types and row polymorphism) [73]. The compilation scheme translates each FJ class into an F_ω term where fields are collected in one record and methods are collected in a separate record which represents a virtual method table shared by all instances of the class. In this type-preserving translation, compiled F_ω terms preserve the typing information of their source level counterparts. The type system of FJ is not the only security mechanism, since classes can have **`private`** fields that are securely compiled due to the adoption of existential types.

10.3.2 Cryptography-Based

This section describes works which devise compilers for distributed and concurrent languages, which are subject to a Dolev-Yao attacker. These works achieve secure compilation by exploiting target-level cryptographic primitives to protect messages exchanged between target-level processes.

Abadi, Fournet and Gonthier extensively studied the application of cryptographic primitives to securely compile inter-process, message passing-based communication, both in concurrent and distributed settings [2, 3, 4]. The authors adopt source languages that are variations of the Join calculus: a model of concurrency where processes send and receive messages on channels, which are treated as first-class citizens. The Join calculus is reminiscent of the π-calculus [85, 106]; they are also equivalently expressive (up to weak barbed congruence [43]). The target language chosen in their works is the SJoin calculus: an extension of the Join calculus with security primitives for encryption ($\{M\}_k$) and decryption ($\texttt{case}\ L\ \texttt{of}\ \{M\}_k\ \texttt{in}\ P$) of message M with key k.

In their first work, the authors presented a fully abstract compilation scheme for processes which are given secure local and global communication primitives [3]. Here, translated processes are wrapped in a “firewall” process that (i) maintains key pairs for cryptographic primitives and (ii) transforms communication on global channels into security protocols employing those primitives. The translation is proven to preserve and reflect weak bisimulation, which is what frequently replaces contextual equivalence for concurrent calculi.

Subsequently, the authors developed a secure compilation scheme for the Join calculus extended with support for principals, so that the calculus has authentication primitives [4]. In the target code, principals are translated into key pairs that are used to generate unforgeable certificates that prove a principal’s identity. The translation is proven to preserve and reflect weak bisimulation, but only in the presence of noise in the network (i.e., enough encrypted messages) to prevent traffic analysis.

Moreover, relying on similar techniques, the authors provided a secure compilation scheme for the Join calculus extended with constructs to create secure channels [2]. In this translation, target processes are given a cryptographic key for each communication channel they define and they are placed behind a “firewall” that keeps track of key usage. Communication is translated into the execution of cryptographic protocols which use nonces and other techniques to thwart different kind of attacks. Full abstraction of the translation is again proven to preserve and reflect weak bisimulation and it again relies on the presence of noise in the network.

This line of work was further expanded by Bugliesi and Giunti, who provided a secure compilation scheme from a dynamically-typed π-calculus to the applied Spi-calculus which relies on cryptographic operations to secure channel communication [24]. The Spi-calculus is the extension of the π-calculus with cryptographic primitives much as the SJoin calculus is to the Join calculus. The usage of the π-calculus (as opposed to the Join calculus of previous works by Abadi, Fournet and Gothier), allows forward secrecy attacks to be modelled (i.e., the logging of encrypted data to be decrypted in the future). This is because in the π-calculus a communicated channel can also be used to perform input, while this is not possible in the Join calculus. This work presents a translation that protects against these attacks by extending translated processes with self-signed certificates and a proxy server. Self-signed certificates are the target-level implementation of source-level channels; these certificates include the channel identity and two encryption keys corresponding to the input and output capabilities. The proxy server keeps track of cryptographic keys related to channels, to preserve the expected interactions between processes. Also in this case, full abstraction of the compilation scheme is proven to preserve and reflect weak bisimulation.

Techniques similar to those employed by Abadi, Fournet and Gonthier were also employed by Adão and Fournet, who developed a secure compilation scheme for a π-calculus extended with secure channels, mobile names and high-level certificates [9]. The characteristic of their work is the target language and the adversary model. The target language is a set of machines that have input and output network interfaces and can perform cryptographic operations. The adversary is modelled as a probabilistic algorithm that controls that network and some corrupted machines. The compilation scheme is proven secure by showing that it preserves and reflects weak bisimulation.

The work of Laud has also exploited encrypted and signed messages to securely compile ABS into the applied π-calculus extended with cryptographic operations [72]. ABS is a concurrent, object-oriented language with asynchronous method calls and futures [66]. The compilation scheme translates each asynchronous method call into an explicit message which is uniquely identified by means of a fresh cryptographic key. Compiled objects are also uniquely identified by means of fresh keys associated to them. The semantics of both the source and the target languages are given in terms of LTSs, where attackers are modelled as other LTSs that synchronise on visible actions. Full abstraction of the translation is proven to preserve and reflect weak bisimulation defined on these LTSs.

Duggan provided a secure compilation scheme from a π-calculus-like language with cryptographic types and principals to the Spi-calculus [39]. Cryptographic types express cryptographic guarantees on values at the type system level, since types mention that certain values are encrypted or signed by certain principals. The information regarding principals is hidden for communication over insecure medium, while it is exposed when the medium is trusted. The type system performs static checks to ensure that values are used in the expected manner, but it also performs dynamic checks on cryptographic-typed values when principals data is unavailable. The compilation scheme inserts cryptographic operations only when a dynamic check is required thus minimising the computational overhead introduced by cryptography. As in previously-mentioned works, principals are translated to pairs of keys which are used for encryption and signing of network messages. Here, contextual equivalence is defined to be a weak bisimulation; the work then proves the presented translation to be fully abstract.

Session types ensure that distributed parties comply to a protocol; the latter is encoded as a session: a sequence of actions detailing what message is communicated between various peers [26]. Corin *et al.* presented a secure compiler from $F^{\#}$ extended with session types to $F^{\#}$ extended with libraries providing cryptographic primitives [30]. The secure compiler produces code that is shielded from the attempts of other peers to deviate from their session by exploiting the cryptographic primitives of the target language. The compiler does not introduce additional messages but it maps each session action to a cryptographic message between the same sender and receiver. Each cryptographic message contains a unique session identifier and the signatures of the sender and of the senders of the previous messages, so it can be uniquely identified. Any attempt to tamper with the integrity of the session can thus be detected and any such message can be dropped. The secure compiler is proven fully abstract by adopting a labelled operational semantics which makes explicit the communication between secure code and a potential attacker.

