Adriaan Larmuseau

Protecting Functional Programs
From Low-Level Attackers
Abstract


Software systems are growing ever larger. Early software systems were singular units developed by small teams of programmers writing in the same programming language. Modern software systems, on the other hand, consist of numerous interoperating components written by different teams and in different programming languages. While this more modular and diversified approach to software development has enabled us to build ever larger and more complex software systems, it has, however, made it harder to ensure the reliability and security of software systems.

In this thesis we study and remedy the security flaws that arise when attempting to resolve the difference in abstractions between components written in high-level functional programming languages and components written in imperative low-level programming languages. High-level functional programming languages, treat computation as the evaluation of mathematical functions. Low-level imperative programming languages, on the contrary, provide programmers with features that enable them to directly interact with the underlying hardware. While these features help programmers write more efficient software, they also make it easy to write malware through techniques such as buffer overflows and return oriented programming.

Concretely, we develop new run-time approaches for protecting components written in functional programming languages from malicious components written in low-level programming languages by making using of an emerging memory isolation mechanism. This memory isolation mechanism is called the Protected Module Architecture (PMA). Informally, PMA isolates the code and data that reside within a certain area of memory by restricting access to that area based on the location of the program counter.

We develop these run-time protection techniques that make use of PMA for three important areas where components written in functional programming languages are threatened by malicious low-level components: foreign function interfaces, abstract machines and compilation. In everyone of these three areas, we formally prove that our run-time protection techniques are indeed secure. In addition to that we also provide implementations of our ideas through a fully functional compiler and a well-performing abstract machine.

Keywords: Security, Functional Programming, Compilation, Interoperation, Bisimulation, Memory Protection

Adriaan Larmuseau, Department of Information Technology, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

© Adriaan Larmuseau 2016

ISSN 1104-2516
urn:nbn:se:uu:diva-281318 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-281318)
Security is not a feature, it’s a process.
Acknowledgements

First and foremost I would like to thank my supervisor Dave Clarke, without whom this thesis and the associated research would not exist. The fact that this research was never derailed or delayed by a move between countries and universities is a testament to your commitment to mentoring me.

I would like to thank Marco Patrignani for all our collaborations, arguments, research visits and continuing discussions. It has been a delight to spitball ideas with you.

Many others have contributed directly and indirectly to my research efforts. Specifically I’d like to thank my Uppsala co-advisor Tobias Wrigstad, my KULeuven co-advisor Frank Piessens, my colleagues Raoul Strackx and Job Noorman for providing me with valuable technical advise, and Deian Stefan for highlighting alternative approaches.

Thank you Rula Sayaf for being an awesome and endlessly entertaining office mate. Never change. I’d also like to thank my Uppsala colleagues Albert Mingkun Yang, Francisco Fernandez Reyes, Stephan Brandauer and Elias Castegren whose Swedish talents make the svenska in this thesis possible. I have of course not forgotten my earlier colleagues at KULeuven, thank you José Proença, Gijs Vanspauwen, Willem Penninckx and Amin Timany. Thanks is also in order to my friends in the departments Gender Equality Group: Virginia Grande, Andreína Francisco, Anne-Kathrin Peters, Per Mattsson, Karolina Holmgrem, Anna-Lena Forsberg and Åsa Cajander.

My deepest gratitude goes to my friends and family. In particular my parents Lieven and Lieve, my sisters Paulien and Elise, my brother Niels, my long-time buddy Bjorn, and of course Zhouqian, I really couldn’t have done this without you.
Mjukvara håller sakta men säkert på att ta över världen. Innovationstakten inom mjukvarusystem och informationsteknik har inte bara vida överstigit innovationstakten inom områden som energi, medicin och transport. Mjukvara kommer även att dominera många av dessa områden i framtiden genom smarta elnät, självkörande bilar, robotkirurgi, etc.

Samtidigt som mjukvarusystem blir allt mer relevanta för samhället, så växer de även i storlek. Tidiga mjukvarusystem var sammanhållna enheter utvecklade av små grupper av programmare som alla skrev i samma programspråk. Moderna mjukvarusystem däremot består av ett stort antal samverkande komponenter skrivna av olika utvecklingsgrupper i olika programspråk. Även om det här mer modulära och mångsidiga sättet att utveckla programvara låter oss bygga allt större och mer komplexa mjukvarusystem, så har det också blivit svårare att garantera att mjukvaran är pålitlig och säker. Ju fler komponenter ett mjukvarusystem består av, desto svårare blir det att upprätthålla kompatibiliteten mellan olika komponenter, hitta orsakerna till buggar och undvika säkerhetsbrister.

I den här avhandlingen studerar och avhjälper vi säkerhetsbristerna som uppstår när man överbrygger abstraktionsskillnaden mellan funktionella högnivåspråk och imperativa lågnivåspråk. Funktionella högnivåspråk hanterar beräkningar som uträkning av matematiska funktioner, vilket minimerar sidoeffekter såsom tillståndsförändringar. Dessa programspråk tillhandahåller bekväma abstraktioner, som typsystem, anonyma funktioner, modulsystem, och så vidare, som inte bara gör det lättare för programmerare att specifisera sina program, utan även möjliggör vissa säkerhetsgarantier. Typsystem, till exempel, hjälper inte bara programmerare att undvika fel, utan kan även användas för att gömma information, upprätthålla invarianter och begränsa möjliga inmatningar till ett system.

Imperativa lågnivåspråk, å andra sidan, struktureras kring ett programs tillstånd och erbjuder funktionalitet som låter programmerare utveckla mjukvara som interagerar direkt med hårdvaran. Även om denna funktionalitet låter programmerare skriva mer effektiv mjukvara, så ger de inte många möjligheter för programmerare att gömma information eller att garantera ett programs säkerhet. Den här bristen på säkerhetsegenskaper hos lågnivåspråk gör att komponenter som skrivs i dessa språk utgör en säkerhetsrisk för komponenter skrivna i funktionella högnivåspråk på två sätt. För det första kan en illvillig aktör skriva skadliga lågniväskomponenter som extraherar konfidentiell information.
ur högnivåkomponenter eller på annat sätt bryter deras säkerhet. För det andra gör de begränsade säkerhetsegenskaperna hos lågnivåspråk att komponenter skrivna i dessa språk blir enklare mål för skadliga program som använder tekniker som buffertöverskridningar eller returorienterad programmering.

I den här avhandlingen utvecklar vi nya sätt att under köring skydda komponenter skrivna i funktionella programspråk mot skadliga komponenter skrivna i lågnivåspråk. Vi gör detta genom att utnyttja en nyligen utvecklad mekanism för minnesisolerings kallad “Protected Module Architecture” (PMA). Informellt kan man säga att PMA isolerar kod och data i en viss del av minnet genom att begränsa tillgången till detta minne beroende på programräknarens värde. Eftersom PMA stöds av Intel i deras senaste generation av processorer så anser vi att detta är en rimlig byggnatest i vårt utvecklande av nya säkerhetstekniker. Mer konkret så inför vi tekniker för skydd under köring som använder PMA i tre viktiga områden där komponenter skrivna i funktionella språk hotas av skadliga lågnivåkomponenter: gränssnitt mot utomstående programspråk, abstrakta maskiner och kompilering.

Ett gränssnitt mot ett utomstående programspråk (FFI, eng. foreign function interface) låter två olika programspråk interagera med varandra genom att specificera hur funktioner kan anropas över språkgränsen, samt hur utbyte av datastrukturer får ske. Ett gränssnitt mellan ett funktionellt högnivåspråk och ett imperativt lågnivåspråk innebär en explicit säkerhetsrisk för funktionella program som använder det, på grund av de tidigare nämnda möjligheterna att införa skadlig mjukvara i lågnivåkomponenten. I den här avhandlingen visar vi ett gränssnitt mellan ett enkelt funktionellt programspråk och ett motsvarande lågnivåspråk som helt tar bort lågnivåspråkets möjlighet till skadlig inverkan.

Abstrakta maskiner är både teoretiska modeller som används för att studera språkegenskaper och praktiska modeller av hur språk är implementerade. Bortsett från misstag i implementenationen, så hotas abstrakta maskiner av den lågnivåmiljö som den befinner sig i. I den här avhandlingen visar vi häradningen och implementenationen av en abstrakt maskin för ett enkelt funktionellt programspråk. Vi bevisar formellt att maskinen garanterar att program skrivna i det funktionella språket hålls i säkert förvar inom minne skyddat av PMA.

En kompilator är ett program som översätter ett program från ett programspråk till ett annat. För att uppnå prestanda eller portabilitet väljer programmerare ofta kompilatorer som kompilerar funktionella högnivåspråk till lågnivåspråk. Men, som nämnts tidigare, så har dessa lågnivåspråk mycket svagare säkerhetsgarantier än deras motparter på högre nivå. I den här avhandlingen visar vi en kompileringsprocess där ett enkelt funktionellt språk med ett tillhörande modulsystem kompileras till ett lågnivåspråk vars kod skyddas av PMA.

I alla tre områden som behandlas i den här avhandlingen bevisar vi formellt att våra skyddsmechanismer verkligen är säkra. Vår formalisering av säkerhet centreras kring begreppet helt abstrakt översättning (eng. fully abstract
Informellt kan man säga att våra skyddsmekanismer är säkra om två funktionella program som inte går att särskilja inte heller kan särskiljas av någon lågniväkomponent. Dessutom kommer alla kompilerade lågniväkomponenter som inte går att särskilja härstamma från funktionella program som inte heller går att särskilja. Man säger att två program inte går att särskilja när det inte finns något tredje program, ofta kallat en kontext, som kan kan observera någon skillnad i deras beteenden.

För att utöka våra teoretiska resultat till vardagligt användande av funktionell programmering innehåller den här avhandlingen inte bara formaliseringar och bevis, utan även implementationer av våra idéer genom en fullt fungerande kompilator och en väl fungerande abstrakt maskin. Vi förväntar oss inte att våra tekniker kommer att bli vardagsmat inom den närmsta framtiden, trots att dessa implementationer finns tillgängliga. Säkerheten som våra skyddsmekanismer under köring tillhandahåller medför också en prestandaförlust. Det är inte troligt att mjukvaruindustrin kommer att gå med på denna prestandakostnad förrän den sortens attacker som våra skyddsmekanismer förhindrar har blivit ett vanligt och allvarligt problem. Men ändå, allttaforsom PMA blir både snabbare och mer tillgängligt i olika sorters hårdvara finns det potential för att våra tekniker i framtiden kommer att kunna ansluta sig till mjukvarans pågående övertagande av världen.
Contents

1 Introduction .......................................................................................... 17
  1.1 Contributions .................................................................................. 21
  1.2 Publications ................................................................................... 22
  1.3 Outline ............................................................................................ 23

2 Background .......................................................................................... 26
  2.1 MiniML ............................................................................................ 26
  2.2 The Source Language ModuleML ..................................................... 29
     2.2.1 The ModuleML Module System ............................................... 30
     2.2.2 The Core Language ................................................................. 36
     2.2.3 ModuleML programs ............................................................... 37
  2.3 Contextual Equivalence .................................................................... 38
     2.3.1 Contextual Equivalence ........................................................... 38
     2.3.2 Contextual Equivalence as a Security Property ...................... 40
  2.4 Fully Abstract Translation ................................................................ 42
  2.5 Bisimilarity and Trace Equivalence ................................................ 43
     2.5.1 Bisimilarity ............................................................................. 43
     2.5.2 Trace Equivalence .................................................................. 45
  2.6 The Protected Module Architecture (PMA) ..................................... 45
  2.7 The Low-Level Attacker Model A+l ................................................ 46

3 A Secure Foreign Function Interface ....................................................... 50
  3.1 Contextual Equivalence for MiniML ................................................ 51
     3.1.1 Examples of Contextual Equivalence ...................................... 51
  3.2 The Attacker Model MiniML\textsuperscript{a} ......................................... 52
     3.2.1 Lack of Contextual Equivalence ............................................. 55
  3.3 Multi-Language Semantics and its Limitations .................................. 55
     3.3.1 \textit{MiniML}\textsuperscript{M}: A natural embedding of MiniML and MiniML\textsuperscript{a} ................. 56
     3.3.2 Why PMA can’t by lifted into \textit{MiniML}\textsuperscript{M} ......................... 56
  3.4 The \textit{MiniML}\textsuperscript{+}-Calculus: a Secure FFI .......................... 58
     3.4.1 Our Approach ....................................................................... 58
     3.4.2 The Syntax of \textit{MiniML}\textsuperscript{+} .......................................... 60
     3.4.3 Operational Semantics ......................................................... 61
     3.4.4 Type Soundness .................................................................... 67
     3.4.5 Contextual Equivalence ....................................................... 69
  3.5 A Fully Abstract Translation ............................................................ 70
C.1 Preservation of the High-Level Attacker ............................................. 193
C.2 Reflection of the Low-Level Attacker ................................................ 203
1. Introduction

Software is slowly but surely consuming the world [8]. The rate of innovation in software and information technology has not just vastly surpassed the innovation in areas such as energy, medicine and transportation [71], software will also come to dominate many of those same areas in the future through smart grids, self-driving cars, robot-assisted surgery, et cetera.

As the relevance of software systems has grown, so to has their size. Early software systems were singular units developed by small teams of programmers writing in the same programming language. Modern software systems, on the other hand, consist of numerous interoperating components written by different teams and in different programming languages. While this more modular and diversified approach to software development has enabled us to build ever larger and more complex software systems, it has, however, made it harder to ensure the reliability and security of software systems. The more components a software system consists of, the harder it becomes to maintain compatibility between the different components, identify the causes of bugs and avoid the introduction of security flaws.

Having components written in different programming languages, as many modern software systems now do, makes it even harder to maintain the compatibility between different components. Programming languages do not just differ in the symbols that they use, they often make use of different abstractions; meaning that different programming languages can differ in the manner in which they manage the complexity of software. Resolving these differences cannot only be difficult, if not done correctly it can also introduce additional security flaws into the software system.

In this thesis we study and remedy the security flaws that arise when attempting to resolve the difference in abstractions between high-level functional programming languages and imperative low-level programming languages. High-level functional programming languages such as ML or Haskell, treat computation as the evaluation of mathematical functions, minimizing side-effects such as statefulness. These programming languages provide programmers with convenient abstractions such as type systems, anonymous functions (also known as lambda terms), module systems and so forth. These abstractions do not just make it easier for programmers to specify their programs, they also provide the programmers with various security features. Type systems, for example, do not just help programmers avoid errors they can also be used to hide information, enforce invariants and constrain possible inputs.

Low-level imperative programming languages such as C or Assembly, on the contrary, are structured around program state and provide programmers
with features that enable the development of software that directly interacts with the underlying hardware. While these features help programmers write more efficient software, they do not provide programmers with many opportunities to hide information or to preserve program integrity.

This lack of security features in low-level programming languages make components written in these languages a security threat to other components written in high-level functional languages in two ways. First, this difference enables malicious actors to write malicious low-level components that extract confidential information from high-level components or break their integrity [1]. Secondly, the more limited security constraints of low-level programming languages make components written in them easier targets for malware injection through techniques such as buffer overflows and return oriented programming [58].

There exist two different categories of approaches that address the threat that low-level components pose to multi-component software systems. The first kind are static approaches such as: software component verification [20], proof carrying code [63] and multi-language type systems [29, 30]. While these static approaches are often highly efficient, they only succeed in scenarios where all of a system’s components are known beforehand and where each component adheres the static approach. However, there are many software systems where this is not the case. Some software systems dynamically link-in new components at run-time, others link with proprietary third-party components that do not support any of these static approaches.

The second kind of approach to dealing with the threats by low-level components are run-time approaches such as: software monitoring [78, 82], sandboxing [9] and memory isolation [80, 66]. These run-time approaches overcome the limitations of the static approaches as they do not need to know all of the involved components ahead of time nor do they need all of the components in the system to willingly participate. This advantage, however, often comes with a substantial cost to the run-time performance of the system.

In this thesis we develop new run-time approaches for protecting components written in functional programming languages from malicious components written in low-level programming languages by making using of an emerging memory isolation mechanism. This memory isolation mechanism is called the Protected Module Architecture (PMA) [21, 57, 80]. Informally, PMA isolates the code and data that reside within a certain area of memory by restricting access to that area based on the location of the program counter (we provide a more complete and technical explanation in Section 2.6). We believe this to be an appropriate building block for the development of new run-time protection techniques for components written in functional programming languages as Intel supports PMA in its new generation of processors, through the Intel SGX instruction set.\(^1\)

\(^1\)https://software.intel.com/en-us/isa-extensions/intel-sgx
We introduce run-time protection techniques that make use of PMA for three important areas where components written in functional programming languages are threatened by malicious low-level components: foreign function interfaces, abstract machines and compilation. A foreign function interface (FFI) enables two different programming languages to interoperate with each other by specifying how functions are to be called across the language barrier as well as how data structures are to be exchanged. An FFI between a high-level functional programming language and a low-level imperative programming language introduces an explicit security risk to functional programs that make use of it. As mentioned earlier, the lack of security relevant constraints in low-level programming languages give them a concrete malicious advantage in their interactions with high-level functional programs.

High-level functional programming language, for example, provide programmers with a very abstract notion of memory. This very abstract notion of memory constrains the access to memory locations, only program terms that are explicitly given access to a memory location can read and write to it. In low-level programming languages, in contrast, memory is very concrete, the only constraint to memory access being the underlying hardware and operating system. When a functional program interacts with low-level code, that low-level code is thus able to bypass all of the memory security properties that functional programming languages provide.

In this thesis we present a foreign function interface between a simple functional programming language and a low-level counterpart that eliminates the malicious advantage of the low-level counterpart and formally prove that this foreign function interface is indeed secure.

Abstract machines are both theoretical models used to study language properties and practical models of language implementations. Nowadays, several languages, especially functional ones, are implemented using abstract machines. For example, Scheme-based languages run on SECD and CESK machines, OCaml’s bytecode runs on the Zinc abstract machine [47] and the Glasgow Haskell Compiler uses the Spineless Tagless G-machine [44] internally. Outside of implementation mistakes, abstract machine implementations are threatened by the low-level context in which they reside. In practice, an abstract machine implementation will, like the final executable produced by a compiler, use or interact with various, possibly malicious, low-level components. In this thesis we present the derivation and implementation of an abstract machine for a light-weight functional programming language. We formally prove that this abstract machine secures the programs written in this functional language within the protected memory of PMA without sacrificing its ability to interact with the outside world.

A compiler is piece of software that translates a program from the original programming language that it was written in (called the source language) to a different language (called the target language). Motivated by speed, memory efficiency and portability programmers often choose compilers that
compile high-level functional programming languages to low-level target lan-
guages [14, 51]. However, as mentioned earlier, these low-level languages
have much weaker security features than their high-level functional counter-
parts. The security risks posed by this fact are rarely considered in existing
compilers as it is often assumed that the compiler compiles a whole program,
isolating it from malicious attackers. In practice, however, the final executable
will consist of more than just the program in the functional language, it will be
linked with various, low-level components (such as libraries) that may be writ-
ten with malicious intent or susceptible to code injection attacks [58]. Some
research into this problem has already been performed for high-level imper-
ative programming languages such as Java [66], however, the specific chal-
lenes that arise when compiling a functional programming language are not
considered int those works. In this thesis we present a compilation scheme
that compiles a light-weight functional programming language with a module
system to the protected memory of PMA. This compilation scheme compiles
an input functional program in a way that is formally proven to preserve the
security properties of the input program from malicious low-level components.

In every one of the three areas addressed in this thesis, we formally prove
that our run-time protection techniques are indeed secure. Our formalisation
of security is centered around a notion fully abstract translation. Informally,
our run-time protection techniques for functional programming languages are
secure if two indistinguishable functional programs are indistinguishable to
every possible low-level component. Moreover, indistinguishable low-level
components derive from functional programs. Two programs are said to be
indistinguishable when there exists no third program, often referred to as a
context, that can observe a difference in their behaviours (in Section 2.3 and
Section 2.4 we provide more formal definitions of these notions).

Components written in low-level programming languages pose a security
threat to functional programs precisely because their limited security con-
straints enable them to observe more information about a functional program,
than any other functional program ever could. Formally proving full abstrac-
tion for our run-time protection techniques — that our run-time protection
techniques are able to prevent low-level components from observing more in-
formation from functional programs than any other functional program could
— is thus a valuable security result. Note that this full abstraction result is,
however, a conservative security result. Our run-time protection techniques
only protect functional programs from the additional capabilities of low-level
components, they do not eliminate the security flaws that programmers them-

Throughout this thesis we devise run-time protections techniques for pro-
grams in two different high-level functional programming languages: MiniML
and ModuleML. MiniML is a light-weight ML featuring higher-order func-
tions, references, tuples and recursion. MiniML is the source language of
our secure FFI and also the source language for our secure abstract machine.
ModuleML is another light-weight version of ML that features higher-order functions, sequences, references, recursion and a module system consisting of structures, higher-order functors and signatures. ModuleML is the source language for our secure compiler.

To extend our theoretical results into the everyday usage of functional programming, this thesis provides not only formalisations and proofs but also provides implementations of our ideas through a fully functional compiler and a well-performing abstract machine. We do not expect our techniques to become common place anytime soon even with these readily available implementations. The security provided by our run-time protection techniques does come with a cost to performance. It seems unlikely that the industry will rush to absorb this performance cost before the kinds of attacks that our run-time protection techniques prevent, become an overwhelmingly consistent problem. However, as PMA becomes readily available in commercial hardware and ever faster, there is the promise that our techniques will join software in consuming the world.

1.1 Contributions

This thesis presents four main contributions.

The first contribution is the formalisation of a secure foreign function interface between the source language MiniML, whose programs reside within the protected memory of PMA, and a low-level language such as assembly or C (Chapter 3). This FFI provides a complete formal model for secure language interoperation that considers not just function calls and basic data types but also addresses in detail the exchange of data structures and memory locations between the functional programming language and the low-level language.

The low-level language is not explicitly formalised in the secure FFI. Instead the FFI formalisation of the FFI contains an attacker model that captures all the threats to full abstraction that a component in a low-level component may pose. The second contribution of this thesis is a formal proof of the applicability of this simplified attacker model (Chapter 4).

The third contribution is the derivation and implementation of an abstract machine for MiniML that runs on a processor enhanced with PMA (Chapter 5). To guarantee the security of the abstract machine, we derive the formalisation of the machine through a methodology whose every step is accompanied by formal properties that ensure that the step has been carried out properly.

Our final contribution is a secure compilation scheme from ModuleML to untyped assembly residing within PMA, which is proven to reflect contextual equivalence (Chapter 6). This reflection property ensures that the security properties of ModuleML cannot be violated by any possible low-level component residing outside of the protected memory. To better explain the secure
compilation scheme, we also introduce a new security pattern for compilation, referred to as the Secure Abstract Data Type pattern (Section 6.2). This pattern bundles together some of the techniques applied in previous secure compilation and full abstraction works.

1.2 Publications
The conference and journal papers which are the basis for this thesis are listed below. In most chapters of the thesis various extensions and improvements were made to the published material.

Conference Papers
I A. Larmuseau, M. Patrignani, D. Clarke: Operational Semantics for Secure Interoperation
In Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security, PLAS@ECOOP 2014, pages 40-52, ACM.

II A. Larmuseau, D. Clarke:
Formalizing a Secure Foreign Function Interface

EXTENDED IN:
A. Larmuseau, D. Clarke:
Formalizing a Secure Foreign Function Interface - Extended Version

III A. Larmuseau, M. Patrignani, D. Clarke:
A High-Level Model for an Assembly Language Attacker by Means of Reflection

EXTENDED IN:
A. Larmuseau, D. Clarke:
Modelling an Assembly Attacker by Reflection
Technical report 2015-026, Dept. of IT, Uppsala University, August 2015.

IV A. Larmuseau, M. Patrignani, D. Clarke:
A Secure Compiler for ML Modules
In Programming Languages and Systems - 13th Asian Symposium,
Journal Papers

VI  M. Patrignani, A. Ahmed, A. Larmuseau, D. Clarke:
    Formal Approaches to Secure Compilation
    In ACM Computing Surveys, Currently in revision.

VII A. Larmuseau, M. Patrignani, D. Clarke:
    A High-Level Model for an Assembly Language Attacker by Means of Reflection
    In Formal Aspects of Computing, Invited paper, Currently in submission.

Comments on my participation
For all papers but paper VI, I had the main role in developing the ideas, the formalisation and the implementation as well as writing the papers. In paper VI, I participated in the discussions, research and writing of the paper.

1.3 Outline
This thesis is organised as follows.

Chapter 2: Background
This chapter introduces the basic concepts that the following chapters are based upon. Firstly it formalises MiniML, the source language of the FFI and abstract machine, and ModuleML, an extension of MiniML that is the source
language of the secure compiler. Next this chapter discusses full abstraction and contextual equivalence, providing examples of how they can be used to express security properties such as confidentiality and integrity. Lastly the PMA security architecture and the capabilities of malicious low-level components are detailed.

Chapter 3: A Secure Foreign Function Interface
This chapter introduces a formal model of a secure foreign function interface, referred to as MiniML\textsuperscript{+}, between MiniML programs residing within the protected memory of PMA and MiniML\textsuperscript{a}, a model of malicious low-level components. To establish the security result we prove the existence of a fully abstract translation: an MiniML program set within MiniML\textsuperscript{+} preserves and reflects the equivalences of the original MiniML program. Because direct proofs over contextual equivalence are difficult we develop notions of bisimulation that coincide with contextual equivalence over MiniML and MiniML\textsuperscript{+}. Full abstraction is then established by systematically relating the states of both notions of bisimulation.

Chapter 4: A Reflection Based Model for a Low-Level Attacker
This chapter provides a formal proof that MiniML\textsuperscript{a}, despite being simple to derive and formalise, is an accurate model of a malicious low-level component when focussed on the security concerns captured by full abstraction. The proof technique proceeds as follows: first we develop a notion of bisimulation over the interactions between a malicious low-level component and programs in MiniML residing withing PMA by adopting the labels of a previously developed fully abstract trace semantics for malicious low-level components. Next, we establish our result by proving that the latter bisimulation is a full abstraction of our bisimulation over MiniML\textsuperscript{+} and vice versa.

Chapter 5: A Secure Abstract Machine
This chapter presents the derivation and implementation of an abstract machine for MiniML that runs on a hardware enhanced with PMA. To guarantee the security of the implemented abstract machine, we follow a four step methodology to derive the formalisation of the abstract machine. In the first step MiniML is extended with our secure foreign function interface of Chapter 3. In the second step we replace the foreign function interface with the bisimulation between the malicious low-level components and MiniML programs of Chapter 4. In the third step we apply the syntactic correspondences of Biernacka and Danvy [12] to obtain the formalisation of a CESK machine implementation for MiniML. For each of these syntactic correspondences, which modify the transitions and syntax of MiniML\textsuperscript{+}, we prove that they do not cause the abstract machine to reveal security sensitive information by developing notions of bisimulation over the result of every correspondence. In
the fourth and final step we implement the formalisation of the abstract ma-
chine on hardware enhanced with PMA.

**Chapter 6: A Secure Compiler for ML Modules**

This chapter introduces a secure compilation scheme from ModuleML to un-
typed assembly code residing within the protected memory in PMA. The com-
piler is secure in that it is proven to reflect contextual equivalence. As is com-
mon in secure compilation works that target a realistic low-level target lan-
guage, we assume that preservation holds. Preservation coincides with com-
piler correctness, it establishes that the secure compiler is a correct MiniML
compiler. While we have tested our implementation of the compilation scheme
intensely, we consider formally verifying the implementation of the compiler
a separate research subject.

**Chapter 7: Related Work**

This chapter presents related work in the areas of secure foreign function in-
terface design, simplified attacker models, secure compilation, abstract ma-
chine formalisation techniques and bisimulation for functional programming
languages.

**Chapter 8: Conclusions and Future Work**

This chapter details possible directions for future work and concludes.
2. Background

This chapter introduces the background notions that form the foundations of the work performed in this thesis. Firstly, it presents the source language of our secure FFI and secure abstract machine, MiniML (Section 2.1) and the source language of our secure compilation scheme (Section 2.2). Then, it formally details contextual equivalence and its relation to security properties (Section 2.3), the full abstraction property that we prove (Section 2.4) and the bisimulation relations used for the proofs (Section 2.5). Lastly, this chapter presents the Protect Module Architecture (PMA) (Section 2.6) and the threat model: the capabilities of malicious low-level components that reside outside of the memory protected by PMA (Section 2.7).

2.1 MiniML

The source language MiniML is a light-weight functional programming language: an extension of the typed $\lambda$-calculus [72] that features constants, references, tuples and recursion but excludes polymorphic types. First we present the syntax of MiniML, followed by its reduction rules and type system.

Syntax

The syntax of MiniML is as follows.

Expressions: 
\[ e ::= v \mid x \mid (e_1 \; e_2) \mid \langle e_i \in l..n \rangle \mid e_1 \; \text{cp} \; e_2 \]
\[ \mid e_1 \; \text{op} \; e_2 \mid e.i \mid \text{if} \; e_1 \; \text{e}_2 \; \text{e}_3 \mid \text{index} \; e \]
\[ \mid \text{let} \; x = e_1 \; \text{in} \; e_2 \mid !e \mid \text{ref} \; e \mid \text{exit} \; e \]
\[ \mid \text{letrec} \; x : \tau = e_1 \; \text{in} \; e_2 \mid e_1 ; e_2 \mid \text{fix} \; e \]

Operands: 
\[ \text{op} ::= + \mid - \mid * \]

Comparators: 
\[ \text{cp} ::= < \mid > \mid == \]

Values: 
\[ v ::= \text{unit} \mid l_i \mid \overline{n} \mid b \mid (\lambda x : \tau. e) \mid \langle v_i \in l..n \rangle \]

Booleans: 
\[ b ::= \text{true} \mid \text{false} \]

Types: 
\[ \tau ::= \text{Bool} \mid \text{Int} \mid \text{Unit} \mid \tau_1 \to \tau_2 \mid \text{Ref} \; \tau \]
\[ \mid \langle \tau_i \in l..n \rangle \]

The value $\overline{n}$ indicates the syntactic term representing the number $n$. 

26
The \texttt{letrec} operator is syntactic sugar for a \texttt{let}-term combined with the fixpoint operator \texttt{fix}.

The \texttt{exit} operator immediately terminates execution with a value \( v \), it can be thought of as a simplified exception.

The locations \( l_i \) are an artefact of the dynamic semantics that do not appear in the syntax used by programmers. The locations are tracked at run-time in a location store \( \mu := \emptyset \ | \ \mu, l_i = v \), which is assumed to be an ideal store: it never runs out of space. The locations are slightly non-standard in that they are allocated deterministically, their order encoded by the index \( i: l_1, l_2, \ldots, l_n \).

The term \texttt{index} \( e \), is the main novelty of the calculus, it implements functionality similar to Java’s \texttt{hashCode} method, in that it converts references to numbers. The \texttt{index} term does this by mapping a location to its index: \( l_i \mapsto i \).

The location index \( i \) and the \texttt{index} function that can extract it, are included to enable us to develop a bisimilarity relation over MiniML and the attacker (Section 3.5.1), as without these additions, locations would be too abstract to capture in a labelled semantics.

**Reduction Rules**

In our definition of the reduction rules we make use of Felleisen-and-Hieb-style evaluation contexts that lift the basic reduction steps, those that reduce the sub-terms of a term, to a standard left-to-right call-by-value semantics [24]. These evaluation contexts \( E \) are defined as follows:

\[
E ::= [\cdot] \mid E \ e \mid v \ E \mid E \ op \ e \mid v \ op \ E \mid E \ cp \ e \\
\mid v \ cp \ E \mid \langle \{ i \in 1..j, E, e_{k}^{(j+1..n)} \} \mid \text{index} \ E \rangle \\
\mid \text{let} \ x = E \ \text{in} \ e \mid \text{if} \ E \ \text{true} \ e_{2} \ e_{3} \mid \text{false} \ e_{2} \ e_{3} \mid \text{ref} \ E \mid E.i \mid \text{fix} \ E \\
\mid E := e \mid v := E \mid E \ e \mid \text{if} \ E \ e_{1} \ e_{2} \mid \text{exit} \ E
\]

where \([\cdot]\), represents a hole: an empty sub-term.

The reduction rules of MiniML are of the form:

\[
\mu \ | \ E[e] \rightarrow \mu' \ | \ E'[e']
\]

where \( \mu \) is the run-time location store, \( E \) the current evaluation context and \( e \) the expression being reduced.

\[
\begin{align*}
(\text{If-True}) & \quad (\text{If-False}) \\
\mu \ | \ E[\text{if} \ \text{true} \ e_{2} \ e_{3}] & \rightarrow \mu \ | \ E[e_{2}] & \mu \ | \ E[\text{if} \ \text{false} \ e_{2} \ e_{3}] & \rightarrow \mu \ | \ E[e_{3}] \\
(\text{Application}) & \\
\mu \ | \ E[(\lambda x : \tau.e) \ n] & \rightarrow \mu \ | \ E[e[n/x]]
\end{align*}
\]
The reduction rules are entirely standard except for the Allocation rule which has been modified to compute a new index $i$ from the size of the store and the Exit rule which discards the evaluation context $E$ to terminate with the program with a value $v$. Note that the Letrec-Sugar rule simply translates the letrec construct into a straight-forward combination of let and fix.

Typing Rules
The typing relation is a three-place relation: $\Gamma \vdash e : \tau$, where $e$ is the expression typed $\tau$ and $\Gamma$ is the typing environment formally defined as follows:

$$\Gamma ::= \emptyset \mid \Gamma, l_i : \tau \mid \Gamma, x : \tau$$

The typing rules of MiniML, listed below, are entirely standard.
2.2 The Source Language ModuleML

The source language ModuleML is a light-weight functional programming language that features a higher-order ML-style module system [53, 59]. The formalisation of ModuleML is divided into a two sub-languages: a module language and a core language. The module language is an adaptation the standard ML module system extended with Leroy’s manifest types for signatures [48] (Section 2.2.1). The core language is the previously detailed MiniML language (Section 2.1) extended with paths to the value and type
bindings introduced by the module systems and memory locations without indices (Section 2.2.2).

2.2.1 The ModuleML Module System

The higher-order module system of ModuleML consists of signatures, structures and functors, as illustrated in Figure 2.1.

<table>
<thead>
<tr>
<th>A Signature</th>
<th>B Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>signature S = sig type X = Int type Y val function: X → Y end</td>
<td>module M : S = struct type X = Int type Y = Int val function x = x + 1 end</td>
</tr>
</tbody>
</table>

(A) Signature (B) Structure

module F = functor(A : S) struct val function2 y = (A.function y) end module M' = F(M);

(C) Functor

Figure 2.1. An example of a ModuleML signature, structure and functor

A ModuleML signature is a sequence of signature components that are either value declarations, abstract or manifest type declarations or module declarations. The signature S listed in Figure 2.1, for example, defines a manifest type X that must be implemented as an Int type, an abstract type Y whose implementation is unknown (hence abstract) and a value declaration function that is a function of type X → Y.

A ModuleML structure is a sequence of structure components that are either value bindings, module bindings or type bindings. The structure M listed in Figure 2.1, for example, binds the manifest type X to Int as required, binds the abstract type Y to Int as its prerogative and binds the value function to a simple addition function.

A ModuleML functor can be considered as a parametrized module, a possibly higher-order function mapping modules to modules. The module F, listed in Figure 2.1, is a functor that maps a structure conforming to S to a new structure that consists only of a value binding function2 that applies the value binding function of the functor argument A, accessed through the path A.function, to an argument y. The module M’, for example, is the result of applying the functor F to the structure M.

Syntax
The formal syntax of the module language of ModuleML is as follows.
Paths:  
\[ p ::= X_i \quad \text{Module Identifier} \]
\[ | \quad p.X_i \quad \text{Module Id of a Structure} \]

Module  
\[ m ::= p \quad \text{Paths} \]

Expressions:
\[ | \quad \text{struct } s \text{ end} \quad \text{Module Structure} \]
\[ | \quad (m : M) \quad \text{Type Ascription} \]
\[ | \quad \text{functor}(X_i : M) m \quad \text{Functor with body } m \]
\[ | \quad m_1(m_2) \quad \text{Module Application} \]

Structure Body:
\[ s ::= \varepsilon \quad \text{Empty Body} \]
\[ | \quad c; s \quad \text{Sequence} \]

Structure  
\[ c ::= \text{val } v_i = e \quad \text{Value Binding} \]

Components:
\[ | \quad \text{type } T_i = \tau \quad \text{Type Binding} \]
\[ | \quad X_i = m \quad \text{Module Binding} \]

Programs:  
\[ P ::= \text{struct } s \text{ end } ; ; e \]

Module Types:
\[ M ::= \text{sig } S \text{ end} \quad \text{Signature Type} \]
\[ | \quad \text{Functor}(X_i : M) \to M' \quad \text{Functor Type} \]

Signature Body:
\[ S ::= \varepsilon \quad \text{Empty Signature} \]
\[ | \quad C; S \quad \text{Sequence} \]

Signature  
\[ C ::= \text{val } v_i : \tau \quad \text{Value Declaration} \]

Components:
\[ | \quad \text{type } T_i \quad \text{Abstract Type} \]
\[ | \quad \text{type } T_i = \tau \quad \text{Manifest Type} \]
\[ | \quad \text{module } X_i : M \quad \text{Module Declaration} \]

Note that the identifiers used in this module system: \( v_i, T_i \) and \( X_i \) are bound by the \text{struct, sig} and \text{functor} constructs and can be alphaconverted provided their name part \( v, T \) and \( X \) does not change [48].

To keep the formalisation as simple as possible the module system does not feature standalone signature bindings of the following kind.

\[ \text{signature } S = \text{sig } \ldots \text{end} \]

Introducing these module types as structure components significantly complicates the typing rules [36]. Our secure compiler (Chapter 6) does support simple usage of the \text{signature} construct that can be compiled by duplicating module type expressions. Our compiler also supports simplified module inclusion through an \text{open } X_i instruction not present in the formalisation.

\section*{Reduction Rules}

The dynamic semantics for the ModuleML source language are an adaptation of Leroy’s calculus of Applicative functors [49] to our more generalised and generative module language. The dynamic semantics are defined through big step reduction rules that convert a module \( m \) into an environment \( \Theta \) and a location store \( \mu \). The environment \( \Theta \) stores the module and value bindings defined in the module. Formally \( \Theta \) is defined as:

\[ \Theta ::= \varepsilon \quad | \quad v_i \mapsto v; \Theta \quad | \quad X_i \mapsto \Theta'; \Theta \quad | \quad (\lambda X_i.m); \Theta \]

31
As was the case previously in the formalisation of MiniML (Section 2.1), \( \mu \) is a sequence of mappings between locations \( l \) and the values \( v \) of the core language MiniML, formally defined as:

\[
\mu ::= \emptyset \mid \mu, l = v
\]

Note that, in contrast to MiniML, locations \( l \) in ModuleML do not have an index \( i \). We detail the technical reasons for this further on in Section 2.2.2 when we formalise the core language of ModuleML.

The reduction rules for the modules of ModuleML involve three different relations: the top-level relation \( \rightarrow_m \), the structure body relation \( \rightarrow_s \) and the reduction rules for the core language \( \rightarrow \). The top-level relation \( \rightarrow_m \) is of the form:

\[
\Theta \mid \mu \vdash m \rightarrow_m \Theta' \mid \mu' \vdash \Theta''
\]

its reduction rules convert an ModuleML module \( m \) into an environment \( \Theta \) and a location store \( \mu \) as illustrated below.

\begin{align*}
&\text{(Module-Identifier)} & (\text{Ascription}) \\
&\Theta(X_i) = \Theta' & \Theta \mid \mu \vdash m \rightarrow_m \Theta' \mid \mu' \vdash \Theta'' \\
&\Theta \mid \mu \vdash X_i \rightarrow_m \Theta \mid \mu \vdash \Theta' & \Theta \mid \mu \vdash (m : M) \rightarrow_m \Theta' \mid \mu' \vdash \Theta'' \\
&\text{(Module-Path)} & (\text{Ascription}) \\
&\Theta \mid \mu \vdash p \rightarrow_m \Theta \mid \mu \vdash \Theta' \mid \Theta'(X_i) = \Theta'' & \Theta \mid \mu \vdash (\lambda X_i . m) \rightarrow_m \Theta' \mid \mu' \vdash \Theta'' \\
&\Theta \mid \mu \vdash p.X_i \rightarrow_m \Theta \mid \mu \vdash \Theta' & (\text{Ascription}) \\
&\Theta \mid \mu \vdash \text{struct } s \text{ end} \rightarrow_m \Theta' \mid \mu' \vdash \Theta'' & \Theta \mid \mu \vdash (\text{functor}(X_i : M)m) \rightarrow_m \Theta \mid \mu \vdash (\lambda X_i.m); \Theta
\\
&\text{(Functor-Application)} & \Theta \mid \mu \vdash m_1 \rightarrow_m \Theta \mid \mu' \vdash (\lambda X_i.m_f); \Theta_f & \Theta \mid \mu' \vdash m_2 \rightarrow_m \Theta'' \mid \mu'' \vdash \Theta'' \\
&\Theta \mid \mu \vdash m_1(m_2) \rightarrow_m \Theta_f \mid X_i \mapsto \Theta'' \mid \mu'' \vdash m_f & \Theta \mid \mu \vdash (\text{functor}(X_i : M)m) \rightarrow_m \Theta \mid \mu \vdash (\lambda X_i.m); \Theta
\end{align*}

Note that in rule Ascription the ascription type information is discarded as module type ascription only affects the type of the module. Note also that in the Structure rule it is not the \( \rightarrow_m \) relation that reduces the body of a structure, but that control is instead diverted the structure body relation \( \rightarrow_s \).

This structure body relation \( \rightarrow_s \) is of the form:

\[
\Theta \mid \mu \vdash s \rightarrow_s \Theta \mid \mu' \vdash \Theta
\]
its reduction rules convert a structure body $s$ consisting of a sequence of structure components $C$ into an environment of value and module bindings $\Theta$ and a location store $\mu$, as listed below.

\[
\begin{align*}
\text{(Module-Binding)} & \\
\Theta | \mu \vdash m \rightarrow_{m} \Theta | \mu' \vdash \Theta'' & \\
\Theta; X_i \mapsto \Theta'' | \mu' \vdash s \rightarrow_{s} \Theta' | \mu'' \vdash \Theta^s
\end{align*}
\]

\[
\Theta | \mu \vdash \text{module } X_i = m; s \rightarrow_{s} \Theta | \mu'' \vdash (X_i \rightarrow \Theta''; \Theta^s)
\]

\[
\begin{align*}
\text{(Value-Binding)} & \\
\Theta | \mu \vdash e \rightarrow^{*} \Theta | \mu' \vdash v & \\
\Theta; v_i \mapsto v | \mu' \vdash s \rightarrow_{s} \Theta' | \mu'' \vdash \Theta^s
\end{align*}
\]

\[
\Theta | \mu \vdash \text{val } v_i = e; s \rightarrow_{s} \Theta | \mu'' \vdash (v_i \mapsto v; \Theta^s)
\]

\[
\begin{align*}
\text{(Type-Binding)} & \\
\Theta | \mu \vdash s \rightarrow_{s} \Theta' | \mu' \vdash \Theta_s
\end{align*}
\]

\[
\Theta | \mu \vdash \text{type } t = \tau; s \rightarrow_{s} \Theta | \mu' \vdash \Theta_s
\]

\[
\Theta | \mu \vdash \text{val } \epsilon \rightarrow_{s} \Theta | \mu \vdash \epsilon
\]

Note again that in rule \textit{Type-Binding} the type binding is discarded as type bindings only affect the type of a module not its run-time evaluation.

**Typing Rules**

The typing rules for the ModuleML module language are largely standard. In this work we use SML style generative functors which return "fresh" abstract types with each application, Consider the following example:

```
signature S_a = sig
  val func: Int -> Int
end

argument Signature

signature S_r = sig
  type X
  val func: Int -> X
end

return Signature

module F = functor(A : S_a)
struct
  type X = int
  val func y = y
end : S_r

generative Functor

module M = struct
  val function x = x * 2
end
P = F(M)
Q = F(M)
```

Generating Abstract Types

where we have a functor F that returns a structure whose signature defines an abstract type X. Every time this functor F is applied to an argument that is a subtype of the argument signature $S_a$, it \textit{generates} a new abstract type for the resulting module. For our given example we thus have that:

\[P.X \neq Q.X\]
as applying the functor \( F \) results in fresh abstract types that are not equal.

Our generative functors stand in contrast to Leroy’s applicative functors [49] that return the same abstract type with each application. While applicative functors allow for a higher degree of modularity, generative functors provide better data encapsulation, is the original ML module formalism for application, and is more in line with our overall goal of secure compilation [19].

We now give an overview of the typing rules, which assign module types to module expressions: \( \Gamma \vdash m : M \), and assign signatures to structures: \( \Gamma \vdash s : S \). The typing rules for modules also rely on the typing of expressions of the core language: \( \Gamma \vdash e : \tau \), which was discussed previously in Setion 2.1. The typing environment \( \Gamma \) is formally defined as follows.

\[
\Gamma ::= \emptyset \mid \Gamma, l : \tau \mid \Gamma, x : \tau \mid \Gamma, C
\]

It differs from the typing environment of MiniML in that it also tracks the defined signature components.

The ModuleML typing rules are as follows.

\[
\begin{align*}
\text{(Type-Module-Identifier)} &\quad \Gamma = \Gamma'\; \text{module} \; \ast_{i} \; M; \; \Gamma'' \quad \Gamma \vdash \ast_{i} : M \\
\text{(Type-Empty-Structure)} &\quad \Gamma \vdash \ast : \ast \\
\text{(Type-Path)} &\quad \Gamma \vdash p : M/p \\
\text{(Type-Subtyping)} &\quad \Gamma \vdash m : M' \quad \Gamma \vdash M' <: M \quad \Gamma \vdash m : M \\
\text{(Type-Structure)} &\quad \Gamma \vdash \text{struct} \; s \; \text{end} : \text{sig} \; s \; \text{end} \\
\text{(Type-Functor)} &\quad X \not\in \text{Dom}(\Gamma) \quad \Gamma \vdash \text{Functor}(X) \; m : \text{Functor}(X) \to M' \\
\text{(Type-Functor-Application)} &\quad \Gamma \vdash m_{1} : \text{Functor}(X) \to M' \quad \Gamma \vdash m_{2} : M \\
\text{(Type-Module-Access)} &\quad \Gamma \vdash p.\ast_{i} : M\{n_{i} \mapsto p.n \mid n_{i} \in \text{Dom}(S_{1})\} \\
\text{(Type-Value-Binding)} &\quad \Gamma \vdash \text{val} \; v_{i} = e ; s : \text{val} \; v_{i} ; \tau ; S \\
\text{(Type-type-Binding)} &\quad \Gamma \vdash \text{val} \; v_{i} = e ; \ast : \ast \\
\end{align*}
\]
The rule *Type-Module-Binding* that the type of the argument must be well-formed, i.e., a valid module type. These well-formedness rules are entirely derivative of the typing rules and are thus omitted.

In the premise of rule *Type-Module-Access* we consider the path $p$ as a special case of module expression. The rule says that $p$ in $p.X_i$ must refer to a structure with a module component named $X_i$, the type for this component gives the type for $p.X_i$.

The rule *Type-Path* relies on a type strengthening operation: $(M/p)$, that enriches the module type $M$ to reflect that its abstract type components come from the path $p$. The strengthened type $M/p$ is a subtype of $M$. Hence, it is always safe to apply the strengthening rule before checking type inclusion. Type strengthening is defined as follows:

\[
\begin{align*}
\text{(sig $S$ end)}/p &= \text{sig $S/p$ end} \\
\text{(functor}(X_i : M) \rightarrow M')/p &= \text{functor}(X_i : M) \rightarrow M' \\
\varepsilon/p &= \varepsilon \\
\text{(val $v_i : \tau$; $S$)/$p$ &= \text{val $v_i : \tau$; $S/p$} \\
\text{(type $T_i$; $S$)/$p$ &= \text{type $T_i = p.T$; $S/p$} \\
\text{(type $T_i = \tau$; $S$)/$p$ &= \text{type $T_i = p.T$; $S/p$} \\
\text{(module $X_i : M$; $S$)/$p$ &= \text{module $X_i : M/p.X_i$; $S/p$}
\end{align*}
\]

As denoted in the rule *Type-Subtyping*, the ModuleML defines sub-typing rules between the modules. These sub-typing rules, are defined as follows.

\[
\begin{align*}
\text{(Subtype-Functor)} & \\
\Gamma \vdash M_2 < : M_1 & \\
(\Gamma; \text{module } X_i : M_2) \vdash M'_1 < : M'_2 & \\
\Gamma \vdash \text{Functor}(X_i : M_1) \rightarrow M'_1 < : \text{Functor}(X_i : M_2) \rightarrow M'_2 & \\
\text{(Subtype-Signature)} & \\
\forall i \in \{1, \ldots, l\}. \Gamma; C_i ; \ldots; C_k \vdash C_{f(i)} < : C'_i & \\
\Gamma \vdash \text{sig } C_1 ; \ldots; C_k \text{ end} < : \text{sig } C'_1 ; \ldots; C'_k \text{ end}
\end{align*}
\]
The rules Subtype-Abstract-Manifest and Subtype-Manifest-Binding make use of a type equivalence relation ≈. This relation captures the type assignments encoded in manifest types, as follows.

\[
\begin{align*}
\Gamma &\vdash \tau \approx \tau' \\
\Gamma &\vdash (\text{type } T_i = \tau) \Rightarrow (\text{type } T_i = \tau')
\end{align*}
\]

Type equivalence for the core language types is entirely standard.

2.2.2 The Core Language

The core language of ModuleML extends MiniML with paths \( p \) to value identifiers and type identifier to enable their evaluation and typing. The locations of MiniML are also simplified to more standard locations without indices, leading to the removal of the index construct as well. As discussed in Chapter 6, our secure compiler does not consider certain low-level errors such as out of memory exceptions. Location indices help formalise low-level memory details such as size and allocation order [40] that are relevant in our development of a secure foreign function interface (Chapter 3) but in contrast with the goals of our secure compiler.

In what follows we formalise the modifications to the syntax of MiniML as well as the additional reduction rules and typing rules.

Syntax

The syntax of the core language removes the index construct and replaces the previously indexed locations \( l_i \) with abstract locations \( l \). It also adds module paths \( p \) to the syntax as denoted below.
Expressions:  
\[ e ::= \cdots \]
\[ \mid p.v_i \quad \text{Value Identifier of a Structure } p \]
\[ \mid v_i \quad \text{Value Identifier} \]

Types:  
\[ \tau ::= \cdots \]
\[ \mid p.T_i \quad \text{Type Component of a Structure } p \]
\[ \mid T_i \quad \text{Type Identifier} \]

Note that \( v_i \) and \( T_i \) are identifiers of the module system (Section 2.2.1).

Reduction Rules
To enable the evaluation of the module paths \( p \) within the expressions of the core language the reduction relation is updated to include \( \Theta \): the environment of value and module bindings, as follows.

\[ \Theta \mid \mu \vdash E[e] \longrightarrow \Theta \mid \mu' \vdash E'[e'] \]

This additional environment is used for reducing value identifiers \( v_i \) as denoted below.

\[
\begin{align*}
\Theta(v_i) &= e \\
\Theta \mid \mu \vdash E[v_i] &\longrightarrow \Theta \mid \mu \vdash E[e] \\
\Theta \mid \mu \vdash p &\longrightarrow \mu' \Theta' \mid \mu \Theta'(v_i) &= v \\
\Theta \mid \mu \vdash E[p.v_i] &\longrightarrow \Theta \mid \mu \vdash E[v]
\end{align*}
\]

Typing Rules
The typing rules of the core language make use of the enhanced typing environment of ModuleML (\( \Gamma \)) to type check the added type identifiers \( T_i \).

\[
\begin{align*}
\Gamma = \Gamma' ; \quad \text{val } v_i : \tau ; \Gamma'' &\quad \Gamma \vdash v_i : \tau \\
\Gamma \vdash p : (\text{sig } S_1 ; \quad \text{val } v_i : \tau ; \quad S_2 \text{ end}) &\quad \Gamma \vdash p.v_i : \tau \{ n_i \leftarrow p.n \mid n_i \in \text{Dom}(S_1) \}
\end{align*}
\]

2.2.3 ModuleML programs
Having formalised the module and core language, we can now formalise full ModuleML programs. A ModuleML program \( P \), informally, is a top level structure followed by a core language expression \( e \) that functions as the starting point of the program \( P \).

\[ P ::= \text{struct } s \text{ end } ;; e \]

To type check a ModuleML program we thus first type check the structure body \( s \), followed by the expression \( e \).

\[
\begin{align*}
\Gamma \vdash s : S &\quad \Gamma \vdash \text{sig } S \text{ end} \vdash e : \tau \\
\Gamma \vdash \text{struct } s \text{ end } ;; e &\text{ ok}
\end{align*}
\]

Full program reduction, captured by the reduction relation \( \longrightarrow_p \), is of the following form.
\[ \Theta | \mu | P \xrightarrow{p} \Theta' | \mu' | P' \]

This \( \xrightarrow{p} \) relation utilises the previously defined \( \xrightarrow{m} \) and \( \xrightarrow{e} \) relations as follows.

\[
\varepsilon | \emptyset \vdash s \xrightarrow{m} \Theta | \mu \vdash \Theta | \mu \vdash e \xrightarrow{e} \Theta | \mu' \vdash v
\]

\[ \varepsilon | \emptyset \vdash \text{struct s end}; e \xrightarrow{p} \Theta | \mu' \vdash v \]

2.3 Contextual Equivalence

The security properties considered in this thesis are captured through the notion of contextual equivalence. The notion of contextual equivalence (Definition 3 below) relies on the definition of context and of divergence (Definition 1 and 2), which are introduced in Section 2.3.1. Next, this section discusses the pros and cons of contextual equivalence as security property (Section 2.3.2).