Focussing on distributed languages, Fournet *et al.* [44], presented a fully abstract compilation scheme for a distributed WHILE language featuring a type system with security levels. In this case, the target language is a WHILE language extended with cryptographic libraries and with threads that reside at different locations. The compiler performs four passes of the source code in order to generate secure target code. Firstly, source code with location annotations is sliced into local programs, each meant to run in a different location. Secondly, each local program is extended with global variables to keep track of its state. Then, each global variable is given local replicas and additional functionality for explicit updates between these replicas. Finally, global variable updates are protected with cryptographic operations and the keys that regulate these operations are disseminated to the threads.

Another class of distributed languages is multi-tier languages, which are adopted to develop web applications split into several tiers (i.e., client, server, database, etc) that can reside in different machines. Baltopoulos and Gordon [18] described a secure compilation scheme for the multi-tier language TINYLINKS into $F7$, an ML dialect extended with refinement types. The secure compiler is proven to preserve data integrity and control integrity properties of well-typed source programs in the generated target programs by exploiting authenticated encryption mechanisms. Malicious attackers are modelled as untyped contexts which also have power over the network connecting the different tiers. The compiler is secure because it is proven to translate well-typed programs into robustly-safe $F7$ expressions. These expressions are a subset of $F7$ expressions which are immune to the aforementioned attacks when interoperating with any attacker.

10.3.3 Memory Protection Mechanisms-Based

Memory-related attacks have resulted a large body of research on memory protection mechanisms.

Abadi and Gordon adopted ASLR to achieve (probabilistic) fully abstract compilation in a λ-calculus setting [6]. Their source language is a simply-typed λ-calculus extended with an abstract memory: a mapping from locations to values; locations can be public or private. Their target language is a λ-calculus extended with a concrete memory: a mapping from natural numbers (in this case they assume an unbounded memory) to values. Abadi and Gordon prove that with a large enough memory ASLR ensures that an attacker operating at the target level has a negligible chance of guessing values, thus achieving probabilistic full abstraction.

Subsequently, Jagadeesan *et al.* [62] extended this secure compilation scheme to a source language with more complex features: dynamic memory allocation, higher-order references and call-with-current-continuation. The source language is $\lambda\mu\mathsf{hashref}$-calculus, a λ-calculus extended with operations for testing the hash of a reference, and the target language is the $\lambda\mu\mathsf{probref}$-calculus, a λ-calculus with the ability to reverse the hash of a reference. Reversing a hash succeeds when a reference is known, but it is complex when the reference is unknown due to the large memory layout and the random allocation of references in memory. To prove full abstraction of the translation, the authors develop LTSs for both $\lambda\mu\mathsf{hashref}$-calculus and $\lambda\mu\mathsf{probref}$-calculus which yield trace semantics which is then used to guide the proofs.

Agten *et al.* were the first to present a fully abstract compilation scheme that use PMA to preserve confidentiality and integrity properties of their source languages [10]. The authors devised a secure compilation scheme for a language with objects, interfaces and first-class method references to an assembly language extended with PMA. The compilation scheme places the objects to be secured in the PMA protected memory partition. Then, it creates entry points for methods appearing in interfaces, so that external code can call them. Secure methods activation records are allocated on a secure stack that resides within the protected memory section. Dynamic checks are introduced for all values communicated to and from the unprotected section in order to prevent ill-formed values affecting the computation. For example, primitive-typed values are checked to be inhabitants of that type, i.e., a **`bool`** value is checked to be one of two predefined values: the compiled versions of `true` and `false`.

Patrignani *et al.* expanded the work of Agten *et al.* to a source language with dynamic memory allocation, exceptions and inner classes [97]. This source language is an extension of Java Jr. [65]: a Java-like object-oriented language that provides strong encapsulation of classes and objects, which are not visible outside the package that defines them. Moreover, packages communicate based on exported interfaces and exported objects. With the introduction of class types, the secure compilation scheme introduces more dynamic checks. Objects are checked to define the method they are called on, which is possible as objects are allocated in the secure memory partition alongside their type information (which is also used for dynamic dispatch). Similar checks are performed on parameters whose type is a securely-defined class. Moreover, the identities of objects passed to the unprotected code are replaced with natural numbers, so as to obscure the allocation strategy of objects in the protected memory section and prevent related attacks. Whenever an exception is thrown from unprotected code, it is also checked to be among the exceptions that could be thrown (i.e., defined in the method signature) and not a maliciously crafted exception.

When proving full abstraction of the compilation schemes, both works assume the reflection of contextual equivalence claiming that most compilers achieve this. By relying on a fully abstract trace semantics for the target language [95], both works prove the contrapositive of the preservation of contextual equivalence by devising an algorithm that always constructs a context that differentiates between two components that exhibit different target-level traces.

10.3.4 Dynamic Checks Insertion-Based

Some of the presented works add dynamic checks to make their compilation schemes secure, for example Agten *et al.*'s primitive value checks. Those works relied on other mechanisms besides the checks, while this section describes works that achieve secure compilation mainly through the dynamic checks added in the generated target code. It is crucial when adding dynamic checks in the generated target code that the attacker cannot tamper with these checks, which would render them void. All of the works presented below adopt different techniques to protect the inserted checks.

Ghica *et al.* [48] described a fully abstract compilation scheme from a λ-calculus extended with iteration to VHDL digital circuits. Security of compilation is achieved through the addition of a runtime monitor that forces external code communicating with the generated digital circuits to respect the expected communication protocol. The attacker is prevented from tampering with the hardware and thus cannot disrupt the runtime monitor.

Fournet *et al.* [45], used defensive wrappers in concert with other techniques to securely translate f^*, a monomorphic ML-like language with references, to JavaScript, the assembly of the web. The compiler firstly translates f^* terms into JavaScript terms, then proceeds to defensively wrap these terms against the features of JavaScript that can be exploited by a malicious attacker. Defensive wrappers (similar to the dynamic checks introduced by the compilers of Agten *et al.* [10] and Patrignani *et al.* [97]) provide dynamic type checks for the untyped JavaScript code. For the compiler to be secure, it firstly makes a local copy of the implementation of trusted values from the global namespace, to prevent an attacker from redefining these values. Secondly, it exports defensively-wrapped, translated terms into the global namespace to make them globally available. Even though the compiler emits JavaScript code, for the proof of full abstraction the authors employ js*, a JavaScript model in a monadic version of f^*, as target language. This is so they can translate f^* types into js* types in order specify inariants and let automate proofs via F^*. As a proof technique, the authors use a labelled bisimulation called applicative bisimulation instead of contextual equivalence.

10.4 On Fully Abstract Semantics

Full abstraction has been largely studied in two fields: inter-language translation (e.g., compilers) and language semantics. Several results on the former have been presented in Section 10.3. As discussed by Gorla and Nestman [52], fully-abstract translation are interesting when they are used to enforce a language property (e.g., security) and not to show that a translation exists. In fact, recently, Parrow has proven that a fully-abstract translation from a source to a target language is only possible if the target language has strictly more equivalence classes than the latter [93].

This section only focusses on the latter kind of results, fully-abstract language semantics, that are related to this work. In fact, the literature on the subject is enormously vast and spans several decades of research. Full abstraction has been largely studied as a way to formalise the correctness of a denotational semantics with respect to an operational one [99]. It has been studied for different programming languages paradigms, such as the λ-calculus [83] and the π-calculus [63]. The reader interested in full-abstraction results for process algebra is referred to the survey of Parrow [92].

Trace semantics was developed to study the behaviour of concurrent CSP [46] and it has been adopted for describing concurrent and distributed language behaviour [104]. Several works have devised fully abstract trace semantics for functional [6, 49, 62, 69] and object-oriented [65, 120] languages. Abadi and Plotkin [6] developed a fully abstract trace semantics for a λ-calculus with references in order to prove a secure compilation using Address Space Layout Randomisation secure. Jagadeesan *et al.* [62] extended the results of Abadi and Plotkin to a λ-calculus with more advanced language features and equipped that language with a fully abstract trace semantics for secure compilation purposes. While the languages are different, the goal of the trace semantics of these works and of the presented work are analogous, as the trace semantics is used to prove secure compilation results related to the language. Laird [69] presented a fully abstract trace semantics for a functional language with locally declared general references that does not focus on the security aspects of that language. Ghica and Tzevelekos [49] provided a fully abstract trace semantics, with regards to a game operational semantics, of a C-like language that, unlike this work, does not present a protection mechanism. Jeffrey and Rathke [65] provided a fully abstract trace semantics for a core Java-like language that enforces strong encapsulation of objects in packages and of fields in classes. Welsch and Poetzsch-Heffter [120] devised a fully abstract trace-based semantics for class libraries in Java-like languages, focussing on backward compatibility for class libraries instead of security.

10.5 Principals in Related Work

In the programming language community, principal information have been made explicit in programming languages for a different purpose than the one of this thesis. The closest work is that of Zdancewic *et al.* [125], where principals are embedded in a λ-calculus and the execution migrates from the environment of a principal to that of another one. Principals have also been investigated in the information flow setting. Tse and Zdancewic [115] developed a system where the runtime information on principals is used to enforce non-interference properties and safety of declassification for principals. Finally, Chothia *et al.* [29] enforced information-flow properties on a distributed, object-oriented language with multiple principals that relies on cryptographic keys. For the sake of simplicity, JEM does not incorporate advanced type systems that use principal information for the enforcement of information flow-like properties.

Chapter 11

Conclusion and Future Work

Three things fears the wise man:
a moonless night, the storming
sea, and the anger of a quiet man.

Pathrick Rothfuss – The wise
man's fear

This section contains concluding remarks (Section 11.1) on the presented work. Moreover, it discusses trajectories for future work on the topic of secure compilation and security in programming languages (Section 11.2).

11.1 Conclusion

Most prominently, this thesis presents the development of a secure compilation scheme from J+E to A+I. The source language of the compilation scheme is called J+E. J+E is a strongly-typed, single-threaded, component-based, object-oriented language that supports dynamic memory allocation and exceptions. The target language of the compilation scheme is called A+I. A+I is an untyped assembly language enhanced with a protected module architecture – a memory isolation mechanism of emerging processors. To guide the implementation of the secure compilation scheme, this thesis highlighted mistakes that make a naïve compilation scheme not secure and how to correct them. To prove that the compilation scheme is secure, it is proven to be fully abstract, i.e., it preserves and reflects contextual equivalence between source-level components and their compiled counterparts. The connection between full abstraction and security has also been discussed and this thesis provided examples of security properties that can expressed via fully abstract compilation. To devise the proof of fully abstract compilation, this thesis relied on a fully-abstract trace semantics for A+I. It presented two different characterisation of traces for A+I and proved that both are fully abstract, i.e., they are as precise as contextual equivalence. Finally, this thesis discussed the security vulnerabilities that arise when languages with multi-principal support are considered for secure compilation. A secure compiler for multi-principal languages has also been devised and its security has been argued to hold.

The work presented in this thesis could have been carried out in different ways. An improvement over the presented work would be to formalise the $[\![\cdot]\!]^{\mathsf{J+E}}_{\mathsf{A+I}}$ compiler. This would allow us to prove Theorem 6, ensuring that none of the presented countermeasures conflict with source-language features implementation (even though we are quite sure this is not the case). A verified implementation of that compiler formalisation could also be provided, were all the theory presented in this thesis formally proven with a proof assistant such as Coq. However, before going down the proof assistant path, we believe a better proof technique (i.e., without the algorithm) is necessary to simplify the Coq development. Such a better proof technique for proving a compilation scheme fully abstract is currently under development.