2.3.1 Contextual Equivalence

Contextual equivalence (also known as observational equivalence) provides a formal means for observing the behaviour of a software component \( c \): a program, module or term that can interact with other software. Informally, contextual equivalence states when two components exhibit the same observable behaviour.

The behaviour of these two components is observed by means of a context \( C \), formally defined as follows.

**Definition 1 (Context)** A context \( C \) is a program with a hole (denoted by \( [·] \)), which can be filled by a component \( c \), generating a new whole program: \( C[c] \).

Based on the source language, contexts can assume a variety of forms. For example, if \( c \) is the MiniML expression:

\[ (\lambda x : \text{Bool}. x) \]

then the context \( C \) is another MiniML expression with a hole, such as the following examples.

\[ (\lambda y : \text{Bool} \rightarrow \text{Bool}. y) [·] \quad \text{Or} \quad [·] \text{true}. \]

Analogously, when the software component \( c \) is a ModuleML module \( M \):

\[
\begin{array}{c}
\text{module } M = \text{struct} \\
\text{val } v_1 = \text{ref } \top \\
\text{end} \\
\end{array}
\]

then the context \( C \) is an incomplete ModuleML program \( P \) that imports the module \( M \) and uses the bindings it provides, such as, for example, the ModuleML program listed below.
From the semantic perspective, plugging a component \( c \) into a context \( C \) makes the program whole, so its behaviour can be observed via the operational semantics of the source language. A different way to close the term would be via system-level semantics \([31]\). In this kind of approach, the context (called the opponent) is not constrained by the syntax of a language as is the case with contextual equivalence, so it can model a powerful attacker to the code. Later on in the formalisation of our secure foreign function interface (Chapter 3) we will use slightly similar technique, our contexts will still be constrained by the syntax of a programming language, but it will be a different from the source language of the component \( c \).

Our contexts \( C \) as currently defined can interact with a component \( c \) and observe its behaviour. However, these observations are quite meaningless if the context has no means of communicating them. Throughout this work contexts communicate their observation by means of divergence.

**Definition 2 (Divergence)** A program \( P \) diverges if it performs an unbound number of reduction steps. Denote that \( P \) diverges as \( P \uparrow \).

Contextual equivalence can now be formalised as follows.

**Definition 3 (Contextual equivalence \([74]\))** Two components \( c_1 \) and \( c_2 \) are contextually equivalent if they are interchangeable in any context without affecting the observable behaviour of the program:

\[
c_1 \simeq c_2 \stackrel{\text{def}}{=} \forall C. \ C[c_1] \uparrow \iff C[c_2] \uparrow.
\]

Alternatively, contextual equivalence can be formalised by using reduction to the same value, in contrast to divergence, as in the work of Curien \([15]\). Another alternative definition is the probabilistic contextual equivalence of Abadi et al. \([2]\), where two components are contextual equivalent if they behave the same to a certain probability.

Note that in statically typed languages such MiniML and ModuleML two contextually equivalent components \( c_1 \) and \( c_2 \), are of the same type as otherwise they cannot be plugged into the same contexts \( C \).

**Examples**

Two MiniML terms \( \bar{\tau} \) and \( \bar{\tau} \) are, for example, not contextually equivalent as a context:

\[
C = (\textbf{if} \ (\varepsilon) \ == \bar{\tau} \ \Omega \ \textbf{true})
\]

where \( \Omega \) is a diverging term, can distinguish between them.

MiniML’s \( \lambda \)-terms, in contrast, introduce many equivalences. The internal body of a \( \lambda \)-term is not observable unless the expression in the body leaks
information out to the context by applying an outside function or an assignment to a location. There is no context \( C \), for example, that can distinguish the following terms.

\[
(\lambda x : \text{Int}.(x + x)) \quad (\lambda x : \text{Int}.(2 \times x))
\]

### 2.3.2 Contextual Equivalence as a Security Property

From our security based perspective, contexts can be considered not just a model for observing the behaviour of components, but can also be used as model for attacks against these components. Consider, for example, the following two MiniML higher-order \( \lambda \)-terms:

\[
(a) \ (\lambda f : \text{Int} \to \text{Int}.(f \ 2) + (f \ 2)) \\
(b) \ (\lambda f : \text{Int} \to \text{Int}.(f \ 2) \times 2)
\]

both \( \lambda \)-terms apply the argument \( f \) to the number 2 and double the result. The \( \lambda \)-term (a), however, achieves this by applying \( f \) to 2 a second time, whereas \( \lambda \)-term (b) simply multiplies the result of the first application by two. In a purely functional \( \lambda \)-calculus with no side-effects, these two terms are contextually equivalent as there is no context that can distinguish them. In a \( \lambda \)-calculus such MiniML that includes references these two terms are, however, not equivalent as the following context can be crafted as an attack to distinguish between them.

\[
\mathcal{C}_a = \text{let } r = (\text{ref } 0) \text{ in} \\
(\lfloor \cdot \rfloor (\lambda y : \text{Int} \to \text{Int}.r := !r + 1; y)); \\
(\text{if } !r = 2 \text{ then } \Omega \text{ else } 1)
\]

Plugging \( \lambda \)-term (a) into the context \( \mathcal{C}_a \) will result in divergence (\( \Omega \)) as the reference \( r \) will be increased twice. Plugging \( \lambda \)-term (b) into the context \( \mathcal{C}_a \), however, does not result in divergence as the reference \( r \) is only increased once. The context \( \mathcal{C}_a \) can thus be considered a successful attack against the implementation details of these two \( \lambda \)-terms.

The fact that contexts are not just models of interaction but can also encode all kinds of attacks against components, allows us to use contextual equivalence as a means of capturing security properties such as confidentiality and integrity.

#### Confidentiality

Consider the following two contextually equivalent MiniML terms.

\[
(a) \ \text{let secret} = \text{ref } 0 \text{ in} \\
(\lambda x : \text{Int}.\text{secret} := \text{secret} + 1; x) \\
(b) \ \text{let secret} = \text{ref } 0 \text{ in} \\
(\lambda x : \text{Int}.x)
\]
Terms (a) and (b) both reduce to a $\lambda$-term that is an identity function. However, whereas term (b) reduces to a side-effect free identity function, term (a), in contrast, reduces to an identity function that will update the secret location. Every time the result of term (a) is used by MiniML the value of secret will increase.

Despite this difference in implementation, the two terms are contextually equivalent as there exists no context $C$ that could distinguish between them. While locations can be converted to numbers in MiniML, by using the `index` construct, the reverse: converting a number, or any other value, to a location is not possible, as context $C$ thus has no means to access the location secret and inspect its contexts.

In this example contextual equivalence captures the confidentiality properties of MiniML locations.

**Integrity**

Consider the following contextually equivalent higher-order MiniML terms.

(a) $(\lambda f : \text{Int} \to \text{Int}.)$

\[
\text{let } y = \text{ref } 0 \text{ in let } y = \text{ref } 0 \text{ in (f 1)} \\
\text{let } r = (f 1) \text{in} \\
\text{if } (!y = 0) \text{ then } r \text{ else } (-1) 
\]

(b) $(\lambda f : \text{Int} \to \text{Int}.)$

\[
\text{let } y = \text{ref } 0 \text{ in (f 1)} \\
\text{let } r = (f 1) \text{in} \\
\text{if } (!y = 0) \text{ then } r \text{ else } (-1) 
\]

The higher-order $\lambda$-terms (a) and (b) both apply the argument $f$ to the number 1. Term (a), however, differs from term (b) in that it performs an integrity check after applying $f$ to 1: it checks that the location $y$ was not modified while control was diverted to the outside argument.

Despite this difference in implementation, the two terms are contextually equivalent as there exists no context $C$ that can provide an argument $f$ capable of modifying the reference $y$. We say that in this example contextual equivalence captures the integrity properties of the $\lambda$-term.

**The Limitations of Contextual Equivalence**

While contextual equivalence can be used to express integrity and confidentiality properties, it does not capture all possible security properties relevant to software components. This limitation is due to the limited range of attacks that a context $C$ can express.

A context $C$ is defined by the source language of the components that it interacts with. It can thus not be used to express side-channel attacks that rely on, for example, timing or power consumption information. These types of attacks will thus be disregarded by the run-time protection techniques that we introduce in this thesis.
2.4 Fully Abstract Translation

For every one of the three areas that we develop run-time protection techniques: foreign function interfaces, secure compilation and abstract machines, we formally prove the security of our protection techniques through the notion of fully abstract translation. A translation is fully abstract if it translates contextually equivalent source-level programs into contextually equivalent target-level programs. Moreover, contextually equivalent translated programs derive from contextually equivalent source-level ones.

Consider a source language $S$ and a target language $T$, the translated version of a program $P$ in $S$ is denoted as: $\{P\}_T$. Fully abstract translation can now be defined as follows.

**Definition 4 (Fully Abstract Translation)** A translation between two languages $S$ and $T$ is fully abstract if it preserves and reflects contextual equivalence:

$$P_1 \simeq^S P_2 \iff \{P_1\}_T \simeq^T \{P_2\}_T$$

The preservation and reflection of contextual equivalence implies that no additional security flaws are introduced by the translation. However, an already insecure program in the source-language when translated into a target language will still be insecure. A fully abstract translation is thus a conservative security mechanism as it introduces no more vulnerabilities than the ones already present in the source-level program.

The main benefit of fully abstract translation is that it provides the programmer with source-level reasoning. Source level reasoning ensures that a programmer need not be concerned with the security challenges presented by the target language. From a security point of view, this property ensures that the security properties of a software system can be established by reviewing the source code [11].

The proof of full abstraction is split into two separate theorems: preservation and reflection. Preservation of contextual equivalence means that the translation results in target-level programs that behave as their source-level counterparts. Reflection implies that the source-level security properties are not violated by the generated target-level output.

In this thesis we prove that our secure foreign function interface (Chapter 3) and secure abstract machine (Chapter 5) uphold both theorems, they are thus fully abstract translations to their respective target languages. Our secure compilation scheme (Chapter 6) is only proven to uphold the reflection theorem. As is common in secure compilation works that target a realistic low-level target language [66], we assume that preservation holds. Preservation coincides with correctness, it establishes that the compiler is a correct compiler. While we have tested our implementation of the compilation scheme intensely, we consider formally verifying the implementation of our compiler a separate research subject.
2.5 Bisimilarity and Trace Equivalence

The definition of full abstraction contains a universal quantification over all possible contexts, due to the expansion of the definition of contextual equivalence. This makes proofs of full abstraction challenging, as reasoning (and proving properties) about contexts is notoriously complex [7, 27, 40, 66]. To adapt to this challenge, we make use of two different forms of equivalence, bisimilarity (Section 2.5.1) and trace equivalence (Section 2.5.2) which are proven to be as precise as contextual equivalence.

2.5.1 Bisimilarity

Bisimilarity has been frequently employed in the field of process algebra in order to state when two processes exhibit the same behaviour [76]. Intuitively, two processes have the same behaviour when they perform the same actions and become new processes that continue to have the same behaviour.

The notion of bisimilarity (Definition 6) relies on the concept of labelled transition system (LTS), formally defined as follows.

**Definition 5 (LTS)** A labelled transition system is a triplet \((S, \Lambda, \rightarrow)\) where \(S\) is a set of states, \(\Lambda\) 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 \(S\) performs computations; labels often concern inputs and outputs, as presented in the following example.

**Example 1 (LTS [76])** Consider the LTS of a vending machine that produces tea or coffee after receiving coins, after the appropriate request is made. It is formalised as an LTS: \(\{(S_I, S_R, S_T, S_C), \{\text{coin}, \text{req-tea}, \text{req-coffee}, \text{tea}, \text{coffee}\}, \{S_I \xrightarrow{\text{coin}} S_R, S_R \xrightarrow{\text{req-tea}} S_T, S_R \xrightarrow{\text{req-coffee}} S_C, S_T \xrightarrow{\text{tea}} S_I, S_C \xrightarrow{\text{coffee}} S_I\}\) and it is depicted below.

\[
\begin{align*}
S_I & \xrightarrow{\text{coin}} S_R \\
S_R & \xrightarrow{\text{req-tea}} S_T \\
S_R & \xrightarrow{\text{req-coffee}} S_C \\
S_C & \xrightarrow{\text{coffee}} S_I
\end{align*}
\]

\(S_I\) 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 waits 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 provides to the user: tea or coffee.

Often, labels are also equipped with decorations that indicate the direction of the action: \( ! \) is an observable produced from the program, \( ? \) is an observable received by it. The aforementioned transitions can thus be decorated as follows:

\[
\begin{align*}
S_I &\xrightarrow{\text{coin}？} S_R, \\
S_R &\xrightarrow{\text{req-coffee}？} S_C, \\
S_T &\xrightarrow{\text{tea}!} S_I.
\end{align*}
\]

**Definition 6 (Bisimilarity)**

Given a LTS \((S, \Lambda, \rightarrow)\), a relation \( R \subseteq S \times S \) is a bisimulation if, for any pair \((S_1, S_2) \in R\), for all \( \lambda \in \Lambda \), the following holds:

1. for all \( S_1' \) such that \( S_1 \xrightarrow{\lambda} S_1' \), there exists \( S_2' \) such that \( S_2 \xrightarrow{\lambda} S_2' \) and \((S_1', S_2') \in R\);

2. for all \( S_2' \) such that \( S_2 \xrightarrow{\lambda} S_2' \), there exists \( S_1' \) such that \( S_1 \xrightarrow{\lambda} S_1' \) and \((S_1', S_2') \in R\).

Bisimilarity, denoted as \( \approx \), is the largest union of all bisimulations.

Two states \( S_1 \) and \( S_2 \) are thus bisimilar if there exists a bisimulation relation \( R \) such that \( S_1 R S_2 \).

**Weak Bisimilarity**

A more permissive variant of bisimilarity that is often used for program equivalence is weak bisimilarity. Its definition is the same as that of bisimilarity except that \( \lambda \Rightarrow \) is used in place of \( \lambda \rightarrow \). Relation \( \lambda \Rightarrow \) abstracts away from silent transitions that model internal computations. Formally \( \lambda \Rightarrow \) is defined as \( \tau \rightarrow \star \xrightarrow{\lambda} \tau \rightarrow \star \). Thus, the programs (a) and (b):

\[
\begin{align*}
(a) & \ x := 0; \ x := !x + 1 \\
(b) & \ x := 1
\end{align*}
\]

are weakly bisimilar (if a location \( x \) is not observable) even though they perform a different number of internal steps, while they are not bisimilar according to Definition 6.

**Congruent Bisimilarity**

Bisimilarity can be used in place of contextual equivalence for proving full abstraction, when the labels of the LTS model what a context can observe about a program. In such as an LTS the context is modelled as a black box that triggers transitions. A bisimulation defined over this kind of LTS abstracts the state of the context but captures the reactions of a program to the actions of the context. This abstraction is the great advantage of bisimulation over contextual equivalence.

It is crucial, when replacing contextual equivalence with bisimulation, that all possible context behaviour is captured by the labels of the LTS, so as to
have a precise characterisation of what a context can observe. We say that a bisimilarity \( \approx \) that coincides with the contextual equivalence relation \( \simeq \) over programs \( P \) is a congruent bisimilarity.

**Definition 7 (Congruence of Bisimilarity)** A bisimilarity \( \approx \) over programs \( P \) is a congruent bisimilarity if it coincides with contextual equivalence:

\[
P_1 \approx P_2 \iff P_1 \simeq P_2
\]

### 2.5.2 Trace Equivalence

Trace equivalence is a notion that, like bisimilarity, has been used successfully to replace contextual equivalence. Traces are often defined inductively, enabling a neat, structural argument for proofs adopting them.

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 \( \lambda \), are called traces. The trace semantics of a program is the set of all traces that can be associated with the program, based on its labelled transition system.

**Definition 8 (Trace semantics)** The trace semantics of a program \( P \) are formally denoted as \( \text{Traces}(P) \).

\[
\text{Traces}(P) = \{ \lambda \mid \exists P'. P \xrightarrow{\lambda} P' \}
\]

Two programs are said to be trace equivalent, denoted with \( P_1 \overset{T}{=} P_2 \), if their traces coincide.

**Definition 9 (Trace equivalence)** \( P_1 \overset{T}{=} P_2 \triangleq \text{Traces}(P_1) = \text{Traces}(P_2) \).

The same considerations made for bisimulation (Section 2.5.1) also apply for trace equivalence. Since labels capture context/attacker actions, it is crucial that all possible observations can observe are captured in the labels. This is established by proving full abstraction for the trace semantics: that trace equivalence coincides with contextual equivalence.

**Definition 10 Full Abstraction of Trace Semantics**

\[
P_1 \overset{T}{=} P_2 \iff P_1 \simeq P_2
\]

### 2.6 The Protected Module Architecture (PMA)

PMA is a low-level memory protection mechanism that introduces protected memory modules which provide a secure environment for code and data that needs to be protected from malware with kernel level code injection capabilities. A protected memory module does not completely isolate the protected code and data from the rest of the system. Similar to the modules of
ModuleML (Section 2.2) a protected memory module provides an interface mechanism to allow interoperation with code residing outside of the protected module.

PMA is implemented as a fine-grained, program counter-based, memory access control mechanism that divides memory into a protected memory module and unprotected memory [21, 57, 64, 80]. The protected module is further split into two sections: a protected code section accessible only through a fixed collection of designated entry points, and a protected data section that can only be accessed by the code section. As such the unprotected memory is limited to executing the code at entry points. The code section can only be executed from the outside through the entry points and the data section can only be accessed by the code section. An overview of the access control mechanism is given in Table 2.2.

<table>
<thead>
<tr>
<th>From \ To</th>
<th>Protected</th>
<th>Unprotected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry Point</td>
<td>Code</td>
<td>Data</td>
</tr>
<tr>
<td>Protected</td>
<td>r x</td>
<td>r x</td>
</tr>
<tr>
<td>Unprotected</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.2. The access control policy of PMA

The abstract machine and secure compiler developed as part of this thesis rely on prototype implementations of PMA (Section 6.5 and 5.3.2). However, as of late 2015, Intel supports PMA in its Skylake processors through the Intel SGX instruction set [57].

2.7 The Low-Level Attacker Model A+I

The attacker considered in this work is an assembly program that has kernel-level code injection privileges that can be used to introduce malware into a software system. Kernel-level code injection is a critical vulnerability that bypasses all existing software-based security mechanisms: disclosing confidential data, disrupting applications and so forth. The attacker can thus inspect and manipulate every bit of code and data in the system except for the programs that reside within the protected memory of the PMA mechanism. As noted in Section 2.6, PMA is a program counter-based mechanism, which the kernel-level code injection capabilities of this attacker model cannot bypass [80].

To model this low-level attacker we adopt Patrignani et al.’s A+I (acronym of Assembly plus Isolation) [67]. The A+I language is an untyped assembly language attacker running on a von Neumann machine consisting of a program counter \(p\), a register file \(r\), a flags register \(f\) and a memory space \(m\) that maps addresses to words and contains all code and data. The supported instructions are the standard assembly instructions for integer arithmetic, value
comparison, address jumping, stack pushing and popping, register loading and memory storing. The formal details of these instructions and their operational semantics are not essential to the rest of the thesis and have thus been left out of this these. For a full formalisation of these instructions, we refer the interested reader to Patrignani’s formalisation [65].

There is one unusual constraint on this attacker in that it is split in a code and a data section, just as the protected code of PMA is. This is due to the fact that low-level writes and reads from the protected memory can only be performed on the data section of unprotected code. From the threat modeling perspective this assumption somewhat reduces the attacker’s power, since the attacker is not able to execute the values written by the protected code.

The Low-Level Labelled Transition System
To formally reason about the capabilities and behaviour of this attacker, we make use of the fully abstract trace semantics of Patrignani and Clarke for A+I programs [67]. These trace semantics transition over a state $\Lambda$ with two kinds of sub-states.

$$\Lambda ::= (p,r,f,m,s) \mid (\text{unk},m,s)$$

When code is executing in the protected memory of PMA $\Lambda$ is of the form $(p,r,f,m,s)$, where $m$ represents only the protected memory of PMA and $s$ is a descriptor that details where the protected memory partition starts as well as the number of entry points and the size of the code and data sections of both the protected and unprotected memory. Additionally, the state $\Lambda$ can be $(\text{unk},m,s)$, a state modelling that code is executing in unprotected memory.

The fully abstract trace semantics denote the observations of the low-level A+I contexts that interact with the protected memory through labels $L$. The syntax of these labels $L$ are as follows.

$$L ::= \alpha \mid \tau \qquad \alpha ::= \sqrt{\delta} \mid \gamma? \qquad \gamma ::= \text{call } p(r,f) \mid \text{ret } p(r,f) \qquad \delta ::= \gamma \mid \omega(a,v).\delta \qquad \omega ::= \text{read} \mid \text{write}$$

A label $L$ can be either an observable action $\alpha$ or a non-observable action $\tau$ indicates that an unobservable action occurred in protected memory. Decorations $?$ and $!$ indicate the direction of the observable action: from the unprotected memory to the protected memory (?) or vice-versa (!). Observable actions include a tick $\sqrt{\delta}$ indicating that the evaluation has terminated observable actions. Additionally, observable actions are function calls or returns to a certain address $p$, combined with the registers $r$ and flags $f$. Registers and flags are in the labels as they convey information on the behaviour of the code executing in the protected memory. Observable actions $\omega(a,v)$ from the protected memory to the unprotected memory detail read and writes to the unprotected memory where $a$ is the memory address and $v$ is the value written to the address.

The transitions of the trace semantics of A+I are as follows [65].
In rule Trace-Internal the internal reductions are labelled with $\tau$, the intJump predicate checks that the next instruction address $p'$ resides within the pro-
ected memory. In rule \textit{Trace-Termination} the \( \bot \) superscript denotes a halted state. The rule \textit{Trace-Call} captures a call from the unprotected memory to an entry point \( p \) of the protected memory. The rule \textit{Trace-Returnback} labels a return from the unprotected memory to the protected memory. The unprotected memory does not return directly into the protected but instead must go through a dedicated return entry point. The rule \textit{Trace-Outcall} labels a jump from the protected memory to the unprotected memory as verified by the predicate \texttt{exitJump}. In rule \textit{Trace-return} the protected code returns a value to the unprotected, in this direction the return jump is direct.

For writeouts, rule \textit{Trace-Writeout} ensures writeout labels are always created, where \texttt{unprotectedData} ensures that the address written to is in the unprotected data section of the attacker. Rule \textit{Trace-Writeout-Termination} ensure that no writeout label is created when a program terminates.

For readouts, rule \textit{Trace-Readout} ensures readout labels are always created, where \( \mathcal{W} \) is the set of all possible low-level words and where again \texttt{unprotectedData} ensures that the address read is in the unprotected data section of the attacker.

\section*{Full Abstraction of the Trace Semantics}

Formally the traces of an assembly program \( P \), denoted as \( \text{Tr}(P) \), are computed as follows: \( \text{Tr}(P) = \{ \alpha | \exists \Lambda. \Lambda_0(P) \xrightarrow{\alpha} \Lambda \} \). Where \( \Lambda_0 \) is the initial state and the relation \( \Lambda \xrightarrow{\alpha} \Lambda' \) describes the traces generated by transitions between states.

These trace semantics are fully abstract as they uphold Definition 10. Whenever two assembly programs \( P_1 \) and \( P_2 \) are contextually equivalent they will produce the same sequence of traces (the same inputs and observations by attackers) as follows:

\textbf{Proposition 1 (Full Abstraction of the Traces [68])}

\[ P_1 \simeq_a P_2 \iff \text{Tr}(P_1) = \text{Tr}(P_2) \]

where \( \simeq_a \) denotes contextual equivalence between two A+I programs.
Modern software systems consist of numerous interoperating components written in different source languages. Such language interoperation is usually achieved through a foreign function interface (FFI) that details how data is exchanged and functions are called across the language boundary between the source language and the foreign language. A FFI, however, introduces an explicit security risk: if the contextual equivalences of the source language are not preserved in the foreign language, programs in the foreign language may be able to use the FFI to obtain confidential information or break the integrity of the program in the source language [1].

Previous approaches to securing FFIs, more specifically protecting the contextual equivalences of the source language of FFIs, have relied on static methods that check both the components individually as well as the interoperation between them [29, 30, 82]. However because some software components may be dynamically linked at run-time, written in languages with no abstractions or susceptible to code injection attacks, these static solutions are easily circumvented in practice [58].

In this chapter we introduce a formalism of a foreign function interface that securely interoperates between the source language MiniML (Section 2.1) and a low-level language such as assembly or C, without relying on any static checks on the low-level code. Instead, the memory protection techniques of PMA (Section 2.6) are lifted into the formalism of our secure foreign function interface.

The introduced foreign function interface, referred to as MiniML+, also differs from previous formalisms such as Matthews’ and Findler’s multi-language semantics [55] in that it is less abstract. In contrast to multi-language semantics where the concrete details of function calls and data exchange are left to the implementation, our model MiniML+ provides concrete insight into how to implement these mechanisms in a secure fashion. In part this achieved by encoding the interoperation between MiniML and the low-level attacker into partial evaluation stacks, similar to how continuation passing style conversion makes continuations explicit.

The low-level language is not explicated in MiniML+. It is instead simplified to an attacker model that captures all the threats to the contextual equivalences of the source language that a machine-level language may pose. Later on in Chapter 4 we prove the accuracy of this simplification.

To establish that our FFI, is capable of preserving the contextual equivalence of MiniML in its interactions with the low-level attacker, we prove
that there exists a fully abstract translation between MiniML and our FFI MiniML+. Contextually equivalent terms in the MiniML can thus be translated to contextually equivalent terms in MiniML+ and all translated terms that are contextually equivalent in MiniML+ correspond to contextually equivalent MiniML terms.

Because the contexts used in the formalisation of contextual equivalence lack meaningful structure [27, 81], we prove the full abstraction of this translation scheme by relating two bisimilarity relations that coincide with contextual equivalence for MiniML and MiniML+.

The remainder of this chapter is organized as follows. First this chapter presents some additional details on MiniML contextual equivalences (Section 3.1). Next it introduces the attacker model MiniMLa (Section 3.2). Subsequently, it present Matthews’ and Findler’s multi-language semantics [55] and its limitations (Section 3.3), followed by our secure foreign function interface MiniML+ (Section 3.4). Finally we prove that MiniML+ is indeed secure by proving that a fully abstract translation exists between MiniML and MiniML+ (Section 3.5).

3.1 Contextual Equivalence for MiniML

The secure FFI aims to preserve the contextual equivalences of MiniML. As formalised in Definition 3, a MiniML context C is a MiniML term with a single hole [·], two MiniML programs e1 and e2 are contextually equivalent if and only if there is no context C that can distinguish them. Contextual equivalence for MiniML is formalised as follows.

**Definition 11 Contextual equivalence for MiniML \( (\simeq^M) \) is defined as:**

\[
\mu_1 \mid e_1 \simeq^M \mu_2 \mid e_2 \overset{\text{def}}{=} \forall C. \mu_1 \mid C[e_1] \uparrow \iff \mu_2 \mid C[e_2] \uparrow
\]

where \( \uparrow \) denotes divergence, \( e_1 \) and \( e_2 \) are closed expressions and the contexts \( C \) do not feature explicit locations \( l_i \) as they are not part of the static semantics. Note that two contextually equivalent MiniML programs \( e_1 \) and \( e_2 \) have the same type \( \tau \) as a context \( C \) observes the MiniML typing rules.

3.1.1 Examples of Contextual Equivalence

As previously detailed in Section 2.3 MiniML’s \( \lambda \)-terms introduce many equivalences, there is no context \( C \), for example, that can distinguish the following two \( \lambda \)-terms.

\[
(\lambda x : \text{Int.} \, \bar{0}) \quad (\lambda x : \text{Int.} \, (x - x)) \quad \text{(Ex-1)}
\]

The equivalences over the locations of MiniML are slightly more complex than previously mentioned. Due to the deterministic allocation order and the
inclusion of the \texttt{index} operation, a context can observe the number of locations as well as their indices. The following two terms, for example, are not contextually equivalent.

\[
\begin{align*}
\text{let } x &= \texttt{ref true} \text{ in } \text{let } y = \texttt{ref true} \text{ in } x \\
\text{let } y &= \texttt{ref true} \text{ in } y
\end{align*}
\] (Ex-2)

As the following context \( C_a \):

\[
C_a = (\text{if } (\text{index}[\cdot] == \top) \Omega \text{ true})
\]

can distinguish both terms, where \( \Omega \) is a diverging MiniML term.

Locations when kept secret, however, can still produce equivalences as a context \( C \) cannot contain a location \( l_i \) unless it is shared during evaluation. The following two terms, for example, are thus contextually equivalent.

\[
\begin{align*}
\text{let } x &= \texttt{ref false} \text{ in } \top \\
\text{let } y &= \texttt{ref true} \text{ in } \top
\end{align*}
\] (Ex-3)

### 3.2 The Attacker Model MiniML\( ^a \)

The secure FFI MiniML\( ^+ \) aims to preserve the contextual equivalences of MiniML from malicious machine-level attackers. Such an attacker can break the equivalences/abstractions of a MiniML program that it interoperates with in the following two ways.

**Inspection** An attacker can break the abstractions of MiniML by inspecting and manipulating the internal state of a MiniML program. An attacker can achieve this by either reading and writing to references shared through the foreign function interface (an existing vulnerability in the Java Virtual Machine [82]) or by abusing low-level privileges to inspect the memory directly.

**Breaking Type Safety** When interoperating a MiniML program will not only share values over the FFI, but also receive values from its low-level counterpart. The attacker can take advantage of this by passing language constructs that do not adhere to the typing rules of MiniML.

Our attacker model incorporates both these threats to full abstraction. The attacker model is formalised as a language MiniML\( ^a \) that is derived from MiniML by removing typing safety and incorporating reflection. Removing type safety is achieved by both removing the types and adding a new term \texttt{wr} that captures non reducible expressions.

Reflection is added to MiniML\( ^a \) by means of a syntactical equality testing operator modulo \( \alpha \)-equivalence \( \equiv_\alpha \). Given two terms \( e_1 \) and \( e_2 \), the term \( e_1 \equiv_\alpha e_2 \) will thus only reduce to \texttt{true} if \( e_1 \) and \( e_2 \) are syntactically equal except for the names assigned to the variables.
Syntax
The attacker model MiniML$^a$ is typeset in a serif red font. The syntax of MiniML$^a$ is as follows.

Expressions:  \[ e ::= v \mid x \mid (e_1 e_2) \mid \langle v_i^{1..n} \rangle \mid e_1 \text{ op } e_2 \mid e_1 \text{ cp } e_2 \]
\[ \mid e.1 \mid \text{ if } e_1 e_2 e_3 \mid \text{ let } x = e_1 \text{ in } e_2 \mid !e \mid \text{ ref } e \mid e_1 ; e_2 \]
\[ \mid \text{ fix } e \mid \text{ letrec } x = e_1 \text{ in } e_2 \mid e_1 ::= e_2 \mid \text{ index } e_1 \]
\[ \mid \text{ exit } e \mid \text{ wr } \mid e_1 \equiv \alpha e_2 \]

Operands: \[ \text{ op ::= + | − | } \]

Comparators: \[ \text{ cp ::= = | < | > | == } \]

Values: \[ v ::= \langle v_i^{1..n} \rangle \mid \text{ unit } \mid l_i \mid n \mid (\lambda x. t) \mid b \]

Boolean: \[ b ::= \text{ true } \mid \text{ false } \]

MiniML$^a$ adds the \(\alpha\)-equivalence operator \(\equiv\alpha\) to the syntax as well as a term \(\text{wr}\) that denotes the error state. Note that there are no types as MiniML$^a$ is untyped.

Reduction Rules
Like MiniML, the MiniML$^a$ language defines reduction contexts to lift the basic reduction steps. There are no evaluation context for the sub-terms of the alpha equivalence operator \(\equiv\alpha\) as its reflection operation should not be tampered by the reduction rules.

Evaluation Contexts: \[ E ::= [\cdot] \mid E e \mid v E \mid \langle v_i^{1..n}, E, e_k^{(j+1..n)} \rangle \mid E.i \]
\[ \mid E \text{ op } e \mid E \text{ cp } e \mid v \text{ op } E \mid v \text{ cp } E \mid \text{ if } E e_1 e_2 \]
\[ \mid \text{ let } x = E \text{ in } e \mid !E \mid \text{ ref } E \mid E ::= e \]
\[ \mid E. e \mid v ::= E \mid \text{ fix } E \mid \text{ index } E \]

The reduction rules of MiniML$^a$ are of the form:
\[ \mu | E[e] \rightarrow \mu' | E'[e'] \]
where \(\mu\) is the run-time location store, \(E\) the current evaluation context and \(e\) the expression being reduced.

The reduction rules of MiniML$^a$ are the reduction rules of MiniML extended with new reductions for the \(\alpha\)-equivalence operator as well as reduction rules that capture stuck states.

\[ \text{(Att-If-True)} \]
\[ \mu | E[\text{if true } e_2 e_3] \rightarrow \mu | E[e_2] \]
\[ \text{(Att-If-False)} \]
\[ \mu | E[\text{if false } e_2 e_3] \rightarrow \mu | E[e_3] \]
<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>(Att-If-Error)</strong></td>
<td>( v \neq b )</td>
</tr>
<tr>
<td>( \mu \mid E[\text{if } v \ e_2 \ e_3] \longrightarrow \mu \mid wr )</td>
<td></td>
</tr>
<tr>
<td><strong>(Att-Operand)</strong></td>
<td>( v \neq (\lambda x: \tau. t) )</td>
</tr>
<tr>
<td>( \mu \mid E[(v \ v')] \longrightarrow \mu \mid wr )</td>
<td></td>
</tr>
<tr>
<td><strong>(Att-Projection)</strong></td>
<td>( v \neq \langle v_i \in 1..n \rangle \lor v = \langle v_j \notin 1..n \rangle )</td>
</tr>
<tr>
<td>( \mu \mid E[v_1 \ v_2] \longrightarrow \mu \mid E[wr] )</td>
<td></td>
</tr>
<tr>
<td><strong>(Att-Sequence)</strong></td>
<td>( \mu \mid E[\text{unit} ; e_2] \longrightarrow \mu \mid E[e_2] )</td>
</tr>
<tr>
<td><strong>(Att-Equiv-True)</strong></td>
<td>( e_1 &amp; e_2 \text{ are } \alpha\text{-equiv} )</td>
</tr>
<tr>
<td>( \mu \mid E[e_1 \equiv_\alpha e_2] \longrightarrow \mu \mid E[true] )</td>
<td></td>
</tr>
<tr>
<td><strong>(Att-Equiv-False)</strong></td>
<td>( e_1 &amp; e_2 \text{ are not } \alpha\text{-equiv} )</td>
</tr>
<tr>
<td>( \mu \mid E[e_1 \equiv_\alpha e_2] \longrightarrow \mu \mid E[false] )</td>
<td></td>
</tr>
</tbody>
</table>
3.2.1 Lack of Contextual Equivalence

The addition of reflection in MiniML\textsuperscript{a} through the $\alpha$-equivalence testing operator, renders the abstractions and source level restrictions of MiniML obsolete as contextual equivalence is reduced to $\alpha$-equivalence [85, 60]. Conversely, reflection rules out any sensible notion of security, since the state of a program can be freely inspected.

Consider, for example, the equivalent $\lambda$-terms of Example Ex-1 in Section 3.1.1. The following MiniML\textsuperscript{a} context $C_a$:

$$C_a = (\textbf{if } ((\lambda y. (y - y)) \equiv_{\alpha} [\cdot]) \Omega \text{ true})$$

where $\Omega$ is a diverging term, distinguishes the contextually equivalent terms of the Example Ex-1 due to the $\equiv_{\alpha}$ operator’s ability to inspect and compare the syntax of MiniML terms.

A similar context $C_a$ can thus be built for the contextually equivalent terms of Ex-3 in Section 3.1.1:

$$C_a = (\textbf{if } ((\textbf{let } x = \textbf{ref false in } x) \equiv_{\alpha} [\cdot]) \Omega \text{ true})$$

and for every other pair of contextually equivalent terms.

3.3 Multi-Language Semantics and its Limitations

A commonly used formalisation for foreign function interfaces is the multi-language semantics of Matthews and Findler [55]. In what follows we apply their natural embedding approach (Section 3.3.1), resulting in a new calculus MiniML\textsuperscript{M}, and then detail the formal issues that prevent MiniML\textsuperscript{M} from being a good model for a secure foreign function interface powered by PMA (Section 3.3.2).
3.3.1 MiniML\(^M\): A natural embedding of MiniML and MiniML\(^a\)

The natural embedding introduces two new terms into the combined MiniML\(^M\)-calculus.

MiniML expressions: \(e ::= ... ^\tau \text{SA} e\)

MiniML\(^a\) expressions: \(e ::= ... ^\alpha \text{AS}^\tau e\)

The expression \(\text{AS}^\tau e\) is added to the syntax of MiniML\(^a\), it embeds a MiniML expression \(e\) within the expressions of MiniML\(^a\). Likewise, the expression \(^\tau \text{SA} e\) is added to the syntax of MiniML and embeds MiniML\(^a\) term within the expressions of MiniML. Both terms are annotated with a MiniML type \(\tau\). These type annotations are used to perform dynamic typechecks on the interaction between the terms of the MiniML\(^a\) and MiniML to ensure that the typing properties of MiniML are preserved.

In the combined MiniML\(^M\)-calculus primitive values are simply converted into the respective representation of the other language when they transition between the composed programs. Function calls and locations rely on a wrapping mechanism: when a MiniML\(^a\) \(\lambda\)-term crosses the language boundary, for example, that \(\lambda\)-term is wrapped into a new MiniML \(\lambda\)-term as follows:

\[ ^\tau \rightarrow ^\tau' \text{SA} (\lambda x. t) \rightarrow \lambda y : \tau. (^\tau' \text{SA} ((\lambda x. t) (\text{AS}^\tau y))) \]

This MiniML\(^M\)-calculus is not capable of preserving the abstractions of MiniML from MiniML\(^a\). Reconsider the contextually equivalent terms of Example Ex-1 in Section 3.1.1. In the combined MiniML\(^m\)-calculus these two terms are no longer contextually equivalent as the following MiniML\(^M\)-calculus context can distinguish between them.

\[ \mathbb{C}_a = (\text{AS}^{\text{Int} \rightarrow \text{Int}} (\lambda y : \text{Int}. 0) \equiv _\alpha \text{AS}^{\text{Int} \rightarrow \text{Int}} [\cdot]) \]

The problem is that \(\text{AS}^\tau e\) is a MiniML\(^a\) expression whose contents can be compared against any other MiniML\(^a\) expression by the \(\alpha\)-equivalence operator. In MiniML\(^M\) contextual equivalence is thus reduced to \(\alpha\)-equivalence as is the case for MiniML\(^a\) (Section 3.2.1).

3.3.2 Why PMA can’t by lifted into MiniML\(^M\)

This chapter aims to develop a secure foreign function interface for MiniML by lifting the memory isolation model of PMA into the formalisation. To that end the \(ML^M\)-calculus is investigated for its suitability for lifting PMA into it.

Note that we are aware of the fact that operational semantics of MiniML\(^M\)-calculus was explicitly designed to abstract away low-level details such as memory models. The goal of this section is simply to clarify our reasons for introducing a new operational semantics.
Clearly the MiniML\textsuperscript{M}-calculus is capable, to some extent, of modeling the split memory model of the PMA mechanism: simply assume that the expressions of MiniML reside in the protected memory and that the expressions of MiniML reside in the unprotected memory. By extension the MiniML\textsuperscript{a} expression $\textsf{AS}^\tau e$ represents an entry point to the protected memory where the MiniML expressions reside and the MiniML expression $\textsf{SA} e$ represents a call to the unprotected memory where the expressions of MiniML\textsuperscript{a} reside.

This model, however, is not precise enough. A first issue is the use of $\textsf{AS}^\tau e$ to model the entry point mechanism of PMA. Reconsider our previously problematic MiniML\textsuperscript{M}-context $C_a$. Whether or not this context is able to distinguish between the two contextually equivalent terms when $\textsf{AS}^\tau e$ is assumed to be entry point to a protected piece of memory relies, in practice, on what the binary values of the entry points are. The MiniML\textsuperscript{M}-calculus, however, does not allow us to reason about the binary values of the entry points. This limitation also artificially restricts the attacker model: in the MiniML\textsuperscript{M}-calculus the attacker can only manipulate the entry points that have been shared with it. In practice, however, an attacker is capable of calling any existing entry point by guessing its address.

![Diagram](image)

*Figure 3.1.* The sub-terms of: $\lambda y : \tau. \textsf{SA} ((\lambda x.t) \textsf{AS}^\tau y)$ are spread across protected and unprotected memory.

A second issue is the way the MiniML\textsuperscript{M}-calculus wraps the insecure functions it gains access to. As illustrated in Figure 3.1, a memory representation of what happens when a MiniML program receives a $\lambda$-term $\lambda x.t$ from the MiniML\textsuperscript{a} attacker, every time a MiniML program is given a MiniML\textsuperscript{a} function it wraps that function into a lambda function of its own and as a result writes out a new chunk of memory to the unprotected memory space. As required by MiniML\textsuperscript{M} this memory chunk encodes an application of the received function to an entry point to the bound variable of the enclosing $\lambda$-term.
The problem with this memory chunk is that the attacker in the unprotected memory will be able to observe, copy and execute that memory chunk before the shared function is used, if it is ever used at all. Given that this shared memory chunk provides access to an internal variable of the secured MiniML program, the attacker can thus abuse it to break the confidentiality properties of the MiniML program that resides in the protected memory. The MiniML$^M$-calculus is not capable of modeling such an attack, thus leaving the consequences of the attack open to the implementation. We argue that this wrapping approach to modelling function calls raises more questions and possible security problems than it resolves.

3.4 The MiniML$^+$-Calculus: a Secure FFI

The MiniML$^+$-calculus models the secure interoperation between MiniML and MiniML$^a$ in a manner that secures the MiniML program from the MiniML$^a$ attacker (Section 3.4.1). The MiniML$^+$-calculus introduces new syntax (Section 3.4.2), new operational semantics (Section 3.4.3) and a modified notion of type soundness (Section 3.4.4).

3.4.1 Our Approach

To formalise a secure interoperation between the attacker and the source language the MiniML$^+$-calculus applies the following three insights.

Separated Program States

In this thesis we protect functional programs from machine level attackers by employing the PMA memory isolation mechanism that prevents the attacker from directly accessing the memory of the program being secured (Section 2.6). To that end the program state $P$ of MiniML$^+$ is split into two substates: the attacker state $A$ and the secured program state $M$ that incorporates the MiniML program. Formally a program $P$ is defined as a combination of both.

$$P = A \parallel M$$

Call Stacks

To ensure that the program state is separable, the combined language must encode the interaction between both languages. To do so each state is extended with a call stack. The secure state $M$ encodes this call stack as a type annotated stack of evaluation contexts $\Sigma$:

$$\Sigma ::= E : \tau \rightarrow \tau' \mid \varepsilon$$
where $\mathbf{E}$ denotes a sequence of evaluation contexts $E$ that represent the continuation of computation when a call to the attacker returns and are thus only to be filled in by input originating from the attacker. The stack of evaluation contexts is type annotated. In MiniML$^+$ these annotated types are incorporated into dynamic type checks to ensure that the input from the attacker does not break the type safety of the original MiniML program.

In contrast the attacker encodes the call stack through a sequence of contexts $\mathbf{C}$ not a sequence of evaluation contexts $\mathbf{E}$. An evaluation context $E$ is derived from call-by-value semantics, which limits the hole $[\cdot]_\Omega$ to certain sub-terms. The evaluation context $E$ is thus a less powerful threat to full-abstraction than the context $\mathbf{C}$, where the hole can be anywhere. More specifically, for each possible pair of expressions $e_1$ and $e_2$ received from the MiniML program there exists a context $\mathbf{C}_a$ of the form:

$$(\text{if } (e_1 \equiv_{\alpha} [\cdot]) \Omega \text{ true})$$

that can distinguish them.

Reference Objects

To ensure that the state of the MiniML program is isolated from any kind of inspection by the attacker, the terms of MiniML programs that introduce equivalences/abstractions: namely $\lambda$-terms and locations, should not be shared directly with attacker. Instead, those terms are shared by providing the attacker with reference objects, objects that refer to the original terms of the program in MiniML. These reference objects have two purposes. Firstly they mask the contents of the original term and secondly they enable MiniML$^+$ to keep track of which locations or $\lambda$-terms have been shared with the attacker. The MiniML$^+$-calculus models reference objects for $\lambda$-terms and locations through sets of names $n^f_i$ and $n^l_i$ respectively. Both sets of names are tracked in the secure state through a map $\mathbf{N}$ that records not only the associated term but also the associated type, thus enabling MiniML$^+$ to perform run-time type checks on the attackers interactions with these names. Formally $\mathbf{N}$ is defined as.

$$\mathbf{N} ::= * \mid \mathbf{N}, n^f_i \mapsto (e, \tau) \mid \mathbf{N}, n^l_i \mapsto (e, \tau)$$

A name $n^f_i$ is created deterministically every time a $\lambda$-term is shared between the secure state and the attacker. The name $n^f_1$ refers to the first shared $\lambda$-term, the name $n^f_2$ refers to the second shared $\lambda$-term (even if it is the exact same $\lambda$-term as the first one) and so forth.

In contrast the index $i$ of the name $n^l_i$ will correspond to the index of a location $l_i$ as follows.

$$n^l_i \mapsto l_i$$

This mapping between the indices of the names and the locations is necessary because the index operation in MiniML allows a MiniML context/attacker...
to observe the index of the location, as illustrated in Example Ex-2 of Section 3.1.1. This observational power should thus not be taken away from the attacker in MiniML$^+$. Note that the names $n_f^i$ and $n_l^i$ are terms of the MiniML$^a$-calculus but not of MiniML. We do not compile or translate the $\lambda$-terms and locations of the source MiniML programs into these names. Such an approach would violate the contextual equivalences of MiniML [79]. These names $n_f^i$ and $n_l^i$ are only present in the MiniML$^a$-calculus and are created at run-time as reference objects to $\lambda$-terms and locations.

3.4.2 The Syntax of MiniML$^+$

The syntax of MiniML$^+$ extends both the syntax of the MiniML$^a$ and the syntax of MiniML with new language constructs that enable secure interoperation. It also introduces a new class of expressions called marshalling terms.

**Extensions to MiniML$^a$**

While basic values such as numbers and booleans can simply be converted to the correct representation when exchanged (Section 3.4.3), no such conversion is possible for $\lambda$-terms and locations $l_i$. As detailed in Section 3.4.1, in MiniML$^+$ the MiniML$^a$ attacker is restricted to reference objects formalized as names $n_f^i$ and $n_l^i$ that refer to $\lambda$-terms and locations shared by the MiniML program, respectively.

The MiniML$^a$ attacker can compare these names through its $\alpha$-equivalence testing operator $\equiv_{\alpha}$ and can also apply, read and write them in MiniML$^+$ using the newly added terms:

\[
\text{call } n_f^i \text{ v } \text{deref } n_l^i \text{ set } n_l^i \text{ v}
\]

respectively. The attacker can also create new names $n_l^i$, that point to freshly allocated locations $l_i$ in the MiniML program, through the following expression:

\[
fref^\tau \text{ v}
\]

where $\tau$ represent the MiniML type that the attacker promises the value $v$ conforms to. This promise is checked at run-time within MiniML$^+$.

The syntax of MiniML$^a$ is thus extended as follows.

| Expressions: $e$ ::= ... | call $e_1$ $e_2$ | set $e_1$ $e_2$ | deref $e$ | fref$^\tau$ $e$ |
| Values: $v$ ::= ... | $n_f^i$ | $n_l^i$ |
| Evaluation Contexts: $E$ ::= ... | call $E$ $e$ | call $v$ $E$ | set $E$ $e$ | set $v$ $E$ | deref $E$ | fref$^\tau$ $E$ |
Extensions to MiniML

In contrast to the six additional terms to MiniML\(^a\), the terms of MiniML are only extended with one new value:

\[ \tau F(\lambda x. e) \]

that embeds a MiniML\(^a\) \(\lambda\) -term into a MiniML program, modelling an attacker function that the MiniML program can use. The type \(\tau\) is included with the value to enable MiniML\(^+\) to type check the use of this attacker function at run-time.

The MiniML-calculus is not extended with a term to embed locations of MiniML\(^a\) into MiniML programs, as MiniML programs manipulating the attacker memory harms the full abstraction result. This does not harm the inter-operation, as the attacker can simply create an MiniML location through the \texttt{ref}\(^\tau\) \(e\) term instead of sharing its own.

The syntax of MiniML is thus extended as follows.

Values: \(v ::= \ldots \mid \tau F(\lambda x. e)\)

Marshalling terms

The marshalling process of MiniML\(^+\) converts MiniML values to MiniML\(^a\) values and vice versa, within the secure state \(M\). The marshalling terms \(m\) are defined as follows.

\[ m ::= v \mid v \mid \langle m_i \in 1..n \rangle \]

Marshalling converts MiniML values to MiniML\(^a\) values and vice versa and thus includes all values of both. Marshalling a tuple of size \(n\) is not an immediate action but instead takes \(n\) steps, one for each value defined in the tuple. The marshalling tuple \(\langle m_i \in 1..n \rangle\) is thus needed to capture the intermediate state, where some members of the tuple have already been converted and others members of the tuple are yet to be converted.

3.4.3 Operational Semantics

The reduction rules of the MiniML-calculus are denoted as:

\[ P \rightarrow P' \]

where as described in Section 3.4.1 a program \(P = M \parallel A\) composes two states \(M\) and \(A\). The secure state \(M\) has four different sub-states.

\[
\begin{align*}
(1) \quad & N; \mu \vdash \Sigma \circ e : \tau \\
(2) \quad & N; \mu \vdash \Sigma \triangleright m : \tau \\
(3) \quad & N; \mu \vdash \Sigma \triangleleft m : \tau \\
(4) \quad & N; \mu \vdash \Sigma 
\end{align*}
\]

The secure state \(M\) is either (1) executing an expression \(e\) of type \(\tau\), (2) marshalling out values, (3) marshalling in input from the attacker that is expected
to be of type $\tau$ or (4) waiting on input from the attacker, where $N$ is the name map, $\mu$ the store of locations, $\Sigma$ the type annotated call stack and $m$ are the marshalling terms.

The attacker state $A$ has two different sub-states:

1. $A = \mu \vdash C \circ t$
2. $A = \mu \vdash C$

The attacker state $A$ is either (1) it executes a term $t$ or (2) is suspended waiting on input from the MiniML program, where $\mu$ is the attacker store, and $C$ the stack of attacker contexts.

Note that the attacker state $A$ and the secure state $M$ never execute at the same time. During the evaluation of a program state $P$, the attacker or the secure state will thus be suspended.

We divide the reduction rules over the program state $P$ into four categories: internal computations, marshalling values in, marshalling value out and cross-boundary commands.

**Internal Computations**

Internal computations are reduction rules that only affect the terms of one of the two languages. These are thus simply the reduction rules of MiniML and MiniML$^a$ set within the program state of MiniML$^+$. 

$$
\begin{align*}
\text{(Internal MiniML)} \\
\mu \mid e \rightarrow \mu' \mid e' \\
\mu \vdash C \mid N; \mu \vdash \Sigma \circ e : \tau \rightarrow \mu \vdash C \mid N; \mu \vdash \Sigma \circ e' : \tau
\end{align*}
$$

$$
\begin{align*}
\text{(Internal MiniML$^a$)} \\
\mu \mid e \rightarrow \mu' \mid e' \\
\mu \vdash C \circ e \mid N; \mu \vdash \Sigma \rightarrow \mu' \vdash C' \circ e' \mid N; \mu \vdash \Sigma
\end{align*}
$$

Note that unlike the internal computations of MiniML the internal computations of MiniML$^a$ can modify its call stack. This is because as previously mentioned the attacker reduces to wrong when dealing with errors.

$$
\mu \vdash C \circ E[\text{wr}] \mid N; \mu \vdash \Sigma \rightarrow \mu \vdash \text{wr} \mid N; \mu \vdash \Sigma \quad \text{(A-Wr)}
$$

In practice, a low-level attacker can recover from errors. In MiniML$^+$ however, whenever something goes wrong the attacker is to blame. This similar to how in Wadler’s and Findler’s blame calculus [84] blame is assigned to the less-precise portion of a program. As detailed further on, when an error is the result of an attempt by the attacker to take advantage of the foreign function interface, the MiniML$^+$ program is terminated by clearing out the ML-configuration. As such the actions of an attacker after something has gone wrong are not relevant to the overall security result.
Marshalling Values Out

Whenever the embedded MiniML program reduces to a value \( v \), that value needs to be converted to the appropriate representation before it is shared with the head of the attacker’s call stack \( \mathcal{C} \). If the value is a location or a \( \lambda \)-term then it must be masked with a name \( n_l \) or \( n_f \), and the association between the name, the term and the term’s type recorded in the map \( N \). Otherwise, the value is simply converted to the corresponding MiniML value. This conversion happens in a designated marshalling state as follows.

\[
\mu \not\vdash \mathcal{C} || N; \mu \not\vdash \Sigma \circ v : \tau \rightarrow \mu \not\vdash \mathcal{C} || N; \mu \not\vdash \Sigma \triangleright v : \tau
\] (Setup)

In the following marshalling out rules, we have compressed the full marshalling out state:

\[
\mu \not\vdash \mathcal{C} || N; \mu \not\vdash \Sigma \triangleright m : \tau
\]

into a wrapper:

\[\llbracket m \rrbracket^N\]

that simplifies the formalisation by denoting the only two construct relevant to the marshalling process, the marshalling term \( m \) and the map of shared names \( N \). The marshalling out rules are as follows.