11.2 Future Work

A number of future research trajectories can be envisioned from this point, some have also been mentioned throughout the text.

Firstly, no existing work considers how would a secure compiler interact with a garbage collector. Garbage collection (i.e., automated memory management) is a reality in all mainstream programming languages. To bring secure compilation to mainstream usage, it has to support most features that programmers rely upon: garbage collection is clearly one of those.

Secondly, no existing secure compiler targets concurrent untyped assembly language, i.e., an untyped assembly language running on a machine with multiple cores. As stated in Chapter 10, concurrency-related secure compilation has been studied in the distributed setting but only in a message-passing based model for concurrency or in a concurrent typed assembly setting. However, concurrent untyped assembly language is a reality in modern multi-core machines. Supporting such a target language seems necessary to bring secure compilation to mainstream audiences.

Having worked with *PMA*, one wonders what can be expressed and what can not be expressed with it. The isolation mechanism it brings seems very powerful, but will it allow secure compilation of advanced security policies such as those based on information-flow? In particular, are there security properties that non-interference can capture and that program equivalence (Definition 1) can not? And if so, can they be securely compiled to *PMA*-enhanced assembly? Moreover, will *PMA* support secure compilation of polymophic languages and of dependantly-typed ones? All these points are interesting per-se but we believe that a general result on the expressiveness of *PMA* is also desirable.

As seen in Section 10.1, capability machines are a powerful, promising security architecture apt at being targeted by a secure compiler. In fact, research is already ongoing to develop a secure compiler for existing capability machine implementations, and that research shows that the insights developed in this thesis are applicable outside of the *PMA* architecture. However, no industrial implementation of capability machines exist. So, when developing a secure compilation scheme for them, we (as programming language researchers) can steer the evolution of this security architecture by stating what is desirable and what is not from a programming language perspective. This would (hopefully) result in capability machine implementations that provide building blocks for supporting secure programming.

Bibliography

[1] Abadi, M. Protection in programming-language translations. In *Secure Internet programming*. Springer-Verlag, 1999, pp. 19–34.

[2] Abadi, M., Fournet, C., and Gonthier, G. Secure implementation of channel abstractions. In *Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science* (Washington, DC, USA, 1998), LICS '98, IEEE Computer Society, pp. 105–.

[3] Abadi, M., Fournet, C., and Gonthier, G. Secure communications processing for distributed languages. In *IEEE Symposium on Security and Privacy* (1999), pp. 74–88.

[4] Abadi, M., Fournet, C., and Gonthier, G. Authentication primitives and their compilation. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2000), POPL '00, ACM, pp. 302–315.

[5] Abadi, M., Planul, J., and Plotkin, G. Layout randomization and nondeterminism. *Electron. Notes Theor. Comput. Sci. 298* (Nov. 2013), 29–50.

[6] Abadi, M., and Plotkin, G. On protection by layout randomization. In *CSF '10* (2010), IEEE, pp. 337–351.

[7] Abdullahi, S. E., and Ringwood, G. A. Garbage collecting the internet: A survey of distributed garbage collection. *ACM Comput. Surv. 30*, 3 (Sept. 1998), 330–373.

[8] Abramsky, S. The lazy lambda calculus. In *Research Topics in Functional Programming*, D. A. Turner, Ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990, pp. 65–116.

[9] Adão, P., and Fournet, C. Cryptographically sound implementations for communicating processes. In *Proceedings of the 33rd international conference on Automata, Languages and Programming - Volume Part II* (Berlin, Heidelberg, 2006), ICALP'06, Springer-Verlag, pp. 83–94.

[10] Agten, P., Strackx, R., Jacobs, B., and Piessens, F. Secure compilation to modern processors. In *CSF '12* (2012), IEEE, pp. 171 – 185.

[11] Ahmed, A. Step-indexed syntactic logical relations for recursive and quantified types. In *Proceedings of the 15th European Conference on Programming Languages and Systems* (Berlin, Heidelberg, 2006), ESOP'06, Springer-Verlag, pp. 69–83.

[12] Ahmed, A., and Blume, M. Typed closure conversion preserves observational equivalence. *SIGPLAN Not. 43*, 9 (Sept. 2008), 157–168.

[13] Ahmed, A., and Blume, M. An equivalence-preserving CPS translation via multi-language semantics. *SIGPLAN Not. 46*, 9 (Sept. 2011), 431–444.

[14] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[15] Anati, I., Gueron, S., Johnson, S., and Scarlata, V. Innovative Technology for CPU Based Attestation and Sealing. In *HASP'13* (2013), vol. 13, ACM.

[16] Appel, A. W. *Modern Compiler Implementation in ML: Basic Techniques.* Cambridge University Press, New York, NY, USA, 1997.

[17] Azevedo de Amorim, A., Collins, N., DeHon, A., Demange, D., Hriţcu, C., Pichardie, D., Pierce, B. C., Pollack, R., and Tolmach, A. A verified information-flow architecture. *SIGPLAN Not. 49*, 1 (Jan. 2014), 165–178.

[18] Baltopoulos, I. G., and Gordon, A. D. Secure compilation of a multi-tier web language. In *Proceedings of the 4th international workshop on Types in language design and implementation* (New York, NY, USA, 2009), TLDI '09, ACM, pp. 27–38.

[19] Barthe, G., Rezk, T., and Basu, A. Security types preserving compilation. *Comput. Lang. Syst. Struct. 33*, 2 (July 2007), 35–59.

[20] Barthe, G., Rezk, T., Russo, A., and Sabelfeld, A. Security of multithreaded programs by compilation. *ACM Trans. Inf. Syst. Secur. 13*, 3 (2010), 21:1–21:32.

[21] Belenguer, J., and Calafate, C. A low-cost embedded ids to monitor and prevent man-in-the-middle attacks on wired lan environments. In *Proceedings of the The International Conference on Emerging Security Information, Systems, and Technologies* (Washington, DC, USA, 2007), SECUREWARE '07, IEEE Computer Society, pp. 122–127.