(\text{Marshall-Out-Bool}) \quad (\text{Marshall-Out-Unit})

\[
\begin{array}{c}
\llbracket b \rrbracket^N \rightarrow \llbracket b \rrbracket^N \\
\llbracket \text{unit} \rrbracket^N \rightarrow \llbracket \text{unit} \rrbracket^N
\end{array}
\] (\text{Marshall-Out-Int}) \quad (\text{Marshall-Out-Location})

\[
\begin{array}{c}
\llbracket n \rrbracket^N \rightarrow \llbracket n \rrbracket^N \\
\llbracket l_i \rrbracket^N \rightarrow \llbracket n_f^i \rrbracket^N
\end{array}
\] (\text{Marshall-Out-Tuple})

\[
\forall i \in 1..n. \llbracket v_i \rrbracket^N_{i-1} \rightarrow \llbracket v_i \rrbracket^N_i
\] (\text{Marshall-Out-Lambda})

\[
\llbracket (\lambda x : \tau.e) \rrbracket^N \rightarrow \llbracket n_f^j \rrbracket^N
\] (\text{Marshall-Out-Foreign})

\[
\tau = \tau_1 \rightarrow \tau_2
\]

Note the tuple conversion rule \text{Marshall-Out-Tuple}: it converts every member individually, ensuring that the embedded \( \lambda \)-terms and locations are converted into to the correct names.
In the lambda conversion rule Marshall-Out-Lambda, the index \( j \) of the name \( n^j \) is computed from the number of \( \lambda \)-functions that are stored in the map \( N: |N| \).

There is no error possible during these conversions as the MiniML type system ensures that the value will be of the correct type. Any possible type errors are produced when marshalling values in from the attacker. This is similar to Matthew’s and Findler’s where Multi-Language semantics where dynamic typechecks on the output of the typed language are not required [55].

Once the marshalling succeeds the result is shared with the attacker and the marshalling out state is dismantled as follows.

\[
\mu \vdash C, C 
\mid N; \mu \vdash \Sigma \triangleright v : \tau \mapsto \mu \vdash C \circ [v] \mid N; \mu \vdash \Sigma \quad \text{(Share)}
\]

**Marshalling Values In**

Whenever the attacker reduces to a value and the secure state’s call stack \( \Sigma \) is not empty the value is input into the secure state.

\[
\mu \vdash C \circ v 
\mid N; \mu \vdash \Sigma, E : \tau \rightarrow \tau' \mapsto \mu \vdash C \mid N; \mu \vdash \Sigma \triangleleft v : \tau \quad \text{(Input)}
\]

The input value must be marshalled to the correct representation before it is plugged into the head of the stack of evaluation contexts \( \Sigma \). As denoted in the reduction rule Input the marshalling rules will also verify that the input value matches the argument type \( \tau \) of the to be plugged evaluation context.

In the following marshalling out rules, we have, again, compressed the full marshalling in state:

\[
\mu \vdash C 
\mid N; \mu \vdash \Sigma \triangleleft m : \tau
\]

into a wrapper:

\[
\llbracket m \rrbracket^N_{\tau}
\]

that simplifies the formalisation by denoting the only two constructs relevant to the marshalling process, besides the marshalling term \( m \): the expected type \( \tau \) and the map of shared names \( N \).

The marshalling in reduction rules are analogous to the previously detailed marshalling out reductions in that they perform the reverse operation: they convert the input into the appropriate MiniML representation, fetching names from the map \( N \) instead of introducing names. Additionally the marshalling in rules incorporate rules to capture the incorrect inputs from the attacker. The marshalling in rules are as follows

\[
\begin{align*}
\text{(Marshall-In-Bool)} & & \text{(Marshall-In-Bool-Error)} \\
\llbracket b \rrbracket^N_{\text{Bool}} \rightarrow \llbracket b \rrbracket^N_{\text{Bool}} & & \llbracket v \neq b \rrbracket^N_{\text{Bool}} \rightarrow \llbracket \text{wr} \rrbracket^N_{\text{Bool}}
\end{align*}
\]
The cross boundary commands enable the MiniML program to manipulate shared \( \lambda \)-terms as follows.

\[
\mu \not\vdash \overline{C}, C \mid N; \mu \vdash \Sigma \omega \omega E : \tau \rightarrow \nu : \tau \rightarrow \mu \not\vdash \overline{C}, C[E[v]] : \tau
\]

\[\text{(M-Call)}\]

As listed in (M-Call), a MiniML program is able to apply a MiniML \( \lambda \)-term to a MiniML value. The application is done in two steps as it consists of
two components: the shared λ-term and an argument v. In the first step an
evaluation context that consists of an application of the shared λ-term to a hole
[·] is placed inside the context C while the secure state is setup for marshalling.
In a second step the argument v is then marshalled out as described previously
and plugged into the newly constructed evaluation context after which control
is reverted to the attacker.

Note that this cross boundary function application serves as an input to
the attacker as the application is plugged into the top context/attack
C. This is because the attacker must be able to inspect this function call as accurately as the machine-level attacker who is able to observe
which of its functions are called with what arguments.

There is no cross boundary command that enables the MiniML program
to dereference or assign memory locations of the attacker as these are not
accepted as input from the attacker. As explained in what follows, we make
up for this limitation by enabling the attacker to allocate new locations on the
secure side.

The cross boundary commands enable the attacker to manipulate shared
MiniML λ-terms and locations as follows.

\[
\begin{align*}
\mu \vdash \overline{\text{C}} \circ \text{call } n_l^f v \mid \mid N; \mu \vdash \Sigma \Rightarrow \mu \vdash \overline{\text{C}} \circ v \mid \mid N; \mu \vdash \Sigma, (e [\cdot]) : \tau \rightarrow \tau' \\
\text{where } N(n_l^f) = (e, \tau \rightarrow \tau') & \quad \text{(A-Call)} \\
\mu \vdash \overline{\text{C}} \circ \text{set } n_l^f v \mid \mid N; \mu \vdash \Sigma \Rightarrow \mu \vdash \overline{\text{C}} \circ v \mid \mid N; \mu \vdash \Sigma, (e := [\cdot]) : \tau \rightarrow \text{Unit} \\
\text{where } N(n_l^f) = (e, \text{Ref } \tau) & \quad \text{(A-Set)} \\
\mu \vdash \overline{\text{C}} \circ \text{derefer } n_l^f \mid \mid N; \mu \vdash \Sigma \Rightarrow \mu \vdash \overline{\text{C}} \mid N; \mu \vdash \Sigma \circ !l : \tau \\
\text{where } N(n_l^f) = (l, \text{Ref } \tau) & \quad \text{(A-Der)} \\
\mu \vdash \overline{\text{C}} \circ \text{derefer } n_l^f v \mid \mid N; \mu \vdash \Sigma \Rightarrow \\
\mu \vdash \overline{\text{C}} \circ v \mid \mid N; \mu \vdash \Sigma, (\text{derefer } [\cdot]) : \tau \rightarrow \text{Ref } \tau & \quad \text{(A-Ref)} \\
\mu \vdash \overline{\text{C}}, C \circ \text{derefer } n_l^f \mid \mid N; \mu \vdash \Sigma \Rightarrow \mu \vdash \overline{\text{C}} \circ C[\text{wr}] \mid \mid \ast \theta \vdash \epsilon \\
\text{where } n_l^f \notin \text{dom}(N) \text{ or } N(n_l^f) = (e, \tau \rightarrow \tau') & \quad \text{(A-WrD)} \\
\mu \vdash \overline{\text{C}}, C \circ \text{call } n_l^f v \mid \mid N; \mu \vdash \Sigma \Rightarrow \mu \vdash \overline{\text{C}} \circ C[\text{wr}] \mid \mid \ast \theta \vdash \epsilon \\
\text{where } n_l^f \notin \text{dom}(N) \text{ or } N(n_l^f) = (e, \text{Ref } \tau') & \quad \text{(A-WrC)} \\
\mu \vdash \overline{\text{C}}, C \circ \text{set } n_l^f v \mid \mid N; \mu \vdash \Sigma \Rightarrow \mu \vdash \overline{\text{C}} \circ C[\text{wr}] \mid \mid \ast \theta \vdash \epsilon \\
\text{where } n_l^f \notin \text{dom}(N) \text{ or } N(n_l^f) = (e, \tau \rightarrow \tau') & \quad \text{(A-WrS)}
\end{align*}
\]

A command from the attacker is not an input to the MiniML program, but
rather a task it must carry out, and is as such not plugged into the head of
the stack of evaluation contexts Σ, but is instead executed on top the stack.
As was the case for the function application by a MiniML program, applying
a λ-term (A-Call), writing to a shared location (A-Set) or referencing a new
location (A-Ref) requires two steps. In the first step a new evaluation context
is constructed. In the second the argument is marshalled out as described previously.

Every time the command does not conform to the typing rules of MiniML the attacker is updated to the error state \( \text{wr} (A\text{-WrD}, A\text{-WrC}, A\text{-WrS}) \). Dereferencing a shared MiniML location \((A\text{-Der})\) requires just one step as it involves only the shared name \( n_1 \) and thus does not need to marshall out a value.

Note that in each of these rules the current evaluation context of the attacker \((E)\) is discarded. While discarding this evaluation context changes the way MiniML\(^a\) operates within the FFI, in some sense derailing its execution model, that does not affect its usefulness as an attacker model. On the contrary, we remove the evaluation context to strengthen the attacker model. As detailed in Section 3.4.1, the contexts \( C \) of the attackers call stack \( \overline{C} \) pose a real threat to the abstractions MiniML, whereas an evaluation context \( E \) does not.

### 3.4.4 Type Soundness

An important concern for MiniML\(^+\) is whether or not it is type sound, that well typed MiniML\(^+\) programs do not go wrong. Only the secure state \( M \) of a program \( P \) must be type sound as the attacker MiniML\(^a\) is type unsafe by design (Section 3.2). It is, however, required that only one of the two sub-states is active.

\[
\frac{\Gamma \vdash M \quad M \neq N; \mu \Rightarrow \Sigma}{\Gamma \vdash \mu \Rightarrow \overline{C} | M}
\]

\[
\frac{\Gamma \vdash N; \mu \Rightarrow \Sigma}{\Gamma \vdash \mu \Rightarrow \overline{C} \circ e | N; \mu \Rightarrow \Sigma}
\]

The secure state \( M \) is well typed if the reducing expression is well typed, each individual evaluation context of the secure state’s evaluation stack \( \Sigma \) is well typed and also each association in the state’s map \( N \) and each location in the secure store \( \mu \) is well typed.

\[
\frac{\Gamma \vdash N \quad \Gamma \vdash \mu \quad \Gamma \vdash \Sigma}{\Gamma \vdash N; \mu \Rightarrow \Sigma}
\]

\[
\frac{\Gamma \vdash N; \mu \Rightarrow \Sigma; \circ e : \tau}{\Gamma \vdash N; \mu \Rightarrow \Sigma \circ e : \tau}
\]

\[
\frac{\Gamma \vdash N; \mu \Rightarrow \Sigma}{\Gamma \vdash N; \mu \Rightarrow \Sigma \circ e : \tau}
\]

\[
\frac{\Gamma \vdash N \quad \Gamma \vdash e : \tau}{\Gamma \vdash N; [n_i \mapsto (e, \tau)]}
\]

\[
\frac{\Gamma \vdash \star}{\Gamma \vdash \\star}
\]
(Type-Store)  Γ ⊢ µ  Γ ⊢ ν : τ
(Δ)  Γ ⊢ µ, l_i = ν
(Type-Stack)  Γ ⊢ Δ, x : τ_1 ⊢ E[x] : τ_2
(ΓΣ)  Γ ⊢ Δ, E : τ_1 → τ_2
(Δ)  Γ ⊢ ε

Note that in the well-typedness rule for stacks (Type-Stack), we type the holes by filling them with a variable that has the argument type τ_1. Note also that in the well-typedness rules for the marshalling states (Type-Marshalling-Out, Type-Marshalling-In), we do not type check the marshalling term m as this is handled by the run-time typechecks provided by the marshalling rules (Section 3.4.3).

Type checking the MiniML expression of the secure state M is done through the regular MiniML typing rules extended with one additional rule for type checking the additional value \( \text{F} (\lambda x. t) \) that embeds a MiniML\( ^a \) λ-term.

\[
\text{(Type-Foreign-Lambda)}
\]

Γ ⊢ \text{F}(\lambda x. t) : τ

Proving Type Soundness

Because it is only the secure state M that must be type sound, we cannot rely on the traditional notion of type soundness in the style of Wright and Felleisen [88]. Instead, we establish that whenever a program gets stuck or reduces to the error \( \text{wr} \) the attacker is at fault, similar to how Wadler’s and Findler’s blame calculus [84] assigns blame to the less precise part of the program.

In what follows type soundness is, as usual, split into two theorems: type progress and type preservation. Our proof of preservation relies on the completeness of the dynamic type checks, which we prove in the following lemma.

Lemma 1 (Completeness of the Dynamic Type checks)

Given \( \Gamma \vdash N \) we have that either:

1. \( \| v \|_\tau^N \rightarrow^* \| v \|_\tau^N \) where \( \Gamma \vdash v : \tau \)
2. \( \| v \|_\tau^N \rightarrow^* \| \text{wr} \|_\tau^N \)

The proof of this lemma proceeds by induction on the structure of \( \tau \) and is available in Appendix A.1.

Theorem 1 (Type Preservation)

Given \( \Gamma \vdash A \parallel M \) and \( A \parallel M \rightarrow A' \parallel M' \) we have that: \( \Gamma \vdash A' \parallel M' \).

The proof proceeds by induction on the derivation of \( \Gamma \vdash M \), when addressing the marshalling states of MiniML\( ^+ \) we rely on Lemma 1 to prove the thesis. A full proof is available in Appendix A.2.
Next we prove the type progress theorem for MiniML$^+$. 

**Theorem 2 (Type Progress)**

*Given* $\Gamma \vdash M$ *then either*

1. $A || M \rightarrow A' || M'$
2. $A || M \rightarrow A || M'$
3. $A || M \rightarrow A' || M$
4. $A || M \rightarrow \mu \vdash \text{wr} || \star, \emptyset \vdash \varepsilon$
5. $\mu \vdash \varepsilon || M \not\rightarrow A' || M'$
6. $\mu \vdash C \circ \varepsilon || N, \mu \vdash \varepsilon \not\rightarrow A' || M'$

The proof proceeds by induction on a derivation of $\Gamma \vdash M$. A full proof is available in Appendix A.3.

### 3.4.5 Contextual Equivalence

Definition 3 states that a context $C$ is an MiniML$^+$ program with a single hole $[\cdot]$, and two MiniML$^+$ programs $P_1$ and $P_2$ are contextually equivalent if and only if there is no context $C$ that can distinguish them.

When defining contextual equivalence over MiniML$^+$ we must, however, make some slight changes to the definition of the contexts $C$. The MiniML$^+$-calculus program state $P$ combines the secure state $M$ and the attacker state $A$. Our definition of contextual equivalence aims only to relate the secure states $M$ that embed the MiniML program, as preserving the security properties of MiniML within MiniML$^+$ from the attacker MiniML$^a$ is the goal of this chapter. The attacker states $A$ thus serve as the contexts $C$ in which the secure state $M$ operates. We formally define contextual equivalence over MiniML$^+$ as follows.

**Definition 12** Contextual equivalence over MiniML$^+$ ($\simeq^+$) is defined as:

$$M_1 \simeq M_2 \overset{\text{def}}{=} \forall A. (A || M_1)\uparrow \iff (A || M_2)\uparrow.$$ 

**Examples**

Consider, for example, the two contextually equivalent $\lambda$-terms of Example Ex-1 in Section 3.1.1. These $\lambda$-terms remain contextually equivalent when placed within secure states $M$ as follows.

$$\star; \emptyset \vdash \varepsilon \circ (\lambda x : \text{Int.} \overline{0}) \quad \star; \emptyset \vdash \varepsilon \circ (\lambda x : \text{Int.}(x - x))$$

There exists no attacker $A$ that can distinguish these two secure states. The marshalling out rule for $\lambda$-terms (Marshall-Out-Lambda) will convert both $\lambda$-terms to a name $n_f^1$ as they are the first $\lambda$-term to be shared with the attacker. An attacker $A$ will in both cases observe that name, but cannot observe that
the names refer to $\lambda$-terms that are not $\alpha$-equivalent as, due to the dynamic type checking rules of MiniML$^+$, it can only apply the name $n^1_f$ to numbers $\bar{n}$ as in MiniML.

Alternatively, the following two secure states are not contextually equivalent:

$$\text{\ast}, n^1_f \mapsto (\lambda x : \text{Ref Int.} \bar{1}); \emptyset \models \epsilon \quad \text{\ast}, n^1_f \mapsto (\lambda x : \text{Ref Int.} !x); \emptyset \models \epsilon$$

as the following context/attacker $A$:

$$A = (\emptyset \models (\text{if}(\bar{1} == [\cdot]) \Omega \text{true}), \text{call } n^1_f [\cdot] \circ \text{ref} \text{Int} \bar{2})$$

can distinguish between them. Reducing $\text{ref} \text{Int} \bar{2}$

will result in both secure states, as dictated by Marshall-Out-Location, returning a name $n^1_l$ associated with a location $l_1$ that is assigned the value two.

$$l_1 \mapsto \bar{2}$$

This name $n^1_l$ serves as input to the second context on the stack $\overline{\text{C}}$:

$$\text{call } n^1_f [\cdot]$$

where the name $n^1_l$ is applied to it. This name $n^1_l$ refers to different $\lambda$ terms in both examples.

$$n^1_f \mapsto (\lambda x : \text{Ref Int.} \bar{1}) \quad n^1_f \mapsto (\lambda x : \text{Ref Int.} !x)$$

These two $\lambda$-terms are not contextually equivalent in MiniML, a fact that is subsequently observed by the last member of $\overline{\text{C}}$: $(\text{if}(\bar{1} == [\cdot]) \Omega \text{true})$.

### 3.5 A Fully Abstract Translation

To establish that the MiniML$^+$-calculus is capable of securing the contextual equivalences of MiniML from the attacker MiniML$^a$, we show that their exists a translation from MiniML to MiniML$^+$ that preserves the contextual equivalences of MiniML. Direct proofs over contextual equivalence are, however, difficult as one needs to reason about every possible context \[7, 27, 40, 66\]. To work around that challenge, we first develop bisimilarity relations that coincide with contextual equivalence for MiniML (Section 3.5.1) and for MiniML$^+$ (Section 3.5.2). Proving that their exists a fully abstract translation from MiniML to MiniML$^+$ is done by relating these bisimilarity relations (Section 3.5.3).
3.5.1 Congruent Bisimilarity for MiniML

We start by defining a bisimilarity \( \approx^M \) over the programs of MiniML that is congruent: it coincides with the contextual equivalence relation \( \simeq^M \) over MiniML. There have been multiple different bisimilarities and trace semantics over typed \( \lambda \)-calculi with references. In this chapter we use an applicative bisimilarity that is a combination of Jeffrey’s and Rathke’s applicative bisimilarity for the vref-calculus [41] and the fully abstract trace semantics for the \( \lambda\mu \) hashref-calculus by Jagadeesan [40].

As detailed in Section 2.5.1, a bisimilarity is defined through a labelled transition system. This LTS models the interactions between a MiniML context \( C \) and a MiniML program. The LTS is formally defined as a triple: \( (\zeta, \alpha, \xrightarrow{\alpha}) \).

The state:

\[
\zeta \overset{\text{def}}{=} K; \mu \mid e : \tau
\]

is the MiniML run-time state \( \mu \mid e \) and its type \( \tau \) extended with a sequence \( K \) that records the locations \( l_i \) that the opponent has knowledge of.

\[
K ::= (l_i, \text{Ref } \tau) \mid \varepsilon
\]

This is needed to capture fact that locations are not part of the static semantics and thus do not appear in contexts unless made available at run-time. The labels \( \alpha \) of the LTS are defined as follows.

\[
\alpha ::= \gamma \mid \tau \quad \gamma ::= \@v \mid .i \mid l_i := v \mid \text{ref } v \mid !l_i \mid l_i \mid b \mid \overline{n} \mid \text{unit}
\]

The labelled reductions are of the form:

\[
\mu \mid e : \tau \xrightarrow{\gamma} K' ; \mu' \mid e' : \tau'
\]

and are formalised as follows:

\[
\frac{\text{(Sil)}}{K; \mu \mid e \rightarrow K' ; \mu' \mid e' : \tau}
\]

\[
K; \mu \mid \overline{n} : \text{Int} \xrightarrow{\overline{n}} K; \mu' \mid \overline{n} : \text{Int} \quad (\text{O-N})
\]

\[
K; \mu \mid \text{unit} : \text{Unit} \xrightarrow{\text{unit}} K; \mu \mid \text{unit} : \text{Unit} \quad (\text{O-U})
\]

\[
K; \mu \mid \text{true} : \text{Bool} \xrightarrow{\text{true}} K; \mu \mid \text{true} : \text{Bool} \quad (\text{O-T})
\]

\[
K; \mu \mid \text{false} : \text{Bool} \xrightarrow{\text{false}} K; \mu \mid \text{false} : \text{Bool} \quad (\text{O-F})
\]

\[
K; \mu \mid (\langle \nu_i \in 1..n \rangle : \langle \tau_i \in 1..n \rangle \xrightarrow{i} ) K; \mu \mid \nu_i : \tau_i \quad (\text{I-Proj})
\]

\[
K; \mu \mid (\lambda x : \tau. e) : \tau \rightarrow \tau' \xrightarrow{\@l_i} K; \mu \mid ((\lambda x : \tau. e) \ l_i) : \tau' \quad (\text{I-AppL})
\]

where \( K \vdash l_i : \tau \) and \( l_i \in K \).
\[ K; \mu \mid (\lambda x : \tau.e) : \tau \rightarrow \tau' \xrightarrow[@{v \in 1..n}] K; \mu \mid ((\lambda x : \tau.e) \langle v_i \in 1..n \rangle) : \tau' \]

where \( K \vdash v : \tau \) and \( \forall \ l_i \in \langle v_i \in 1..n \rangle, l_i \in K \)

(l-AppT)

\[ K; \mu \mid (\lambda x : \tau.e) : \tau \rightarrow \tau' \xrightarrow[v] K; \mu \mid ((\lambda x : \tau.e) v) : \tau' \]

where \( K \vdash v : \tau \) and \( v \neq l_i \) and \( v \neq \langle v_i \in 1..n \rangle \)

(l-App)

\[ K; \mu \mid v : \tau \xrightarrow[l_i := l_i'] K; \mu \mid l_i := l_i' : \text{Unit} \]

where \( l_i \in K \) and \( l_i' \in K \) and \( K \vdash l_i : \text{Ref} \ \tau \) and \( K \vdash l_i' : \tau \)

(l-SetL)

\[ K; \mu \mid v : \tau \xrightarrow[l_i := \langle v_i \in 1..n \rangle] K; \mu \mid l_i := \langle v_i \in 1..n \rangle : \text{Unit} \]

where \( l_i \in K \) and \( l_i' \in \langle v_i \in 1..n \rangle \), \( l_i' \in K \) and \( K \vdash l_i : \text{Ref} \ \tau \)

and \( K \vdash \langle v_i \in 1..n \rangle : \tau \)

(l-SetT)

\[ K; \mu \mid v : \tau \xrightarrow[l_i := v'] K; \mu \mid l_i := v' : \text{Unit} \]

where \( l_i \in K \) and \( K \vdash l_i : \tau \) and \( K \vdash v : \tau \) and \( v \neq l_i \) and \( v \neq \langle v_i \in 1..n \rangle \)

(l-Set)

\[ K; \mu \mid v : \tau \xrightarrow[l_i] K; \mu \mid !l_i : \tau' \quad \text{where} \ l_i \in K \ 	ext{and} \ K \vdash l_i : \text{Ref} \ \tau' \]

(l-D)

\[ K; \mu \mid l_i : \tau \xrightarrow[l_i] K, (l_i, \tau); \mu \mid l_i : \tau \]

(O-L)

\[ K; \mu \mid v : \tau \xrightarrow[\text{ref} \ v'] K; \mu \mid \text{ref} \ v' : \text{Ref} \ \tau' \quad \text{where} \ K \vdash v' : \tau' \]

(l-Ref)

The typing judgment \( K \vdash e : \tau \) is similar to the regular MiniML typing judgement \( \Gamma \vdash e : \tau \), except for the fact that the type of a location \( l_i \) is looked up in the typing environment \( \Gamma \).

The evaluation of MiniML expressions cannot be observed by a context and is thus labelled as silent through the label \( \tau \) (Sil). Whenever a MiniML program reduces to a value that is not a \( \lambda \)-term or tuple (as it may contain a \( \lambda \)-term), the context can observe that value (O-N,O-L,O-T,O-F,O-U). Likewise a context queries members of a tuple instead of observing it directly (l-Proj). Observing a location (O-L) is a special case as it adds a new location \( l_i \) and its type \( \tau \) to \( K \); the list of observed locations. A context interacts with a \( \lambda \)-term by applying it to values (l-App,l-AppL,l-AppT). An application is captured by three separate rules to ensure that no unshared locations enter the LTS.

A context can also dereference observed locations \( l_i \) (l-D), create new ones (l-Ref) and assign them values (l-Set).

**Bisimulation and Bisimilarity**

We define a weak bisimulation over this LTS as per Definition 6. Define the transition relation \( \zeta \overset{\tau}{\rightarrow} \zeta' \) as \( \zeta \overset{\tau}{\rightarrow}^* \zeta' \) where \( \overset{\tau}{\rightarrow}^* \) is the reflexive transitive closure of silent transitions \( \overset{\tau}{\rightarrow} \). A bisimulation over this LTS is now formally defined as follows.

**Definition 13** A relation \( \mathcal{R}^M \) is a bisimulation iff \( \zeta_1 \mathcal{R}^M \zeta_2 \) implies:

\[ \begin{align*}
\end{align*} \]
(1) Given $\zeta_1 \xrightarrow{\gamma} \zeta'_1$ there is a $\zeta'_2$ such that: $\zeta_2 \xrightarrow{\gamma} \zeta'_2$ and $\zeta'_1 \mathcal{R}^M \zeta'_2$

(2) Given $\zeta_2 \xrightarrow{\gamma} \zeta'_2$ there is a $\zeta'_1$ such that: $\zeta_1 \xrightarrow{\gamma} \zeta'_1$ and $\zeta'_1 \mathcal{R}^M \zeta'_2$

We denote bisimilarity, the largest bisimulation, as $\approx^M$. We now establish that the bisimilarity $\approx^M$ coincides with the contextual equivalence relation over MiniML ($\approx^M$).

**Theorem 3 (Congruence of the Bisimilarity)**

$$\mu_1 \mid e_1 \approx^M \mu_2 \mid e_2 \Leftrightarrow \emptyset; \mu_1 \mid e_1 : \tau \approx^M \emptyset; \mu_2 \mid e_2 : \tau$$

where $\Gamma \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau$.

A proof of this theorem is an adaption of existing results [41, 40, 35]. The proof splits the theorem into two sub-lemma: contextual equivalence implies bisimilarity (Completeness) and bisimilarity implies contextual equivalence (Soundness).

**Lemma 2 (Completeness)**

$$\mu_1 \mid e_1 \approx^M \mu_2 \mid e_2 \Rightarrow \emptyset; \mu_1 \mid e_1 : \tau \approx^M \emptyset; \mu_2 \mid e_2 : \tau$$

where $\Gamma \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau$.

Proving that MiniML contextual equivalence implies bisimilarity is done by showing that the contextual equivalence relation is itself a bisimulation, as per Gordon [35]. The full proof of this lemma is given in Appendix B.1.

**Lemma 3 (Soundness)**

$$\emptyset; \mu_1 \mid e_1 : \tau \approx^M \emptyset; \mu_2 \mid e_2 : \tau \Rightarrow \mu_1 \mid e_1 \approx^M \mu_2 \mid e_2$$

where $\Gamma \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau$.

This soundness lemma is proven by induction over the number of reduction steps (Appendix B.2).

### 3.5.2 A Congruent Bisimilarity for the MiniML$^+$-Calculus

We define a bisimilarity relation $\approx^+$ that coincides with the contextual equivalence relation ($\approx^+$). Again we rely on an applicative bisimulation defined through an LTS. The LTS is a triple:

$$(M, \alpha^+, \xrightarrow{\alpha^+})$$

where the secure states $M$ are the states, $\alpha^+$ the set of labels and $\xrightarrow{\alpha^+}$ the labelled transitions between states. The labels $\alpha^+$, which denote the observations of the attacker, are defined as follows.
The labelled reductions of the LTS are of the form:

\[ M \xrightarrow{\alpha^+} M'. \]

Although the attacker state \( A \) is not represented in these labelled reductions, the changes to the attacker state can be derived from the labels. The transitions are as follows.

\[
\begin{align*}
N; \mu \vdash \Sigma \circ e : \tau & \quad \xrightarrow{\tau} \quad N; \mu' \vdash \Sigma \circ e' : \tau \quad (S-\text{Inner}) \\
N; \mu \vdash \Sigma \circ v : \tau & \quad \xrightarrow{\tau} \quad N; \mu \vdash \Sigma \triangleright v : \tau \quad (S-\text{Setup}) \\
N; \mu \vdash \Sigma, E \triangleleft v : \tau & \quad \xrightarrow{\tau} \quad N; \mu \vdash \Sigma \circ E[v] : \tau \quad (S-\text{Plug}) \\
N; \mu \vdash \Sigma \triangleright m : \tau & \quad \xrightarrow{\tau} \quad N; \mu \vdash \Sigma \triangleright m' : \tau \quad (S-\text{MarshIN}) \\
N; \mu \vdash \Sigma \triangleright m : \tau & \quad \xrightarrow{\tau} \quad N' ; \mu \vdash \Sigma \triangleright m' : \tau \quad (S-\text{MarshOut}) \\
N; \mu \vdash \Sigma, E : \tau \quad \xrightarrow{\nu} \quad N \vdash \Sigma, E : \tau \rightarrow \tau' \triangleleft v : \tau \quad (A-\text{V}) \\
N; \mu \vdash \Sigma \triangleright v : \tau & \quad \xrightarrow{\nu} \quad N ; \mu \vdash \Sigma \quad (M-\text{V}) \\
N; \mu \vdash \Sigma \triangleright \text{wr} : \tau & \quad \xrightarrow{\text{wr}} \quad *; \emptyset \vdash \epsilon \quad (\text{Wr-O}) \\
N; \mu \vdash \Sigma \xrightarrow{\text{wr}} *; \emptyset \vdash \epsilon \quad (\text{Wr-C}) \\
N; \mu \vdash \Sigma \triangleleft \text{wr} : \tau & \quad \xrightarrow{\text{wr}} \quad *; \emptyset \vdash \epsilon \quad (\text{Wr-I}) \\
*; \mu \vdash \epsilon & \quad \xrightarrow{\sqrt{\cdot}} \quad *; \emptyset \vdash \epsilon \quad (\text{Done}) \\
N; \mu \vdash \Sigma \xrightarrow{\ref^r} \quad N; \mu \vdash \Sigma, (\ref^r [] ) : \tau \rightarrow \text{Ref } \tau \quad (A-\text{R}) \\
N; \mu \vdash \Sigma \xrightarrow{\text{inl}} \quad N; \mu \vdash \Sigma \circ !l : \tau \quad \text{where } N(\text{inl}) = (l, \text{Ref } \tau) \quad (D-\text{N}) \\
N; \mu \vdash \Sigma \xrightarrow{\text{inl}^f} \quad N; \mu \vdash \Sigma, (e [ ] ) : \tau \rightarrow \tau' \quad (C-\text{N}) \\
\quad \text{where } N(\text{inl}^f) = (e, \tau \rightarrow \tau') \\
N; \mu \vdash \Sigma \xrightarrow{\text{inl}} \quad N; \mu \vdash \Sigma, (l : [:] ) : \tau \rightarrow \text{Unit} \quad (S-\text{N}) \\
\quad \text{where } N(\text{inl}) = (l, \text{Ref } \tau) \\
N; \mu \vdash \Sigma \xrightarrow{\text{inl}^f} \quad N; \mu \vdash \Sigma, (\ell : [:] ) : \tau \rightarrow \text{Unit} \quad (C-\text{L}) \\
N; \mu \vdash \Sigma \circ E[\downarrow r \rightarrow r F(\lambda x . e) v] : \tau \xrightarrow{\rightarrow (\lambda x . e)} \quad N; \mu \vdash \Sigma, E : \tau_2 \rightarrow \tau \triangleright v : \tau_1 \\
\quad \text{where } N(\text{inl}^f) = (\ell, \text{Ref } \tau) \\
\end{align*}
\]

The internal reduction steps, the marshalling transitions, as well as the rules that setup the marshalling and plug the stack \( \Sigma \) are labelled as silent through the label \( \tau \) (S-\*). The values \( v \) that the attacker returns or inputs are decorated with \( ? \) (A-\text{V}). Likewise the inputs or returned values of the secure state,
converted to MiniML\textsuperscript{a} values \( v \) by the marshalling rules, are decorated with ! (M-V). Whenever the marshalling fails (Wr-O,Wr-I) or the attacker makes an inappropriate call (Wr-C), the transition is labelled as wrong \texttt{wr}. Dereferencing shared names is a one step transition and is labelled accordingly (D-N).

Setting and creating shared locations (S-N,A-R) or applying shared \( \lambda \)-terms (C-N,C-L) are, as detailed in Section 3.4.3, two step operations which are captured by two labels. In the first step, whose label is decorated with \( \gg \), a new context is constructed that encodes the shared term and the operation to be performed on it. In the second step the argument is passed across the boundary as captured by the value sharing rules (A-V,M-V). Note that when the secure state applies a MiniML\textsuperscript{a} (C-L) the argument is marshalled first (S-MarshOut).

**Bisimulation and Bisimilarity**

As in Section 3.5.1 we define a weak bisimulation relation. Define the transition relation \( M \xrightarrow{\gamma^+} M' \) as \( M \xrightarrow{\gamma^+} M' \) where \( \gamma^+ \) is the reflexive transitive closure of the silent transitions \( \tau \xrightarrow{\ast} \tau \). Bisimulation is now defined as follows.

**Definition 14** A relation \( R^+ \) is a bisimulation iff \( M_1 R^+ M_2 \) implies:

1. Given \( M_1 \xrightarrow{\gamma^+} M'_1 \) there is \( M'_2 \) such that: \( M_2 \xrightarrow{\gamma^+} M'_2 \) and \( M'_1 R^+ M'_2 \)
2. Given \( M_2 \xrightarrow{\gamma^+} M'_2 \) there is \( M'_1 \) such that: \( M_1 \xrightarrow{\gamma^+} M'_1 \) and \( M'_1 R^+ M'_2 \)

We denote bisimilarity, the largest bisimulation, as \( \approx^+ \) and prove that it is a congruence.

**Theorem 4 (Congruence of the Bisimilarity)**

\[
M_1 \approx^+ M_2 \iff M_1 \approx^+ M_2
\]

The proof splits the thesis into two sub-lemma: completeness and soundness.

**Lemma 4 (Completeness)**

\[
M_1 \approx^+ M_2 \Rightarrow M_1 \approx^+ M_2
\]

To prove that contextual equivalence implies bisimilarity we show that the contextual equivalence relation is itself a bisimulation. Assume that: \( M_1 \approx^+ M_2 \). and that: \( M_1 \xrightarrow{\gamma^+} M'_1 \) (or its symmetry). We must show that there exists a \( M_2 \) such that \( M_2 \xrightarrow{\gamma^+} M'_2 \) and that \( M'_1 \approx^+ M'_2 \). The proof proceeds by case analysis on the labels \( \gamma^+ \). The labels are divided into two camps: those produced by \( M \) and those produced by the attacker \( A \). In the former case we prove the contra positive: if \( M_1 \xrightarrow{\gamma^+} M'_1 \land M_2 \xrightarrow{\gamma^+} M'_2 \Rightarrow M_1 \neq^+ M_2 \), by showing that for every scenario where the states produce different labels there exists a context \( C \) that can distinguish \( M_1 \) and \( M_2 \). In the latter case we simply show that every label produced by the attacker can be encoded as a context \( C \),
because contextual equivalence is closed under contexts that suffices to imply the thesis.

**Lemma 5 (Soundness)**

\[ M_1 \approx^+ M_2 \Rightarrow M_1 \simeq^+ M_2 \]

The proof proceeds by induction on the number of reduction steps. We show that given \( P_1 = A \parallel M_1 \) and \( P_2 = A \parallel M_2 \) that \( P_2 \) diverges if and only if \( P_1 \) diverges.

Full proofs of both lemmas are provided in Appendix B.3 and Appendix B.4 respectively.

### 3.5.3 A Fully Abstract Translation to MiniML⁺

To prove that the FFI is secure we prove that there exists a fully abstract translation between MiniML and MiniML⁺. We define the translation scheme that injects a MiniML term \( e \) into a secure state of MiniML⁺ as follows.

\[
\{ \mu \mid e \}^+_0 \overset{\text{def}}{=} \star; \mu \vdash e \circ e \quad \text{where } \Gamma \vdash e : \tau
\]

This translation scheme will preserve the abstractions of \( e \) irrespective of which attacker \( A \) it faces. Formally stated: contextually equivalent terms in MiniML remain contextually equivalent when injected to MiniML⁺.

**Theorem 5 (A Secure FFI)**

\[ \mu_1 \mid e_1 \simeq^M \mu_2 \mid e_2 \iff \{ \mu_1 \mid e_1 \}^+_0 \simeq^+ \{ \mu_2 \mid e_2 \}^+_0 \]

The proof splits the thesis into two sub-lemma: preservation and reflection. Instead of directly addressing contextual equivalence, we make use of the bisimilarities \( \approx^M \) and \( \approx^+ \) that we previously proved to be adequate replacements for the contextual equivalence relations (Sections 3.5.1,3.5.2).

**Lemma 6 (Preservation)**

\[ \theta; \mu_1 \mid e_1 : \tau \approx^M \theta; \mu_2 \mid e_2 : \tau \Rightarrow \{ \mu_1 \mid e_1 \}^+_0 \approx^+ \{ e_2 \}^+_0 \]

**Proof Sketch** We must establish that there exists a relation \( \mathcal{R} \), so that:

1. \( \{ e_1 \}^+_0 \mathcal{R} \{ e_2 \}^+_0 \)

2. \( \mathcal{R} \) relates \( M_1 \) and \( M_2 \) as would \( \mathcal{R}^+ \).

We define \( \mathcal{R} \) as

\[ \mathcal{R} = \mathcal{R}_0 \cup \mathcal{R}_1 \cup \mathcal{R}_2 \cup \mathcal{R}_3 \]

one relation for each possible kind of the state \( M \). The relation \( \mathcal{R}_0 \) relates two waiting states: \( N; \mu \vdash \Sigma \) and \( N'; \mu' \vdash \Sigma' \) and enforces that the name maps \( N \) and \( N' \) are equivalent:
\[ \text{Dom}(N) = \text{Dom}(N') \land \forall n_i. N(n_i) \simeq N'(n_i), \]

and that the evaluation stacks are equivalent:

\[ |\Sigma| = |\Sigma'| \land \forall E, E', t. E[e] \simeq E'[e]. \]

The relation \( \mathcal{R}_1 \) relates two states reducing terms \( e_1 \) and \( e_2 \) requiring that \( e_1 \simeq^M e_2 \) in addition to \( \mathcal{R}_0 \). The relations \( \mathcal{R}_2 \) and \( \mathcal{R}_3 \) relate the marshalling states, they require that \( \mathcal{R}_0 \) holds and that the marshalled terms are equal if they are terms of MiniML\(^a\) and contextually equivalent otherwise.

Case (1) now follows from the assumption. Case (2) proceeds by analysis on the label \( \gamma^+ \).

**Lemma 7 (Reflection)**

\[ \{ \mu_1 \mid e_1 \} + \approx^+ \{ \mu_2 \mid e_2 \} + \Rightarrow \emptyset; \mu_1 \mid e_1 \simeq^M \emptyset; \mu_2 \mid e_2 \]

**Proof Sketch** We prove the lemma by the contrapositive:

\[ \emptyset; \mu_1 \mid e_1 \not\approx^M \emptyset; \mu_2 \mid e_2 \Rightarrow \{ \mu_1 \mid e_1 \} + \not\approx^+ \{ \mu_2 \mid e_2 \} + \]

The proof has two cases. In the first case the bisimulation fails immediately as \( e_1 \) and \( e_2 \) produce different transitions after silent reduction:

\[ \emptyset; \emptyset \mid e_1 : \tau_1 \Rightarrow^\gamma \zeta_1 \land \not\exists \zeta_2. \emptyset \mid e_2 \tau_2 \Rightarrow^\gamma \zeta_2 \]

(or its symmetry). In this case we derive the thesis by case analysis over the labels \( \gamma \).

In the second case there is a sequence of context actions (@\( v, \text{ref}_v, !l_i, l_i := v \)) that result in two states where different LTS transitions apply. In this case we establish the thesis by showing that each MiniML context action can be replicated by an MiniML\(^+\) attacker action.

Full proofs for both lemmas are provided in Appendix B.5 and Appendix B.6 respectively.

### 3.6 Summary

This chapter started by developing a high-level attacker model MiniML\(^a\) for a low-level attacker, whose accuracy we prove in Chapter 4. Next we addressed the limitations of Matthews and Findler multi-language semantics [55] when used to model a secure foreign function interface that leverages PMA. We then introduced MiniML\(^+\): a formal model of a secure foreign function interface between MiniML and the attacker MiniML\(^a\). This formal model accurately lifts the low-level memory protection techniques from the PMA mechanism into the resulting multi-language system. The FFI is secure in that it preserves
the abstractions of MiniML in its interactions with the low-level attacker. This
security property was proven by establishing that there exists a fully abstract
translation scheme from contextually equivalent terms in MiniML to context-
ually equivalent secure states of the FFI. Because contextual equivalence is
a challenging formalism for proofs, we proved the existence of this fully ab-
stract translation scheme by means of bisimilarity relations over MiniML and
MiniML\(^{+}\) that coincide with contextual equivalence.
4. A Reflection Based Model of a Low-Level Attacker

To ensure a simpler and more light-weight formalisation, our secure foreign function interface formalisation MiniML$^+$ of Chapter 3 used a high-level attacker MiniML$^a$ derived directly from the source language MiniML. This high-level attacker model serves as a model of the threats posed by our actual low-level attacker A$I$, an assembly language attacker with its code injection capabilities restricted by the protected memory of PMA (Section 2.7), to the contextual equivalences of MiniML programs residing in the protected memory of PMA.

Accurately modeling the impact malicious low-level code has on high-level functional programs is, however, rather challenging as the syntax and semantics of low-level code differs greatly from that of high-level functional programming languages. The validity of high-level models such as the MiniML$^a$ model of Section 3.2 can be called into question.

In this chapter we prove that the high-level attacker model MiniML$^a$ is an accurate model of the A$I$ attacker, by introducing a bisimilarity relation that captures the actions and observations of the A$I$ attacker as it interacts with MiniML programs residing in the protected memory of PMA. Our main contribution is an efficient technique for deriving this bisimilarity relation, by applying the labels of the trace semantics over A$I$ to our bisimilarity relation $\approx^+$ over the MiniML$^a$ attacker model.

Our approach is to replace the high-level interactions between the MiniML$^a$ attacker and the secure MiniML program, with low-level calls, returns, write outs and read outs between the low-level attacker and the protected memory where the MiniML program resides. The following, for example, is a function application of a shared MiniML function by the MiniML$^a$ attacker

$$N, n_1^f [\Rightarrow (e, \tau \rightarrow \tau'); \mu \models \Sigma \Rightarrow n_1^f]$$

$$\langle (N, n_1^f \Rightarrow (e, \tau \rightarrow \tau); \mu \models \Sigma, (e \cdot) : \tau \rightarrow \tau' \Rightarrow \rangle$$

where $n_1^f$ is the name that masks the desired function and $v$ the argument of the MiniML$^a$ attacker. This high-level interaction will become a call to a low-level address with byte word representations of the arguments as follows.

$$\langle (N, n_1^f \Rightarrow (e, \tau \rightarrow \tau); \mu \models \Sigma, s, \alpha) \Rightarrow call_{a_r}(w_r, w_d, a_d) \rangle$$

$$\langle (N, n_1^f \Rightarrow (e, \tau \rightarrow \tau); \mu \models \Sigma, (e \cdot) : \tau \rightarrow \tau' \Rightarrow \rangle, s, a_r : a_d : \alpha \rangle$$
At the low-level, the actions of the high-level attacker will correspond to calls to the entry points of the protected memory, such as the address $a_e$ in this example. Additionally, the values of the MiniML$^a$ attacker become low-level words $w$ of the low-level attacker, in this example, $w_n$ is the byte code representation of the name $n^f_1$ and $\bar{w}$ the byte code representation of the argument. The low-level attacker also inputs and manipulates low-level technical details such as return addresses and heap pointers, captured in our example label as $a_r$ and $a_d$. To deal with these additional low-level details the state of the previous bisimilarity $\approx^+$ over the high-level attacker is expanded with a memory descriptor $s$ and a stack of addresses $\bar{a}$.

To prove that despite all its additional low-level capabilities, the low-level attacker is no more powerful than our reflection based attacker model, we prove that there exists a simple fully abstract translation from MiniML$^+$, our model that features the high-level attacker, to the labelled transition system that captures the powers of the low-level attacker.

The remainder of this chapter is organised as follows. Firstly the chapter summarises the ideas behind our high-level attacker model (Section 4.1). Next, this chapter introduces a bisimilarity relation over the assembly language attacker using the labels of fully abstract trace semantics over the A+I low-level attacker (Section 4.2). Finally this chapter presents a proof for a fully abstract translation between both bisimilarity relations (Section 4.3).

4.1 The High-Level Attacker Model

Our high-level attacker model aims to accurately model the threats posed by an assembly-language attacker to the confidentiality and integrity properties of a source language $\mathcal{L}$ that are captured by our preferred security formalism: contextual equivalence. To ensure that this attacker model can be formalised quickly and easily, we specify it as three simple transformations that one must apply to a source language $\mathcal{L}$ to derive a high-level, but accurate, attacker model $\mathcal{L}_a$, which captures all possible threats to the confidentiality and integrity properties captured by the contextual equivalences of the source language $\mathcal{L}$.

Transformation 1: Removal of Type Safety

Type safety avoids stuck states and constrains program behaviour to uphold the typing rules. Removing the typing rules of $\mathcal{L}$ for the attacker, ensures that the high-level attacker $\mathcal{L}_a$ has no such restrictions.

Transformation 2: Introduction of Reflection

An assembly language attacker is not constrained by the source level restrictions of other more high-level programming languages as it can inspect and
To replicate this observational power we apply an insight from Wand [85] and Mitchell [60], who discovered that the inclusion of reflection into a programming language renders the abstractions and associated source level restrictions meaningless, because a reflection operator can simply look into every term and inspect its contents.

**Transformation 3: Reduce Control Flow**

The assembly language attacker is in complete control of its execution. The assembly language attacker can thus apply reflection to any execution mechanisms of the original source language $\mathcal{L}$. The high-level attacker model $\mathcal{L}_a$, however, is derived from $\mathcal{L}$ and is thus susceptible to the same control flow/execution mechanisms as $\mathcal{L}$. For $\mathcal{L}_a$ to be an accurate model of the assembly language attacker these mechanisms must be relaxed or removed from the semantics of the source language $\mathcal{L}$.

**Limitations**

In all of our experimentation with applying the $\mathcal{L}_a$ attacker model to different source languages $\mathcal{L}$, we have encountered but one constraint. It is only possible to derive an accurate attacker model $\mathcal{L}_a$ from a source language $\mathcal{L}$ whose function calls are observable, as an assembly-language attacker can accurately observe function calls and their arguments. It is thus not possible to derive a $\mathcal{L}_a$ style attacker from a purely functional $\lambda$-calculus, for example, because, as illustrated in the first example of Section 2.3.2, function calls are not observable in the pure $\lambda$-calculus.

### 4.1.1 The High-Level Attacker Model MiniML$^a$ Revisited

In Section 3.2 we applied these three transformations to MiniML to obtain our MiniML$^a$ attacker model. In what follows, we provide additional details on how the formalism of MiniML$^a$ conforms to the general model for high-level attackers of Section 4.1.

**Transformation 1: Removal of Type Safety**

As required MiniML$^a$ removes the types $\tau$ of MiniML as well as the typing rules from the formalism. To capture all the additional error states that become possible without the typing rules a new term $\text{wr}$ is introduced that captures all error states, as given by the ($\ast$-Error) rules. While capturing the stuck states of the attacker is not required, doing so does significantly simplify the proofs over the attacker model without reducing the effectiveness of the attacker.

**Transformation 2: Introduce Reflection**

The most important feature of the $\mathcal{L}_a$ attacker model is the inclusion of a reflection operator, as it bypasses the abstractions and the associated source
level restrictions of a programming language [85, 60]. Reflection was added to MiniML\(^a\) by means of the syntactical equality testing operator modulo \(\alpha\)-equivalence \(\equiv\). This \(\equiv\) operator enables programs in MiniML\(^a\) to compare the syntax of any two expressions.

Note that \(\alpha\)-equivalence operator \(\equiv\) ignores the variable names for a reason. A more aggressive syntactical equivalence operator which does not ignore the variable names would be too strong a reflection operator for modelling the low-level attacker. Even in assembly code variable names are not observable to malicious code, as such they also shouldn’t be observable to the high-level attacker model.

Transformation 3: Reduce Control Flow
The source language MiniML enforces that the evaluation order of the expressions be from left to right using the evaluation contexts \(E\) as listed in Section 2.1. The \(\alpha\)-equivalence testing operator \(\equiv\) of MiniML\(^a\) works around this enforced control flow, by not reducing either of its sub-terms to values. The \(\alpha\)-equivalence compares its sub-terms immediately and can thus not be limited by the control flow rules of MiniML.

Attacks in MiniML\(^a\)
As illustrated in Section 3.2.1, MiniML\(^a\) captures all relevant threats to contextual equivalence by the assembly language attacker, as the addition of reflection in MiniML\(^a\) by means of the \(\alpha\)-equivalence operator reduces contextual equivalence to \(\alpha\)-equivalence [85].

4.2 A Bisimilarity Relation that Captures the Assembly Language Attacker
In this section we introduce a bisimilarity relation over the assembly language attacker that captures its interactions with a MiniML program residing in the protected memory of the PMA mechanism. To accurately capture the inputs and observations of the assembly language attacker we adopt the labels of the fully abstract trace semantics over the interactions between the attacker and the protected memory space (Section 2.7) and apply them to an extended version of the state \(\textbf{M}\) over which the previous bisimilarity relation \(\approx\), which captured the observations and actions of our high-level attacker model MiniML\(^a\), is defined.

First we define this new state, a low level extension of the secure state \(\textbf{M}\) of MiniML\(^+\) (Section 4.2.1). Next, we introduce an updated set of marshalling rules that transition between MiniML values and the low level words of the assembly language attacker (Section 4.2.2). Following that we define the actions and observations of the low-level attacker (Section 4.2.3). Finally, we detail
the bisimilarity relation $\approx^l$ that captures the low-level attacker (Section 4.2.4). Later on in Section 4.3, we relate this bisimilarity relation $\approx^l$ that captures the actions and observations of the low-level attacker to the bisimilarity relation $\approx^+\text{a}$ that captures the actions and observations of the MiniML$^a$ attacker, to prove the accuracy of the MiniML$^a$ attacker.

### 4.2.1 The Low-Level State

The state of the labelled transition system over which our new bisimilarity relation $\approx^l$ is defined, is the following triple.

$$(M, s, \bar{a})$$

This triple is made up of $M$: the secure state of MiniML$^+$ (Section 3.4.3), $s$: a descriptor of the layout of the protected and unprotected memory (Section 2.7) and $\bar{a}$: a stack of addresses $a$.

The MiniML state $M$ captures the current state of the MiniML program interacting with the attacker from within the protected memory. This state replaces many of the low-level details of the protected memory denoted in the state $\Lambda$ (Section 2.7) such as, for example the program counter, that are not relevant to our bisimilarity relation.

The descriptor $s$ is used throughout the transition system to distinguish between addresses that reside in the code section of the attacker and the data section of the attacker, and is also employed to identify the addresses of the entry points to the protected memory. The following predicates, for example, use $s$ to determine whether or not a memory address $p$ is located within the code or data sections of the unprotected memory based on the information contained in $s$.

$$s \vdash \text{unprotectedCode}(p) \quad s \vdash \text{unprotectedData}(p)$$

These predicates enable us to prevent function calls from the MiniML program to the attackers data section and prevent write outs and read outs from the MiniML programs to the attackers code section.