[22] Benton, N., and Hur, C.-K. Biorthogonality, step-indexing and compiler correctness. *SIGPLAN Not. 44*, 9 (Aug. 2009), 97–108.

[23] Benton, N., and Hur, C.-k. Realizability and compositional compiler correctness for a polymorphic language. Tech. rep., MSR, 2010.

[24] Bugliesi, M., and Giunti, M. Secure implementations of typed channel abstractions. In *POPL* (2007), pp. 251–262.

[25] Carter, N. P., Keckler, S. W., and Dally, W. J. Hardware support for fast capability-based addressing. *SIGPLAN Not. 29*, 11 (1994), 319–327.

[26] Castagna, G., Dezani-Ciancaglini, M., Giachino, E., and Padovani, L. Foundations of session types. In *Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming* (New York, NY, USA, 2009), PPDP '09, ACM, pp. 219–230.

[27] Chen, J., Chugh, R., and Swamy, N. Type-preserving compilation of end-to-end verification of security enforcement. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation* (USA, 2010), PLDI '10, ACM, pp. 412–423.

[28] Chlipala, A. A certified type-preserving compiler from lambda calculus to assembly language. *SIGPLAN Not. 42*, 6 (June 2007), 54–65.

[29] Chothia, T., Duggan, D., and Vitek, J. Principals, policies and keys in a secure distributed programming language. In *Foundations of Computer Security - FCS'04* (Turku, Finland, July 2004).

[30] Corin, R., Deniélou, P.-M., Fournet, C., Bhargavan, K., and Leifer, J. A secure compiler for session abstractions. *J. Comput. Secur. 16*, 5 (Dec. 2008), 573–636.

[31] Curien, P.-L. Definability and full abstraction. *Electron. Notes Theor. Comput. Sci. 172* (2007), 301–310.

[32] de Boer, F. S., Bonsangue, M. M., Steffen, M., and Ábrahám, E. A fully abstract semantics for UML components. In *FMCO'04* (2005), vol. 3657 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 49–69.

[33] Denning, D. E. A lattice model of secure information flow. *Commun. ACM 19*, 5 (May 1976), 236–243.

[34] Dennis, J. B., and Van Horn, E. C. Programming semantics for multiprogrammed computations. *Commun. ACM 9*, 3 (Mar. 1966), 143–155.

[35] Dhawan, U., Hritcu, C., Rubin, R., Vasilakis, N., Chiricescu, S., Smith, J. M., Knight, Jr., T. F., Pierce, B. C., and DeHon, A. Architectural support for software-defined metadata processing. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2015), ASPLOS '15, ACM, pp. 487–502.

[36] Dhawan, U., Vasilakis, N., Rubin, R., Chiricescu, S., Smith, J. M., Knight, Jr., T. F., Pierce, B. C., and DeHon, A. Pump: A programmable unit for metadata processing. In *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy* (New York, NY, USA, 2014), HASP '14, ACM, pp. 8:1–8:8.

[37] D'silva, V., Kroening, D., and Weissenbacher, G. A Survey of Automated Techniques for Formal Software Verification. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems 27* (2008), 1165–1178.

[38] Ducournau, R. Implementing statically typed object-oriented programming languages. *ACM Comput. Surv. 43*, 3 (Apr. 2011), 18:1–18:48.

[39] Duggan, D. Type-based cryptographic operations. *J. Comput. Secur. 12*, 3,4 (May 2004), 485–550.

[40] Eldefrawy, K., Francillon, A., Perito, D., and Tsudik, G. SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust. In *NDSS'12* (2012), pp. 1–15.

[41] Erlingsson, U., Younan, Y., and Piessens, F. Low-level software security by example. In *Handbook of Information and Communication Security*. Springer, Berlin, Heidelberg, 2010, pp. 663–658.

[42] Flanagan, D. *Java in a Nutshell. Deutsche Ausgabe der 2. A.* O'Reilly, 1998.

[43] Fournet, C., and Gonthier, G. The reflexive CHAM and the Join-calculus. In *POPL* (1996), pp. 372–385.

[44] Fournet, C., Le Guernic, G., and Rezk, T. A security-preserving compiler for distributed programs: from information-flow policies to cryptographic mechanisms. In *Proceedings of the 16th ACM conference on Computer and communications security* (New York, NY, USA, 2009), CCS '09, ACM, pp. 432–441.

[45] Fournet, C., Swamy, N., Chen, J., Dagand, P.-E., Strub, P.-Y., and Livshits, B. Fully abstract compilation to javascript. In *POPL '13* (New York, NY, USA, 2013), ACM, pp. 371–384.

[46] Francez, N., Hoare, C., and Roever, W. Semantics of nondeterminism, concurrency and communication. In *Mathematical Foundations of Computer Science 1978*, J. Winkowski, Ed., vol. 64 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1978, pp. 191–200.

[47] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[48] Ghica, D. R., and Al-Zobaidi, Z. Coherent minimisation: Towards efficient tamper-proof compilation. In *5th Interaction and Concurrency Experience Workshop* (2012), ICE 2012.

[49] Ghica, D. R., and Tzevelekos, N. A system-level game semantics. *Electr. Notes Theor. Comput. Sci. 286* (2012), 191–211.

[50] Goguen, J. A., and Meseguer, J. Security policies and security models. In *IEEE Symposium on Security and Privacy* (1982), pp. 11–20.

[51] Goldberg, I., Wagner, D., Thomas, R., and Brewer, E. A. A secure environment for untrusted helper applications confining the wily hacker. In *Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6* (Berkeley, CA, USA, 1996), SSYM'96, USENIX Association, pp. 1–1.

[52] Gorla, D., and Nestman, U. Full abstraction for expressiveness: History, myths and facts. *Math Struct Comp Science* (2014).

[53] Grand, J. K. Attacks on and countermeasures for USB hardware token devices. *Proceedings of the Fifth Nordic Workshop on Secure IT Systems Encouraging Co-operation* (2000), 35–57.

[54] Guillemette, L.-J., and Monnier, S. A type-preserving compiler in haskell. *SIGPLAN Not. 43*, 9 (Sept. 2008), 75–86.

[55] Halderman, J., Schoen, S., Heninger, N., Clarkson, W., Paul, W., Calandrino, J., Feldman, A., Appelbaum, J., and Felten, E. Lest we remember: Cold boot attacks on encryption keys. In *USENIX Security Symposium* (2008), pp. 45–60.

[56] Halderman, J. A., Schoen, S., Heninger, N., Clarkson, W., Paul, W., Calandrino, J., Feldman, A., Appelbaum, J., and Edward, F. Lest we remember: cold-boot attacks on encryption keys. In *UProceedings of the 17th USENIX Security Symposium* (San Jose, CA, USA, 2008), USENIX Association, pp. 45–60.

[57] Halfond, W., and Orso, A. Preventing sql injection attacks using amnesia. In *Proceedings of the 28th international conference on Software engineering* (New York, NY, USA, 2006), ICSE '06, ACM, pp. 795–798.

[58] Hoekstra, M., Lal, R., Pappachan, P., Phegade, V., and Del Cuvillo, J. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (2013), ACM, p. 11.

[59] Homescu, A., Neisius, S., Larsen, P., Brunthaler, S., and Franz, M. Profile-guided automated software diversity. In *CGO '13* (2013), IEEE Computer Society, pp. 1–11.

[60] Hur, C.-K., and Dreyer, D. A Kripke logical relation between ML and Assembly. *SIGPLAN Not. 46*, 1 (Jan. 2011), 133–146.

[61] Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., and Piessens, F. Verifast: A powerful, sound, predictable, fast verifier for c and java. In *Proceedings of the Third International Conference on NASA Formal Methods* (Berlin, Heidelberg, 2011), NFM'11, Springer-Verlag, pp. 41–55.

[62] Jagadeesan, R., Pitcher, C., Rathke, J., and Riely, J. Local memory via layout randomization. In *Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium* (Washington, DC, USA, 2011), CSF '11, IEEE Computer Society, pp. 161–174.

[63] Jeffrey, A., and Rathke, J. Full abstraction for polymorphic pi-calculus. In *Proceedings of the 8th international conference on Foundations of Software Science and Computation Structures* (Berlin, Heidelberg, 2005), FOSSACS'05, Springer-Verlag, pp. 266–281.

[64] Jeffrey, A., and Rathke, J. A fully abstract may testing semantics for concurrent objects. *Theor. Comput. Sci. 338*, 1-3 (June 2005), 17–63.

[65] Jeffrey, A., and Rathke, J. Java Jr.: Fully abstract trace semantics for a core Java language. In *ESOP'05* (2005), vol. 3444 of *LNCS*, Springer, pp. 423–438.

[66] Johnsen, E. B., Hähnle, R., Schäfer, J., Schlatte, R., and Steffen, M. ABS: A core language for abstract behavioral specification. In *Proceedings of the 9th International Conference on Formal Methods for Components and Objects* (Berlin, Heidelberg, 2011), FMCO'10, Springer-Verlag, pp. 142–164.

[67] Kennedy, A. Securing the .NET programming model. *Theor. Comput. Sci. 364*, 3 (Nov. 2006), 311–317.

[68] Kursawe, Klaus adn Schellekens, D., and Bart, P. Analyzing trusted platform communication. In *ECRYPT Workshop, CRASH - CRyptographic Advances in Secure Hardware* (Leuven, Belgium, 2005).

[69] Laird, J. A fully abstract trace semantics for general references. In *Automata, Languages and Programming*, vol. 4596 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2007, pp. 667–679.

[70] Larmuseau, A., Patrignani, M., and Clarke, D. Operational Semantics for Secure Interoperation. In *Proceedings of the Ninth ACM SIGPLAN Workshop on Programming Languages and Analysis for Security* (2014), PLAS '14, ACM.

[71] Larsen, P., Brunthaler, S., and Franz, M. Security through diversity: Are we there yet? *IEEE Security & Privacy 12*, 2 (2014), 28–35.

[72] Laud, P. Secure implementation of asynchronous method calls and futures. In *Trusted Systems*, C. Mitchell and A. Tomlinson, Eds., vol. 7711 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 25–47.

[73] League, C., Shao, Z., and Trifonov, V. Type-preserving compilation of Featherweight Java. *ACM Transactions on Programming Languages and Systems 24*, 2 (2002), 112–152.

[74] Leroy, X. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL* (2006), pp. 42–54.

[75] Leroy, X. A formally verified compiler back-end. *J. Autom. Reason. 43*, 4 (Dec. 2009), 363–446.

[76] Leroy, X. Keynote ii: Formally verifying a compiler: Why? how? how far? In *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on* (april 2011), p. xxxi.

[77] Lochbihler, A. Verifying a compiler for java threads. In *Proceedings of the 19th European Conference on Programming Languages and Systems* (Berlin, Heidelberg, 2010), ESOP'10, Springer-Verlag, pp. 427–447.

[78] Matthews, J., and Findler, R. B. Operational semantics for multi-language programs. *ACM Trans. Program. Lang. Syst. 31*, 3 (Apr. 2009), 12:1–12:44.

[79] Matthews, J., and Findler, R. B. Operational Semantics for Multi-Language Programs. *ACM Transactions on Programming Languages and Systems 31*, 3 (Apr. 2009), 12:1–12:44.

[80] McCune, J. M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V., and Perrig, A. Trustvisor: Efficient TCB reduction and attestation. In *SP '10* (Washington, DC, USA, 2010), pp. 143–158.

[81] McCune, J. M., Parno, B. J., Perrig, A., Reiter, M. K., and Isozaki, H. Flicker: an execution infrastructure for TCB minimization. *SIGOPS Oper. Syst. Rev. 42*, 4 (2008), 315–328.