The stack of addresses $\bar{a}$ enables the MiniML program to return its computed values to an address of the attacker as well as to write out a tuple to the data section of the attacker if a tuple is computed by MiniML. As a rule we have that the head of the stack $\bar{a}$ is the return address and the next address is the write out address in the data section.

### 4.2.2 Low-Level Marshalling

The marshalling rules of MiniML$^+$ (Section 3.4.3) convert MiniML values $v$ to MiniML$^a$ values $\text{v}$. The assembly language attacker, however, inputs and outputs words of bits $w$ instead of the high-level values $v$. To adjust to these
low-level words the marshalling rules of MiniML $^+$ over the MiniML state $M$
are adapted to convert to and from low-level words $w$.

The marshalling terms $m$ are thus updated from MiniML $^+$ to replace the
attacker values $v$ with the low-level words $w$ as follows

$$m ::= v | w | \langle m_i^{\in 1..n} \rangle | \overline{w} | wr$$

where $\langle m_i^{\in 1..n} \rangle$ captures the intermediate states where a tuple of length $n$ is
being marshalled in or out and where $\overline{w}$ denotes the result of marshalling out a
tuple: a sequence of words $w$. In what follows we detail the new marshalling
out and marshalling in rules.

**Marshalling Out**

Whenever a MiniML program reduces to a value $v$ that value must be con-
verted to one or more low-level words $w$. To save space in the following
marshalling rules, we compress the marshalling out state

$$\langle N; \mu \vdash \Sigma \triangleright m : \tau, s, a \rangle$$

into the following wrapper:

$$\downarrow \llbracket m \rrbracket^N$$

this wrapper denotes the only construct relevant to the marshalling process:
the map of shared names $N$. The new marshalling out rules are as follows.

\begin{align*}
(Marshall-Out-True-Low) & \quad \downarrow \llbracket \text{true} \rrbracket^N \rightarrow \downarrow \llbracket 0x1 \rrbracket^N \\
(Marshall-Out-False-Low) & \quad \downarrow \llbracket \text{false} \rrbracket^N \rightarrow \downarrow \llbracket 0x0 \rrbracket^N \\
(Marshall-Out-Number-Low) & \quad \downarrow \llbracket \tau_i \rrbracket^N \rightarrow \downarrow \llbracket \text{hex}\{n\} \rrbracket^N \\
(Marshall-Out-Foreign-Low) & \quad \tau = \tau_1 \rightarrow \tau_2 \quad \downarrow \llbracket \langle F(a_f, a_d) \rangle \rrbracket^N \rightarrow \downarrow \llbracket a_f \rrbracket^N \\
(Marshall-Out-Tuple-Low) & \quad \forall i \in 1..n. \quad \downarrow \llbracket v_i \rrbracket^N \rightarrow \downarrow \llbracket \overline{w_i} \rrbracket^N \\
(Marshall-Out-Lambda-Low) & \quad j = |N| + 1 \quad N' = (N, n_f^f \mapsto (\langle \lambda x : \tau.t, \tau \rightarrow \tau' \rangle) \\
(Marshall-Out-Location-Low) & \quad \downarrow \llbracket \langle l_i \rangle \rrbracket^N \rightarrow \downarrow \llbracket \text{hex}\{i\} \rrbracket^{N'}
\end{align*}
Throughout the marshalling rules we denote the conversion of numbers to hex as $\text{hex}\{\cdot\}$. In the (Marshall-Out-Unit-Low) rule, the MiniML value $\text{unit}$ is converted to the hex value 0xDEAD, this hex value was chosen arbitrarily. Because the expected type is stored within the secure state $M$ when marshalling out and in there is no risk of the number 0XDEAD being mistaken for a value $\text{unit}$ by the marshalling in rules.

In the (Marshall-Out-Foreign-Low) rule, an address $a_f$ to an attacker function/method is returned to the attacker. Whereas in MiniML $^+$ the foreign function was a foreign lambda term:

$$^\tau F(\lambda x.e)$$

when interoperating with the low-level attacker the foreign function becomes an address of the attacker $a_f$ with an address $a_d$ for the complex data structures shared with that the functionality at that address:

$$^\tau F(a_f,a_d).$$

In the Marshall-Out-Tuple-Low rule, a MiniML tuple is marshalled into a sequence of word sequences $\bar{w}$, as a tuple may contain other tuples within it.

Note that we still make use of reference objects for securely sharing locations and $\lambda$-terms (Marshall-Out-Location-Low, Marshall-Out-Lambda-Low). However, they are now simply shared as the hexadecimal representation of their respective indices.

**Marshall In**

Whenever the low-level attacker provides the MiniML program with a low-level input that input must be converted to an MiniML value $v$. To save space in the following marshalling rules, we compress the marshalling in state

$$\langle N; \mu \models \Sigma < m : \tau, s, \bar{a} \rangle$$

into the following wrapper:

$$\downarrow_s \mathcal{N} \downarrow^\tau$$

this wrapper denotes the three constructs relevant to the marshalling in process: the expected type $\tau$, the map of shared names $N$ and the memory descriptor $s$. The new marshalling in rules are as follows.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Marshall-In-True-Low)</td>
<td>(Marshall-In-False-Low)</td>
</tr>
<tr>
<td>$\downarrow_s \mathcal{N} \downarrow 0x1 \downarrow^N \mathcal{B}ool$</td>
<td>$\downarrow_s \mathcal{N} \downarrow 0x0 \downarrow^N \mathcal{B}ool$</td>
</tr>
<tr>
<td>(Marshall-In-Bool-Low-Error)</td>
<td>(Marshall-In-Number-Low)</td>
</tr>
<tr>
<td>$w \neq 0x1 \text{ or } w \neq 0x0$</td>
<td>$\downarrow_s \mathcal{N} \downarrow \text{hex}{n} \downarrow^N \mathcal{I}nt$</td>
</tr>
<tr>
<td>$\downarrow_s \mathcal{N} \downarrow w \downarrow^N \mathcal{B}ool$</td>
<td>$\downarrow_s \mathcal{N} \downarrow \bar{w} \downarrow^N \mathcal{B}ool$</td>
</tr>
</tbody>
</table>

85
(Marshall-In-Unit-Low-Low)
\[ \vdash N_{\text{Unit}}^{s} \rightarrow \vdash N_{\text{Unit}}^{s} \]

(Marshall-In-Location-Low)
\[ \vdash \text{hex}(j) \rightarrow \vdash N_{\text{Ref}}^{s} \]

(Marshall-In-Lambda-Low)
\[ \vdash N_{(\lambda x : \tau.t), \tau \rightarrow \tau'}^{s} \]

(Marshall-In-Foreign-Low)
\[ s \vdash \text{unprotectedCode}\{a_f\} \quad s \vdash \text{unprotectedData}\{a_d\} \]
\[ \vdash N_{\tau \rightarrow \tau'}^{s} F(a_f,a_d) \rightarrow \vdash N_{\tau \rightarrow \tau'}^{s} \]

(Marshall-In-Foreign-Low-Error)
\[ s \not\vdash \text{unprotectedCode}\{a_f\} \lor s \not\vdash \text{unprotectedData}\{a_d\} \]
\[ \vdash N_{\tau \rightarrow \tau'}^{s} \rightarrow \vdash \text{wr}^{N}_{\tau \rightarrow \tau'}^{s} \]

(Marshall-In-Tuple-Low)
\[ \forall i \in 1..n. w_i = (\text{read } a_d + (i-1)) \]
\[ \vdash N_{\tau_{i \in 1..n}} \rightarrow \vdash \text{wr}^{N}_{\tau_{i \in 1..n}}^{s} \]

(Marshall-In-Tuple-Low-Error)
\[ \exists i \in 1..n. w_i = (\text{read } a_d + (i-1)) \]
\[ \vdash N_{\tau_{i \in 1..n}} \rightarrow \vdash \text{wr}^{N}_{\tau_{i \in 1..n}}^{s} \]

(Marshall-In-Tuple-Low-Error2)
\[ s \not\vdash \text{unprotectedData}\{a_d\} \lor s \not\vdash \text{unprotectedData}\{a_d + (n - 1)\} \]
\[ \vdash N_{\tau_{i \in 1..n}} \rightarrow \vdash \text{wr}^{N}_{\tau_{i \in 1..n}}^{s} \]

Unlike the marshalling out rules, the marshalling in rules feature error rules (*-Error) that capture those scenarios where the input of the attacker doesn’t conform to the expected types or memory locations.

When marshalling in foreign functions (Marshall-In-Foreign-*) the input must be two words, where the first word is meant as an address to an attacker function, and the second word the data address where complex data structures must be written to. The address to the function \((a_f)\) must be located in the code section of the low-level attacker (Section 2.7), and the address to the data
section must be located in the data section of the low-level attacker. Otherwise, a call back from the MiniML program may produce a run-time error and the read or write outs of the MiniML programs may break the security properties of MiniML. The marshalling rules will thus terminate into the error state \( \text{wr} \).

When marshalling in foreign tuples (\( \text{Marshall-In-Tuple-*} \)) the input word serves as the start address for where the tuple is located in attacker memory. The values that make up the tuple are then read from the unprotected memory in sequence from 1 to \( n \), where \( n \) is determined by the expected tuple type \( \langle \tau_1^{[1..n]} \rangle \). Note that tuples within tuples are supported. Whenever the attacker wants to input a tuple that contains other tuples, she must place the inner tuples at different places within the unprotected memory and store their respective start addresses in the correct locations of the memory sequence that describes the outer tuple.

### 4.2.3 The Actions and Observations of the Low-Level Attacker

We formalise the actions and observations of the low-level attacker in a Labelled Transition System (LTS) defined by the following triple

\[
(\langle M, s, \overline{a} \rangle, L, \xrightarrow{L})
\]

where \( L \) are the labels of the fully abstract trace semantics for the low-level attacker (Section 2.7):

\[
\begin{align*}
L ::= \alpha \mid \tau \\
\alpha ::= \sqrt{} \mid \delta \mid \gamma \mid \gamma \mid \gamma \mid \gamma \mid \gamma \\
\gamma ::= \text{call } p(r, f) \mid \text{ret } p(r, f) \\
\delta ::= \gamma \mid \omega(a, v) \delta \\
\omega ::= \text{read} \mid \text{write}
\end{align*}
\]

and \( \xrightarrow{L} \) denotes the labelled transitions between the states as follows.

\[
\langle M, s, \overline{a} \rangle \xrightarrow{L} \langle M', s, \overline{a}' \rangle.
\]

Note that the descriptor \( s \) never changes during these transitions as the memory layout cannot be modified by any of the actions and observations of the low-level attacker.

The labels \( L \) denote only low-level calls, returns, reads and writes. The MiniML programs that reside in the protected memory of PMA, however, feature more functionality than those four simple primitives. To enable the low-level attacker to have the same kinds of interactions with the MiniML program as our high-level attacker model does, each type of MiniML operation: function application, reference allocation, and so on, is given its own entry point. The addresses \( a \) of these entry points are stored in the memory descriptor \( s \), which will use that information to identify the intent of a call by the low-level attacker.

In what follows we have split up the actions and observations of the low-level attacker into: setup, secure internal computations, returns from the secure, write outs, read outs, attacker returns, attacker calls and calls from the secure.
Setup

At the low-level control starts in the unprotected memory. The MiniML program residing in the protected memory is thus halted until it receives a signal from the attacker within the unprotected memory. This signal is defined by a start entry point to the protected memory. To start the interoperation with the MiniML program residing in the protected memory, the low-level attacker calls this start entry point.

\[
\langle (*; \emptyset \vdash \Sigma \circ e : \tau), s, \emptyset \rangle \xrightarrow{\text{call } a(a_r, a_d)?} \langle (*; \emptyset \vdash \Sigma \circ e : \tau), s, a_r : a_d : \emptyset \rangle
\]

where \( s \vdash \text{startEntryPoint}(a) \) (A-Start)

In this rule A-Start the descriptor \( s \) is used to identify that the address called by the attacker is the address of the start entry point through the predicate:

\( s \vdash \text{startEntryPoint}(a) \).

The only arguments the low-level attacker must provide to this start entry point is a return address \( a_r \), where it expects control to be returned to once the secure state is done computing its request, and an address \( a_d \) for complex data structures to be written to, in case the computation started must share a tuple with the low-level attacker. Both addresses \( a_r \) and \( a_d \) are added onto the empty stack of the state of the LTS, with \( a_r \) the new head of the stack.

Secure Internal Computations

The start entry point should only be called once, at the start of the interaction between the low-level attacker and the MiniML program. To that end, all secure internal computation, whether it be reducing a MiniML expression \( e \), plugging the stack or marshalling in or out require that the stack of addresses not be empty. All these transitions within the secure \( \mathbf{M} \) state, do not involve any interaction with the low-level attacker, and are thus labelled as silent using the label \( \tau \).

\[
\langle (N; \mu \vdash \Sigma \circ v : \tau), s, a_r : a_d : \emptyset \rangle \xrightarrow{\tau} \langle (N; \mu \vdash \Sigma \triangleright v : \tau), s, a_r : a_d : \emptyset \rangle
\]

(S-Setup)

\[
\langle (N; \mu \vdash \Sigma, E \sqsubset v : \tau), s, a_r : a_d : \emptyset \rangle \xrightarrow{\tau} \langle (N; \mu \vdash \Sigma \circ [v] : \tau), s, a_r : a_d : \emptyset \rangle
\]

(S-Plug)

\[
\frac{\mu \mid e \rightarrow \mu' \mid e'}{\langle (N; \mu \vdash \Sigma \circ e : \tau), s, a_r : a_d : \emptyset \rangle \xrightarrow{\tau} \langle (N; \mu' \vdash \Sigma \circ e' : \tau), s, a_r : a_d : \emptyset \rangle}
\]

(S-Inner)
Note that the LTS incorporates the previously defined marshalling rules from MiniML to low-level words (Section 4.2.2). The marshalling rules are generally labelled as silent ($\tau$), there is, however, an exception when the expected type is a tuple type $\langle \tau_i \in 1..n \rangle$. As addressed later on, marshalling in an expected tuple type produces observable low-level read traces.

When the marshalling in process fails, due to the input of the low-level attacker not conforming to the required type $\tau$, then the secure state is cleared:

$$\langle (N; \mu \models \Sigma \triangleright m : \tau), s, a_r : a_d : a \rangle \xrightarrow{\tau} \langle (N; \mu \models \Sigma \triangleright m' : \tau), s, a_r : a_d : a \rangle$$

(S-MarshOut)

$$\downarrow ||m||^N \rightarrow \downarrow ||m'||^N$$

(S-MarshIn)

$$\langle (N; \mu \models \Sigma \triangleleft m : \tau), s, a_r : a_d : a \rangle \xrightarrow{\tau} \langle (N'; \mu \models \Sigma \triangleleft m' : \tau), s, a_r : a_d : a \rangle$$

$\tau \neq \langle \tau_i \in 1..n \rangle$

and all interoperation is halted. Once the interoperation is halted, the attacker can only observe the fact that the interaction has been halted denoted as the label $\checkmark$ (Done).

Returns From the Protected Memory

Once the MiniML program in the protected memory has computed a value and marshalled it to a low-level representation, there are two possibilities depending on whether or not the value is a tuple or not. When the value $v$ is not a tuple and its low-level representation is thus a single word $w$, control will revert to the low-level attacker address $a_r$ with $w$ stored in one of the registers of the machine as long as the attacker address $a_r$ resides within the code section of the unprotected memory either revert control immediately to the head of the stack $a_r$ passing as its argument.

$$\langle (N; \mu \models \Sigma \triangleright w : \tau), s, a_r : a_d : a \rangle \xrightarrow{\text{ret } a_r \left( w \right)!} \langle (N; \mu \models \Sigma), s, a \rangle$$

(M-Ret)

where $s \vdash \text{unprotectedCode}(a_r)$ and $\tau \neq \langle \tau_i \in 1..n \rangle$

$$\langle (N; \mu \models \Sigma \triangleright \emptyset : \tau), s, a_r : a_d : a \rangle \xrightarrow{\checkmark} \langle (\ast; \emptyset \models \varepsilon), s, \emptyset \rangle$$

(Wr-MR)

Write Outs

When the MiniML program in the protected memory reduces to a tuple, the low-level representation of that tuple is not just a single low-level word $w$, but
a sequence of words \( \bar{w} \), whose number of elements are possibly larger than the amount of available registers. As such the sequence is written out to the unprotected data section of the attacker, provided there is enough space for it.

\[
(M\text{-RetT})
\]

\[
s \vdash unprotectedCode(a_r)
\]

\[
s \vdash unprotectedData(a_d) \quad s \vdash unprotectedData(a_d + |\bar{w}|)
\]

\[
\langle (\text{N}; \mu \models \Sigma ; \{ \tau_i^{1..n} \}), s, a_r : a_d : \bar{a} \rangle \xrightarrow{\text{write}(a_d, \bar{w}); \text{ret} a_r} \langle (\text{N}; \mu \models \Sigma ), s, \bar{a} \rangle
\]

\[
(Wr\text{-MRT})
\]

\[
s \not\vdash unprotectedCode(a_r) \lor s \not\vdash unprotectedData(a_d) \lor s \not\vdash unprotectedData(a_d + |\bar{w}|)
\]

\[
\langle (\text{N}; \mu \models \Sigma ; \{ \tau_i^{1..n} \}), s, a_r : a_d : \bar{a} \rangle \xrightarrow{\gamma} \langle (\ast; \emptyset \models \varepsilon ), s, \emptyset \rangle
\]

Denoting low-level write outs in a trace semantics, introduces many challenges to full abstraction [68]. In this work we circumvent these challenges by having the write outs always be a sequence of ordered write outs to incrementally increasing addresses, immediately followed by a return of control to the attacker. There is thus no overwriting of addresses within the same transaction, no reading of memory immediately after writing it and no termination after the write out.

**Read Outs**

When marshalling in a tuple input by the low-level attacker into the protected memory, the secure state will produce an observable trace of read outs, if all of the attacker specified data conforms to the typing rules.

\[
(\text{Read-Tuple})
\]

\[
\langle (\text{N}; \mu \models \Sigma ; \langle \tau_i^{1..n} \rangle), s, \bar{a} \rangle \rightarrow^* \langle (\text{N}; \mu \models \Sigma ; \langle \tau_i^{1..n} \rangle), s, \bar{a} \rangle
\]

\[
\langle (\text{N}; \mu \models \Sigma ; \langle \tau_i^{1..n} \rangle), s, \bar{a} \rangle \xrightarrow{\tau} \langle M'; s, \bar{a} \rangle \xrightarrow{\delta_!} \langle M'', s, \bar{a} \rangle
\]

\[
(\text{Read-Tuple-Incorrect})
\]

\[
\langle (\text{N}; \mu \models \Sigma ; \langle \tau_i^{1..n} \rangle), s, \bar{a} \rangle \rightarrow^* \langle (\text{N}; \mu \models \Sigma ; \text{wr} ; \langle \tau_i^{1..n} \rangle), s, \bar{a} \rangle
\]

\[
\langle (\text{N}; \mu \models \Sigma ; \langle \tau_i^{1..n} \rangle), s, \bar{a} \rangle \xrightarrow{\gamma} \langle (\ast; \emptyset \models \varepsilon ), s, \emptyset \rangle
\]

Like low-level writes, low-level reads introduce many challenges to full abstraction [68]. In this work we circumvent these challenges by having the reads always be a sequence of read ins from incrementally increasing addresses. There are thus no double reads of the same address within the same transaction. Every word read also directly impacts the behaviour of the secure state as depending on the data read in the type checks will or will not fail.
Attacker Returns
The low-level attacker, as mentioned explained in Section 2.6, cannot jump
to an address of the protected memory apart from the entry points, and must
thus return the values it computes to an entry point designated specifically for
returns from the attacker \((A-Ret)\).

\[
\langle (N; \mu \vdash \Sigma, E : \tau \rightarrow \tau'), s, \overline{a} \rangle \xrightarrow{\text{ret } a(\overline{w})?} \langle (N; \mu \vdash \Sigma, E : \tau \rightarrow \tau' \triangleleft \overline{w} : \tau), s, \overline{a} \rangle
\]

\(\text{where } s \vdash \text{returnback}(a)\) \hspace{1cm} (A-Ret)

Any returns to another addresses will lead to an end of the interoperation be-
tween the low-level attacker and the protected MiniML program.

\[
\langle (N; \mu \vdash \Sigma, E : \tau \rightarrow \tau'), s, \overline{a} \rangle \xrightarrow{\text{ret } a(\overline{w})?} \langle (\ast; \emptyset \vdash \varepsilon), s, \emptyset \rangle \quad (\text{Wr-AR})
\]

\(\text{where } s \not\vdash \text{returnback}(a)\)

Attacker Calls
The low-level attacker calls a separate entry point for each type of operation
on MiniML terms. These entry points take as an argument a byte word repre-
sentation of the indexed names that model the shared reference objects \((w_n)\) as
well as low-level words that represents the arguments.

\[
s \vdash \text{derefEntryPoint}(a) \quad w_n = \text{hex}\{i\} \quad N(\overline{n}_i) = (l_i, \text{Ref } \tau)
\]

\[
\langle (N; \mu \vdash \Sigma), s, \overline{a} \rangle \xrightarrow{\text{call } a(w_n,a_r,a_d)?} \langle (N; \mu \vdash \Sigma \circ l_i : \tau), s, a_r : a_d : \overline{a} \rangle
\]

\((A-Deref)\)

\[
s \vdash \text{applyEntryPoint}(a) \quad w_n = \text{hex}\{i\} \quad N(\overline{n}_i) = (e, \tau \rightarrow \tau')
\]

\[
E_1 = ( \langle e \rangle : \tau \rightarrow \tau' \quad \overline{a'} = a_r : a_d : \overline{a} \rangle
\]

\[
\langle (N; \mu \vdash \Sigma), s, \overline{a} \rangle \xrightarrow{\text{call } a(w_n,a_r,a_d)?} \langle (N; \mu \vdash \Sigma, E_1 \triangleleft \overline{w} : \tau), s, \overline{a'} \rangle
\]

\((A-Apply)\)

\[
s \vdash \text{setEntryPoint}(a) \quad w_n = \text{hex}\{i\} \quad N(\overline{n}_i) = (l_i, \text{Ref } \tau)
\]

\[
E_1 = ( l_i := \langle \rangle ) : \tau \rightarrow \text{Unit} \quad \overline{a'} = a_r : a_d : \overline{a} \quad
\]

\[
\langle (N; \mu \vdash \Sigma), s, \overline{a} \rangle \xrightarrow{\text{call } a(w_n,a_r,a_d)?} \langle (N; \mu \vdash \Sigma, E_1 \triangleleft \overline{w} : \tau), s, \overline{a'} \rangle
\]

\((A-Set)\)

\[
s \vdash \text{allocEntryPoint}(a) \quad \text{convt}(w_i) = \tau
\]

\[
E_1 = ( \langle \text{ref } \rangle : \tau \rightarrow \text{Ref } \tau \quad \overline{a'} = a_r : a_d : \overline{a} \rangle
\]

\[
\langle (N; \mu \vdash \Sigma), s, \overline{a} \rangle \xrightarrow{\text{call } a(w_n,a_r,a_d)?} \langle (N; \mu \vdash \Sigma, E_1 \triangleleft \overline{w} : \tau), s, \overline{a'} \rangle
\]

\((A-Ref)\)

The entry points handle function calls \((A-Apply)\), the setting and dereferencing
of shared location \((A-Set,A-Deref)\) as well as the allocation of new references
\((A-Ref)\). In the case of the latter rule we make use of a conversion function
convt to convert a hexadecimal value into a type τ. Note that during each of these operations the expected type is always inferred at run-time, from the type information stored in Σ and N, to ensure that the operations happen in a type safe manner.

Whenever the assembly-language attacker makes a mistake by calling an inaccessible address of the protected memory, the secure state is cleared and all interoperation is halted.

\[
\langle (N; \mu \vdash \Sigma), s, \overline{a} \rangle \xrightarrow{\text{call } a(\overline{w}, \overline{a}, a_d) ?} \langle (\ast; \emptyset \vdash \varepsilon), s, \emptyset \rangle \quad \text{(Wr-AC)}
\]

where \(s \not\vdash \text{derefEntryPoint}(a)\) and \(s \not\vdash \text{ApplyEntryPoint}(a)\)
and \(s \not\vdash \text{SetEntryPoint}(a)\) and \(s \not\vdash \text{AllocEntryPoint}(a)\)

**Calls From the Protected Memory**

While the attacker makes many different types of calls to the protected memory, the MiniML program residing in the protected memory only calls attacker functions \(a_f\) whenever it applies them to an MiniML value during an interal computation. As was the case for returning a value to the attacker, when calling an attacker function \(a_f\) the traces will differ depending on whether or not the type of the value applied to the attacker function is a tuple. When the value is not a tuple and can thus be marshalled into a single word \(w\), the secure code will simply call the address \(a_f\) with the word \(w\) stored in one of the registers.

\[
\begin{align*}
\langle (N; \mu \vdash \Sigma \circ E[(\overline{\tau_1 \rightarrow \tau_2} F(a_f, a_d) \nu)] : \tau), s, \overline{a} \rangle & \rightarrow \\
\langle (N; \mu \vdash \Sigma \circ E : \tau_2 \rightarrow \tau \triangleright \nu : \tau_1), s, \overline{a} \rangle & \rightarrow^* \\
\langle (N'; \mu \vdash \Sigma, E : \tau_2 \rightarrow \tau \triangleright w : \tau_1), s, \overline{a} \rangle & \rightarrow \\
\langle (N'; \mu \vdash \Sigma, E) & \rightarrow (N'; \mu \vdash \Sigma) \rangle \\
\langle (N; \mu \vdash \Sigma \circ E[(\overline{\tau_1 \rightarrow \tau_2} F(a_f, a_d) \nu)] : \tau), s, \overline{a} \rangle & \xrightarrow{\text{call } a_f(w)!} \langle M', s, \overline{a} \rangle
\end{align*}
\]

When the value is a tuple, the sequence of words that represent it must first be written out to the address \(a_d\) before calling the address \(a_f\). Note that the address \(a_f\) is not checked to be located in the secure code in any of these rules as this check was performed earlier by the marshalling in rule (Marshall-In-Foreign-Low). The rules (M-Call-Complex-\*) do however check that there is enough space for the to be written out tuple as this can only be checked once the size of the marshalled out tuple is known.

\[
\begin{align*}
\langle (N; \mu \vdash \Sigma \circ E[(\overline{\tau_1 \rightarrow \tau_2} F(a_f, a_d) \nu)] : \tau), s, \overline{a} \rangle & \rightarrow \\
\langle (N; \mu \vdash \Sigma, E : \tau_2 \rightarrow \tau \triangleright \nu : \tau_1), s, \overline{a} \rangle & \rightarrow^* \\
\langle (N'; \mu \vdash \Sigma, E : \tau_2 \rightarrow \tau \triangleright \overline{w} : \tau_1), s, \overline{a} \rangle & \rightarrow \\
\langle (N'; \mu \vdash \Sigma, E) & \rightarrow (N'; \mu \vdash \Sigma) \rangle \\
\langle (N; \mu \vdash \Sigma \circ E[(\overline{\tau_1 \rightarrow \tau_2} F(a_f, a_d) \nu)] : \tau), s, \overline{a} \rangle & \xrightarrow{\text{write}(a_d, w). \text{call } a_f!} \langle S \rangle
\end{align*}
\]
(M-Call-Complex-Error)
\[ \langle (N; \mu \vdash \Sigma, E)[(\tau_1 \rightarrow \tau_2) \mathcal{F}(a_f, a_d) v] : \tau, s, \bar{a} \rangle \rightarrow \langle (N; \mu \vdash \Sigma, E) : \tau_2 \rightarrow \tau \triangleright (\tau_1 \rightarrow \tau_2) : \tau, s, \bar{a} \rangle \rightarrow^* \]
\[ s \not\vdash \text{unprotectedData}(a_d + |\bar{w}|) \]
\[ \langle (N; \mu \vdash \Sigma, E)[(\tau_1 \rightarrow \tau_2) \mathcal{F}(a_f, a_d) v] : \tau, s, \bar{a} \rangle \xrightarrow{\forall} \langle (\ast; \emptyset \vdash \varepsilon), s, \emptyset \rangle \]

4.2.4 A Bisimilarity Relation Over the Low-Level Attacker

As in Section 3.5.2 we define a weak bisimulation relation \( \mathcal{B}^l \) as per Definition 6. Define the transition relation
\[ \langle M, s, \bar{a} \rangle \xrightarrow{\alpha} \langle M', s', \bar{a}' \rangle \]
as \( \langle M, s, \bar{a} \rangle \xrightarrow{\tau} \xrightarrow{\alpha} \langle M', s', \bar{a}' \rangle \) where \( \tau \xrightarrow{\ast} \) is the reflexive transitive closure of the silent transitions \( \xrightarrow{\tau} \).