[82] McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C. V., Shafi, H., Shanbhogue, V., and Savagaonkar, U. R. Innovative instructions and software model for isolated execution. In *HASP '13* (New York, NY, USA, 2013), ACM, pp. 10:1–10:1.

[83] Milner, R. Fully abstract models of typed *lambda*-calculi. *Theor. Comput. Sci. 4*, 1 (1977), 1–22.

[84] Milner, R. A Theory of Type Polymorphism in Programming. *Journal of computer and system sciences 375* (1978), 348–375.

[85] Milner, R. *Communicating and mobile systems - the Pi-calculus.* Cambridge University Press, 1999.

[86] Mitchell, J. C. On abstraction and the expressive power of programming languages. *Sci. Comput. Program. 21*, 2 (1993), 141–163.

[87] Morrisett, G., Felleisen, M., and Harper, R. Abstract models of memory management. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture* (New York, NY, USA, 1995), FPCA '95, ACM, pp. 66–77.

[88] Morrisett, G., Walker, D., Crary, K., and Glew, N. From system F to typed assembly language. *ACM Trans. Program. Lang. Syst. 21*, 3 (May 1999), 527–568.

[89] Myreen, M. O. Verified just-in-time compiler on x86. *SIGPLAN Not. 45*, 1 (Jan. 2010), 107–118.

[90] Necula, G. C. Proof-carrying code. In *POPL* (1997), pp. 106–119.

[91] Noorman, J., Agten, P., Daniels, W., Strackx, R., Van Herrewege, A., Huygens, C., Preneel, B., Verbauwhede, I., and Piessens, F. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *Proceedings of the 22Nd USENIX Conference on Security* (Berkeley, CA, USA, 2013), SEC'13, USENIX Association, pp. 479–494.

[92] Parrow, J. Expressiveness of process algebras. *Electronic Notes in Theoretical Computer Science 209*, 0 (2008), 173 – 186. Proceedings of the {LIX} Colloquium on Emerging Trends in Concurrency Theory (LIX 2006).

[93] Parrow, J. General conditions for full abstraction. *Math Struct Comp Science* (2014).

[94] Patrignani, M., Agten, P., Strackx, R., Jacobs, B., Clarke, D., and Piessens, F. Secure Compilation to Protected Module Architectures. *ACM Trans. Program. Lang. Syst. 37*, 2 (Apr. 2015), 6:1–6:50.

[95] Patrignani, M., and Clarke, D. Fully Abstract Trace Semantics of Low-level Isolation Mechanisms. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing* (2014), SAC '14, ACM, pp. 1562–1569.

[96] Patrignani, M., and Clarke, D. Fully abstract trace semantics for protected module architectures. *Computer Languages, Systems & Structures 42*, 0 (2015), 22 – 45. Special issue on the Programming Languages track at the 29th {ACM} Symposium on Applied Computing.

[97] Patrignani, M., Clarke, D., and Piessens, F. Secure Compilation of Object-Oriented Components to Protected Module Architectures. In *Proceedings of the 11th Asian Symposium on Programming Languages and Systems (APLAS'13)* (2013), vol. 8301 of *LNCS*, pp. 176–191.

[98] Pierce, B. *Types and Programming Languages.* MIT Press, 2002.

[99] Plotkin, G. D. LCF considered as a programming language. *Theoretical Computer Science 5* (1977), 223–255.

[100] Power, J., and Shkaravska, O. From comodels to coalgebras: State and arrays. *Electron. Notes Theor. Comput. Sci. 106* (Dec. 2004), 297–314.

[101] Riecke, J. G. Fully abstract translations between functional languages. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1991), POPL '91, ACM, pp. 245–254.

[102] Ritter, E., and Pitts, A. M. A fully abstract translation between a lambda-calculus with reference types and Standard ML. In *Proceedings of the Second International Conference on Typed Lambda Calculi and Applications* (London, UK, UK, 1995), TLCA '95, Springer-Verlag, pp. 397–413.

[103] Roemer, R., Buchanan, E., Shacham, H., and Savage, S. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur. 15*, 1 (Mar. 2012), 2:1–2:34.

[104] Sangiorgi, D. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms.* PhD thesis CST–99–93, Department of Computer Science, University of Edinburgh, 1992.

[105] Sangiorgi, D. *Introduction to Bisimulation and Coinduction.* Cambridge University Press, 2012.

[106] Sangiorgi, D., and Walker, D. *The Pi-Calculus - a theory of mobile processes.* Cambridge University Press, 2001.

[107] Sevcik, J., Vafeiadis, V., Zappa Nardelli, F., Jagannathan, S., and Sewell, P. Relaxed-memory concurrency and verified compilation. *SIGPLAN Not. 46*, 1 (Jan. 2011), 43–54.

[108] Sharif, M. I., Lee, W., Cui, W., and Lanzi, A. Secure in-vm monitoring using hardware virtualization. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2009), CCS '09, ACM, pp. 477–487.

[109] Singaravelu, L., Pu, C., Härtig, H., and Helmuth, C. Reducing TCB complexity for security-sensitive applications: three case studies. *SIGOPS Oper. Syst. Rev. 40*, 4 (2006), 161–174.

[110] Strackx, R. *Security Primitives for Protected-Module Architectures Based on Program-Counter-Based Memory Access Control.* PhD thesis, KU Leuven, December 2014.

[111] Strackx, R., Agten, P., Avonds, N., and Piessens, F. Salus: Kernel support for secure process compartments. In *Accepted for publication in Endorsed Transactions on Security and Safety*, EAI.

[112] Strackx, R., and Piessens, F. 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* (New York, NY, USA, 2012), CCS '12, ACM, pp. 2–13.

[113] Strackx, R., Piessens, F., and Preneel, B. Efficient isolation of trusted subsystems in embedded systems. In *SecureComm* (2010), pp. 344–361.

[114] Strackx, R., Younan, Y., Philippaerts, P., Piessens, F., Lachmund, S., and Walter, T. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security* (New York, NY, USA, 2009), EUROSEC '09, ACM, pp. 1–8.

[115] Tse, S., and Zdancewic, S. Run-time principals in information-flow type systems. *ACM Trans. Program. Lang. Syst. 30*, 1 (Nov. 2007).

[116] van Dooren, M., Clarke, D., and Jacobs, B. Subobject-oriented programming. In *Formal Methods for Components and Objects* (2013), vol. 7866 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 38–82.

[117] Volpano, D., Irvine, C., and Smith, G. A sound type system for secure flow analysis. *Journal of Computer Security 4* (1996), 167–187.

[118] Wand, M. The theory of fexprs is trivial. *Lisp and Symbolic Computation 10*, 3 (1998), 189–199.

[119] Watson, R. N. M., Anderson, J., Laurie, B., and Kennaway, K. Capsicum: Practical capabilities for UNIX. In *USENIX Security Symposium* (2010), USENIX Association, pp. 29–46.

[120] Welsch, Y., and Poetzsch-Heffter, A. A fully abstract trace-based semantics for reasoning about backward compatibility of class libraries. *Science of Computer Programming* -, 0 (2013), –.

[121] Winter, J. Eavesdropping Trusted Platform Module Communication. http://embedded.iaik.tugraz.at/downloads/evtpm-paper.pdf, 2009.

[122] Winter, J., and Dietrich, K. A hijacker's guide to the LPC bus. In *Proceedings of the 8th European conference on Public Key Infrastructures, Services, and Applications (EuroPKI'11)* (2012), Springer, pp. 176–193.

[123] Witchel, E., Cates, J., and Asanović, K. Mondrian memory protection. *SIGPLAN Not. 37*, 10 (Oct. 2002), 304–316.

[124] Woodruff, J., Watson, R. N., Chisnall, D., Moore, S. W., Anderson, J., Davis, B., Laurie, B., Neumann, P. G., Norton, R., and Roe, M. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Piscataway, NJ, USA, 2014), ISCA '14, IEEE Press, pp. 457–468.

[125] Zdancewic, S., Grossman, D., and Morrisett, G. Principals in programming languages: a syntactic proof technique. *SIGPLAN Not. 34*, 9 (Sept. 1999), 197–207.

[126] Zdancewic, S. A. *Programming Languages for Information Security*. PhD thesis, Cornell University, 2002.

[127] Zeng, B., Tan, G., and Morrisett, G. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2011), CCS '11, ACM, pp. 29–40.

Curriculum Vitae

Marco Patrignani was born in Ravenna, Italy, in 1986.

He received his bachelor degree from Bologna University in October 2008 and his master degree in Computer Science (Theoretical Curriculum) at Bologna University in July 2010. He developed his master thesis under the supervision of prof. Davide Sangiorgi and prof. Dave Clarke.

From November 2011 to September 2015, he has been a Ph.D. student at KU Leuven at the iMinds-Distrinet research group under the supervision of prof. Dave Clarke and prof. Frank Piessens. Since October 2012, his Ph.D. is funded by the Research Foundation Flanders (FWO).

Publications

Journal Articles

- Patrignani, M., and Clarke, D. Fully Abstract Trace Semantics for Protected Module Architectures. *Computer Languages, Systems & Structures* (2015). Special issue on the Programming Languages track at the 29th ACM Symposium on Applied Computing.

- Patrignani, M., Agten, P., Strackx, R., Jacobs, B., Clarke, D., and Piessens, F. Secure Compilation to Protected Module Architectures. *ACM Trans. Program. Lang. Syst. 37*, 2 (Apr. 2015), 6:1–6:50.

International Conference Articles

- Patrignani, M., and Clarke, D. Fully Abstract Trace Semantics of Low-level Isolation Mechanisms. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing* (2014), SAC '14, ACM, pp. 1562–1569.

- Patrignani, M., Clarke, D., and Piessens, F. Secure Compilation of Object-Oriented Components to Protected Module Architectures. In *Proceedings of the 11th Asian Symposium on Programming Languages and Systems (APLAS'13)* (2013), vol. 8301 of *LNCS*, pp. 176–191.

- Patrignani M., Clarke D., and Sangiorgi D. Ownership Types for the Join Calculus. In *FMOODS/FORTE 2011*, volume 6722 of *LNCS*, pages 289–303, 2011.

Peer-Reviewed International Workshop Articles

- Larmuseau, A., Patrignani, M., and Clarke, D. Operational Semantics for Secure Interoperation. In *Proceedings of the Ninth ACM SIGPLAN Workshop on Programming Languages and Analysis for Security* (2014), PLAS '14, ACM.
- Patrignani M., and Clarke D. Fully Abstract Trace Semantics of Low-level Protection Mechanisms –Extended Abstract–. In *Proceedings of the 24th Nordic Workshop on Programming Theory*, NWPT 2012, pages 43–45, 2012.
- Patrignani M., Matthys N., Proença J., Hughes D., and Clarke D. Formal Analysis of Policies in Wireless Sensor Network Application. In *Proceedings of the 3rd international Workshop on Software Engineering for Sensor Network Applications*, SESENA 2012, pages 21–28, 2012.

Technical Reports

- Patrignani, M., Clarke, D., and Piessens, F. Secure Compilation of Object-Oriented Components to Protected Module Architectures – Extended Version. CW Reports CW646, Dept. of Computer Science, K.U.Leuven, September 2013.
- Patrignani M., and Clarke D. Fully Abstract Trace Semantics for Low-level Isolation Mechanisms – Extended version. CW Reports CW651, Dept. of Computer Science, K.U.Leuven, November 2013.
- Patrignani M., Clarke D., and Sangiorgi D. Ownership types for the Join calculus. CW Reports CW603, Dept. of Computer Science, K.U.Leuven, March 2011.

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
IMINDS-DISTRINET
Celestijnenlaan 200A
B-3001 Heverlee
first.last@cs.kuleuven.be
http://people.cs.kuleuven.be/ marco.patrignani/Home.html