**Definition 15** \( \mathcal{B}^l \) is a bisimulation iff \( \langle M_1, s_1, \bar{a}_1 \rangle \mathcal{B}^l \langle M_2, s_2, \bar{a}_2 \rangle \) implies:

1. Given \( \langle M_1, s_1, \bar{a}_1 \rangle \xrightarrow{\alpha} \langle M'_1, s_1, \bar{a}_1' \rangle \), there is \( \langle M'_2, s_2, \bar{a}_2' \rangle \) such that \( \langle M_2, s_2, \bar{a}_2 \rangle \xrightarrow{\alpha} \langle M'_2, s_2, \bar{a}_2' \rangle \) and \( \langle M'_1, s_1, \bar{a}_1' \rangle \mathcal{B}^l \langle M'_2, s_2, \bar{a}_2' \rangle \)

2. Given \( \langle M_2, s_2, \bar{a}_2 \rangle \xrightarrow{\alpha} \langle M'_2, s_2, \bar{a}_2' \rangle \), there is \( \langle M'_1, s_1, \bar{a}_1' \rangle \) such that \( \langle M_1, s_1, \bar{a}_1 \rangle \xrightarrow{\alpha} \langle M'_1, s_1, \bar{a}_1' \rangle \) and \( \langle M'_1, s_1, \bar{a}_1' \rangle \mathcal{B}^l \langle M'_2, s_2, \bar{a}_2' \rangle \)

We denote bisimilarity, the largest bisimulation as, \( \approx^l \).

4.3 A Fully Abstract Translation Between the High-Level and Low-Level Attacker

To prove the effectiveness of our high-level attacker MiniML\(^a\), we prove that there is no sequence of observations and actions by the low-level attacker that alters MiniML programs, residing in the protected memory of PMA, in ways that cannot be replicated by the high-level attacker. We prove this by proving that states that are bisimilar in \( \approx^+ \), the bisimilarity relation that captures the actions and observations of the high-level attacker, are bisimilar in \( \approx^l \), the bisimilarity relation over the assembly-language attacker, given a simple translation scheme between the states.

**Theorem 6 (Fully Abstract Translation)** Given a low-level memory descriptor \( s \) that describes the protected memory space of PMA and the entry points, we have that:
\[ \{\mu_1 \mid e_1\} \approx^+ \{\mu_1 \mid e_1\} \iff \langle \{\mu_1 \mid e_1\}, s, \emptyset \rangle \approx^l \langle \{\mu_2 \mid e_2\}, s, \emptyset \rangle \]
where \( {} + \) is the translation scheme from MiniML programs to states \( M \) of MiniML\(^ + \) of Section 3.5.3.

The proof of this theorem is similar to the proof of Theorem 5. Whereas in the proof of the latter, we showed that the observations and actions of the MiniML\(^ a \) attacker coincided with those of MiniML contexts, the proof of Theorem 6 shows that the observations and actions of the low-level attacker coincide with those of the MiniML\(^ a \) attacker using similar proof techniques.

Again the proof is split into two sub-lemma: preservation and reflection.

**Lemma 8 (Preservation)**

\[
\{ \mu_1 \mid e_1 \} + \approx \{ \mu_1 \mid e_1 \} + \Rightarrow \langle \{ \mu_1 \mid e_1 \} + , s, \emptyset \rangle \approx \langle \{ \mu_2 \mid e_2 \} + , s, \emptyset \rangle
\]

**Proof Sketch** We must establish that there exists a relation \( \mathcal{R} \), such that:

1. \( \langle \{ \mu_1 \mid e_1 \} + , s, \emptyset \rangle R \langle \{ \mu_2 \mid e_2 \} + , s, \emptyset \rangle \)
2. \( \mathcal{R} \) relates low-level states \( \langle M, s, a \rangle \) and \( \langle M', s', a' \rangle \) as would \( \approx l \).

We define \( \mathcal{R} \) as a union of four relations

\[
\mathcal{R} = \mathcal{R}_0 \cup \mathcal{R}_1 \cup \mathcal{R}_2 \cup \mathcal{R}_3
\]

one relation for each possible kind of low-level state. The relation \( \mathcal{R}_0 \) relates halted states: \( \langle (N; \mu \vdash \Sigma), s, a \rangle \) and \( \langle (N'; \mu' \vdash \Sigma'), s', a' \rangle \) and requires that the name maps are equivalent:

\[
\text{Dom}(N) = \text{Dom}(N') \land \forall n_i. N(n_i) \simeq N'(n_i)
\]

that the evaluation stacks are equivalent:

\[
|\Sigma| = |\Sigma'| \land \forall E, E', e. E[e] \simeq E'[e]
\]

that the memory layouts are the same \( s = s' \), and that the return address stacks are equal as well \( a = a' \). The relation \( \mathcal{R}_1 \) relates two states reducing terms contextually equivalent terms \( e \) and \( e' \) in addition to upholding \( \mathcal{R}_0 \). The relations \( \mathcal{R}_2 \) and \( \mathcal{R}_3 \) relate the marshalling states, they require that \( \mathcal{R}_0 \) holds and that the marshalled terms are equal if they are assembly language terms.

Case (1) now follows from the assumption. Case (2) proceeds by analysis on the low-level label \( L \).

**Lemma 9 (Reflection)**

\[
\{ \mu_1 \mid e_1 \} + \not\approx \{ \mu_2 \mid e_2 \} + \Rightarrow \langle \{ \mu_1 \mid e_1 \} + , s, \emptyset \rangle \not\approx \langle \{ \mu_2 \mid e_2 \} + , s, \emptyset \rangle
\]

**Proof Sketch** We prove the contrapositive:

\[
\{ \mu_1 \mid e_1 \} + \not\approx \{ \mu_2 \mid e_2 \} + \Rightarrow \langle \{ \mu_1 \mid e_1 \} + , s, \emptyset \rangle \not\approx \langle \{ \mu_2 \mid e_2 \} + , s, \emptyset \rangle
\]

The proof has two cases. In the first case the bisimulation fails immediately as the MiniML terms \( e_1 \) and \( e_2 \) embedded in \( M \) either reduce to different
values or diverge in one case. These unrelatable states are replicated directly in the low-level bisimilarity relation, the only difference being that it takes one additional LTS transition before the equivalent low-level states are unrelatable. This is due to a start transition with label: \textit{call} a(a_r,a_d) that starts the reduction of the embedded MiniML terms.

In the second case there is a sequence of high-level attacker actions that result in two states where different LTS transitions apply. In this case we establish the thesis by showing that each high-level attacker action can be replicated by an assembly-language attacker action.

Full proofs of both lemmas are available in Appendix C.1 and Appendix C.2 respectively.

4.4 Summary

We started this chapter by introducing the general high-level attacker model \( \mathcal{L}_a \) that captures the threats that an assembly-language attacker poses to the abstractions of programs written in a source language \( \mathcal{L} \) that reside within the memory space protected by PMA. The accuracy of this high-level attacker model was proven for MiniML\(^a\) our application of this high-level attacker in the formalisation of our secure foreign function interface MiniML\(^+\). We proved the accuracy of MiniML\(^a\) by relating the bisimilarity relation over MiniML\(^+\) to a bisimilarity relation over the low-level attacker. This bisimilarity relation over the low-level attacker was obtained by applying the labels of the fully abstract trace semantics over the low-level attacker to the labelled transition system over MiniML\(^+\).
5. A Secure Abstract Machine

This chapter presents the derivation and implementation of an abstract machine for MiniML (Section 2.1) that runs on a processor enhanced with the Protected Module Architecture (PMA) (Section 2.6). This abstract machine executes MiniML programs within this protected memory without exposing its implementation details in its interaction with the outside world.

Abstract machines are both theoretical models used to study language properties and practical models of language implementations. Nowadays, several languages, especially functional ones, are implemented using abstract machines. For example, Scheme-based languages run on SECD machines [23], OCaml’s bytecode is based on the Zinc abstract machine [47] and the Glasgow Haskell Compiler uses the Spineless Tagless G-machine [44] internally.

When security-sensitive applications are run on an abstract machine it is crucial that the abstract machine implementation does not leak security sensitive information. Outside of implementation mistakes, abstract machine implementations are also threatened by the loE-level context in which they reside. In practice, an abstract machine implementation will use or interact with various, loE-level libraries and/or components that may be written with malicious intent or susceptible to code injection attacks. Such malicious loE-level components can bypass software based security mechanisms and may disclose confidential data, break integrity checks and so forth (Section 2.7).

To guarantee the security of the implemented abstract machine for MiniML, meaning that it reflects and preserves the equivalences of MiniML (Section 3.1), we follow a four step methodology to derive the formalisation of the abstract machine. In the first step MiniML is extended with a foreign function interface that captures the interactions between the loE-level attacker and MiniML (Chapters 3 and 4). In the second step the formalisation is simplified by adopting an LTS model of the extended language. In the third step we apply the syntactic correspondences of Biernacka et al. [12] to a labelled transition system, which captures the interactions between MiniML and the attacker, to obtain the formalisation of a CESK machine implementation for MiniML. For each of these syntactic correspondences we prove that they do not result in the abstract machine leaking security sensitive information. In a final step the formalisation of the abstract machine is implemented.

After presenting the relevant security concepts for abstract machines (Section 5.1), this chapter makes the following contributions. It describes our methodology (Section 5.2) and how we apply it to derive the secure CESK
machine for MiniML (Section 5.3). This chapter also details our implementation of the CESK machine and lists the run-time performance of the abstract machine for certain test scenarios (Section 5.3.2).

Limitations
The proposed work is not without limitations. While the formalisation is derived in a correct manner, the implementation is hand-made: possibly introducing mistakes that violate the security properties. As with all software, testing and verification techniques can minimise these risks.

5.1 The LoE-Level Security Concerns of Abstract Machine Implementations
An abstract machine for a source language $L$, is a program that inputs programs written in $L$ or the byte code representation of $L$ and then executes them according to how the semantics of the source language $L$ are encoded into the abstract machine. As noted before, abstract machine implementations have two general security concerns:

1. Implementation mistakes.
2. Malicious behaviour by the loE-level context in which it resides.

In this work we consider the latter concern. More specifically, we consider the threats posed to an abstract machine implementation by our attacker with kernel-level code injection privileges (Section 2.7).

As in previous chapters we capture the security concerns by means of contextual equivalence (Section 2.3). Our loE-level attacker poses four distinct threats to abstract machine implementations, that contextual equivalence can capture.

Inspection and Manipulation
An abstract machine must isolate running programs and their machine state from any kind of inspection and manipulation from outside the abstract machine. A failure to do so breaks the confidentiality and integrity requirements of the programs that the abstract machine runs.

Abuse of References
An abstract machine interoperates with the outside context, including possibly the attacker, by sharing references to data structures. An attacker that accesses and modifies what these references refer to, can manipulate the internal state of the abstract machine and alter the control flow (as is the case for the Java Virtual Machine [82]) of the program it runs.
Observing Implementation Details
Most abstract machines do not encode the exact semantics of their source language, but instead use more concrete and more efficient encodings such as continuations and closures. Because these encodings make the abstractions of the source language more concrete they may reveal to the attacker implementation details not observable to contexts in the source language $\mathcal{L}$.

Violation of Type Safety
When interoperating with the outside context, the abstract machine not only shares data, it also receives it. If the abstract machine does not type check this incoming data, as commonly done for performance reasons, the attacker can violate the abstractions of the source language $\mathcal{L}$.

5.2 Methodology
This section introduces a general four-step methodology to derive secure abstract machines that run on processors enhanced with PMA. The goal of the methodology is to propagate the contextual equivalence of a source language $\mathcal{L}$ all the way down to implemented abstract machine.

Starting from the source language $\mathcal{L}$ the first step of the methodology derives an extended language $\mathcal{L}^+$ that enhances the original language $\mathcal{L}$ with a foreign function interface to the loE-level attacker (Section 5.2.1). The second step removes the explicit formalisation of the attacker model by converting the enriched language $\mathcal{L}^+$ into a labelled transition system representation (LTS) that captures the internal computations of $\mathcal{L}$ as well as the interactions between the attacker and programs in the source language (Section 5.2.2). The third step derives an abstract machine, referred to as $\text{AM}^+$, from this LTS through correspondences [12] (Section 5.2.3). In a final step the machine is implemented and compiled securely (Section 5.2.4).

Each step includes a property that needs to be verified to hold in order to show that the step was carried out correctly. Finally, this section validates the methodology by showing that the combination of these properties ensures that the abstract machines derived with the methodology are indeed secure (Section 5.2.5).

5.2.1 Step 1: Introduce a Secure Foreign Function Interface
The loE-level attacker threatens the security of an abstract machine through its interactions with the machine. These interactions are modelled in this step by creating a language $\mathcal{L}^+$ where the loE-level attacker and $\mathcal{L}$ programs interoperate. The $\mathcal{L}^+$ language must uphold the following requirements.
Separable Program State

The methodology strives to secure only source language programs within the protected memory of PMA. To that end the program state $P$ of the combined language $\mathcal{L}^+$ must be separable into a state of the attacker ($A$) and a state of the source language $\mathcal{L}$ ($S$). We denote this separable program state as follows.

$$P = A \parallel S$$

Note that a program state can differ from the code of that program, as for example languages that feature dynamic memory allocation have states that include a heap.

Isolation

To ensure that the state of a $\mathcal{L}$ program is correctly isolated from any kind of inspection by the attacker, the data structures and code of $\mathcal{L}$ programs should not be directly accessible to the attacker. Sharing data structures from a program in $\mathcal{L}$ with the attacker can thus not be done by sharing direct references to these data structures. Instead, these structures should be shared by either copying them to the attacker, by encrypting them or by providing the attacker with reference objects, which are objects that refer to data structures of a program in $\mathcal{L}$ (Section 3.4.1).

Property 1 (Correctness of the 1st step)

If the languages are combined correctly then there exists a translation $\{\cdot\}_{\mathcal{L}^+}$ that takes programs $P$ of $\mathcal{L}$ and outputs the secure states of programs in the extended language $\mathcal{L}^+$. This translation must be fully abstract. This property is formalised as follows:

$$P_1 \simeq P_2 \iff \forall A. A \parallel \{P_1\}_{\mathcal{L}^+} \simeq_{\mathcal{L}^+} A \parallel \{P_2\}_{\mathcal{L}^+}$$

5.2.2 Step 2: Label and Separate

To derive an abstract machine we start by developing a labelled translation system (LTS) over the source language. In our methodology, this LTS is obtained in two steps:

1. The reduction steps of the combined language $\mathcal{L}^+$ are annotated with labels $\alpha$ as per Laird [46], that consists of either observable interactions between the attacker and the source language ($\gamma$) or ($\tau$) to denote unobservable interactions.

2. The attacker’s explicit state is replaced with an abstract one, since its details are not available to the abstract machine development. Only the interaction with the attacker is of interest, and this is captured by the labels $\gamma$. 

99
The resulting LTS is a triple \((S,\alpha,\alpha \rightarrow)\) where source language states \(S\) are the states, \(\alpha\) the set of labels and \(S \rightarrow \alpha S'\) the labelled transitions between the states. To prove the security properties of the obtained LTS, the methodology makes use of a weak bisimilarity relation \(\approx\) over the LTS as detailed in Definition 6.

**Property 2 (Correctness of the 2nd step)**

This step is correct if the bisimilarity relation \(\approx^{L^+}\) over the LTS is a congruence, that is, it coincides with contextual equivalence in \(L^+ (\simeq^{L^+})\):

\[
S_1 \approx^{L^+} S_2 \iff \forall A. A \parallel S_1 \simeq^{L^+} A \parallel S_2
\]

5.2.3 Step 3: Derive an Abstract Machine

In the third step of the methodology, an abstract machine, referred to as \(AM^+\), is derived from the previously obtained LTS. This is done through a sequence of systematic transformations, often referred to as correspondences. Such correspondences take in a LTS and stepwise modify the state and/or the reduction rules of the LTS in a way that preserves the formal properties of the original LTS in the resulting abstract machine.

The derived abstract machines are defined through triples \((S^{AM},\alpha,\alpha \rightarrow)\) consisting of states \(S^{AM}\), labels \(\alpha\) and a transition relation of the form:

\[
S^{AM} \alpha \rightarrow S^{AM'}
\]

Here, the labels \(\alpha\) are the same labels \(\alpha\) as in the original LTS because a correct derivation is a sequence of correspondences that modify the state and transitions of the original LTS but do not modify the labels of the transitions. The labels of the transitions are defined by what the attacker can do and observe, a correct derivation must not change those properties of the attacker.

**Property 3 (Correctness of the 3rd step)**

For each transformation, the resulting machine must be proven to preserve the security properties of the original LTS. This is done in two steps:

(a) Develop a bisimilarity \(\approx^{AM}\) (Definition 6) over the modified LTS.

(b) Introduce a translation scheme: \(\{\cdot\}_AM\) that translates the source language state \(S\) into a machine state \(S^{AM}\) and upholds the following full abstraction property:

\[
S_1 \approx^{L^+} S_2 \iff \{S_1\} \approx^{AM} \{S_2\}
\]

5.2.4 Step 4: Implementation

The derived abstract machine \(AM^+\) must be (i) programmed in a language that is subsequently (ii) compiled securely to a target language \(\mathcal{T}\) whose programs
we denote as $T$. Such a hand-made implementation may introduce security leaks in point (i) that the methodology cannot detect. However, by adopting a secure (fully abstract) compiler in (ii), the implementation is guaranteed to preserve the security properties of $AM^+$ in the generated $T$ program.

**Property 4 (Correctness of the 4th step)**

*For the implementation of the abstract machine to be secure, the language in which it is written must be compiled with a secure (fully abstract) compiler denoted as: $\{\cdot\}_{T}^\downarrow$.*

$$S_1^{AM} \simeq_{AM} S_2^{AM} \iff \{S_1^{AM}\}^\downarrow \simeq_{T} \{S_2^{AM}\}^\downarrow$$

### 5.2.5 Validation

Once combined, the formal properties of each step of the methodology forms a chain of bi-implications (Figure 5.1) similar to ones found in work on verifying multi-transformation compilers [62].

This chain highlights the preservation and reflection of contextual equivalence in the source language $L$ down to the language $T$ targeted by the implementation of the abstract machine. This propagation ensures that the secure abstract machine implementation in $T$ preserves and reflects the abstractions of the source language $L$.

\[ L: \quad P_1 \simeq P_2 \]
\[ \Downarrow \quad \text{(P.1)} \]

\[ L^+: \quad S_1 \simeq^+ S_2 \quad \text{(P.2)} \]
\[ \Downarrow \quad \text{(P.3)} \]

\[ AM^+: \quad S_1^{AM+} \simeq_{AM+} S_2^{AM+} \quad \text{(P.4)} \]
\[ \Downarrow \]

\[ T: \quad T_1 \simeq_{T} T_2 \]

*Figure 5.1. Concatenation of the formal properties of each step of the methodology. P. is short for Property.*

The chain of propagation takes a detour between the extended language $L^+$ and the final abstract machine $AM^+$ towards bisimilarity relations that may seem less efficient than a direct full abstraction result between $L^+$ and $AM^+$. However, the resulting $AM^+$ is merely the final product of a series of systematic transformations that are applied to the original LTS over $L^+$. Proving that these transformations uphold our desired security properties is more easily achieved by relating congruent bisimilarity relations over the LTSs that result from these transformations. In fact, given a congruent bisimilarity
relation over the original LTS, the definition of such a bisimilarity relation over the derived machines and LTSs is almost straightforward.

5.3 Deriving a Secure CESK\(^+\) Machine for MiniML

Our goal in this chapter is to derive and implement a secure CESK\(^+\) machine for MiniML, denoted as the CESK\(^+\) machine. A CESK machine is a transition system of four elements: the term being evaluated (Control), a map from variables to values (Environment), a map from locations to values (Store) and a stack of evaluation contexts (Kontinuations) [22].

To ensure that we succeed in implementing this secure CESK\(^+\) machine we follow the methodology of Section 5.2. The first two steps of the methodology: extending MiniML with a foreign function interface that enables interaction with the loE-level attacker and deriving an LTS over which we define a congruent bisimulation, are not detailed as they are the focus of Chapter 3 and Chapter 4. In the third step of the methodology we derive a CESK formalisation from the LTS of Section 4.2.4 by making use of syntactical correspondences that preserve the formal properties MiniML (Section 5.3.1). This formalisation is then implemented using the PMA mechanism (Section 5.3.2).

Each step of the methodology of Section 5.3.3 includes a proposition that must proven to hold, establishing that the contextual equivalences of MiniML are preserved in the final CESK\(^+\). We validate our derived CESK\(^+\) by showing that all required propositions hold.

5.3.1 Step 3: Deriving the CESK\(^+\) Machine Through Syntactic Correspondences

We now derive a secure CESK machine, from the LTS of Section 4.2.4, by adapting Biernacka et al.’s syntactic correspondence between context-sensitive calculi and abstract machines [12]. Whereas, Biernacka et al. consider only closed languages and their associated closed-world abstract machines, we apply their syntactic correspondence to our foreign function interface formalisation to obtain an abstract machine that can interact with the outside world. This syntactic correspondence consists of four transformations that modify state and reduction rules.

1. **Context sensitive reduction**: this transformation replaces the evaluation contexts \(E\) with explicit continuation contexts \(K\).
2. **Closure conversion**: this transformation replaces \(\lambda\)-terms with closures.
3. **Refocusing and transition compression**: The first two transformations introduce various additional reduction steps, this transformation compresses the LTS by removing duplication.
4. **Unfolding closures**: this final transformation separates the closures of step 2 into control statements and environments.

Throughout this section the result of each transformation is annotated with a superscript that indicates to which transformation it results from.

For each transformation $T$ we prove that the contextual equivalences of MiniML are preserved in two steps. We first develop a bisimilarity $\approx^T$ (Definition 6) over the modified LTS. In a second step we use that bisimilarity $\approx^T$ to prove that there exist a compilation scheme:

$$\{\cdot\}_T$$

that compiles the current LTS state $S$, starting with the state

$$\langle M, s, \alpha \rangle$$

of Section 4.2.1, into the derived machine state $T^\mathcal{K}$ in a manner that preserves the contextual equivalences of MiniML. This is proven by relating the newly derived bisimilarity relation $\approx^T$ to the previously derived bisimilarity $\approx$ over the FFI, as required by Property 3.

$$\langle M_1, s_1, \alpha_1 \rangle \approx^l \langle M_2, s_2, \alpha_2 \rangle \iff \{\langle M_1, s_1, \alpha_1 \rangle\}_T \approx^T \{\langle M_2, s_2, \alpha_2 \rangle\}_T$$

Where $\approx^l$ is the bisimilarity relation over the foreign function interface between the loE-level attacker and MiniML (Section 4.2).

Each step of the derivation was implemented in Ocaml \(^1\) and tested using MiniML programs. In what follows, we provide an overview of the transformations as well as proof sketches for Property 3.

1. **Context Sensitive Reduction**

In this step explicit continuation contexts are derived to separate reduction contexts from the term being executed. To that end, the LTS defined in Section 4.2.4 is transformed into a new LTS $\mathcal{K}$ defined as the following triple.

$$\langle M^K, L, \xrightarrow{L} \mathcal{K} \rangle$$

The new state $M^K$ is a quintuple

$$\langle \mathcal{C}, \mu, \mathcal{K}, \mathcal{N}, s, \alpha \rangle$$

where $\mathcal{C}$ is a control and $\mathcal{K}$ the context. A control $\mathcal{C}$ is either a MiniML expression $\mathcal{E}$, a marshalling state $\mathcal{M}$ (which include the loE-level words $\mathcal{W}$) or a halt/block state $\mathcal{B}$.

$$\mathcal{C} ::= \mathcal{E} \mid \mathcal{M} \mid \mathcal{B}.$$ 

This block state $\mathcal{B}$ is needed to indicate that the LTS is halted, waiting on input from the attacker.

The contexts $\mathcal{K}$ of the new machine state are derived by first transforming MiniML’s evaluation contexts $\mathcal{E}$ (Section 2.1):

\(^1\)https://github.com/sylvarant/secure-abstract-machine
into explicit continuations contexts $K$, as follows.

\[
K ::= [\cdot] \mid K[[\cdot]e] \mid K[[\cdot]op e2] \mid K[([\cdot]op [\cdot])e2] \\
\mid K[([\cdot]cp e2)] \mid K[[v_1 \cdot []e]] \mid K[[v_j \cdot [\cdot]e]] \\
\mid K[[\langle\cdot\rangle]] \mid K[[let x = [\cdot]in e2]] \mid K[[!\cdot]] \\
\mid K[[if [\cdot] (\cdot)]e2] \mid K[[let x = [\cdot] in e2]]
\]

Next the type-annotated stack of evaluation contexts $\Sigma$ of the secure state $M$ (Section 3.4.1) is converted into an outer continuation that captures the 4 different sub-states of the secure state $M$ within the FFI (Section 3.4.3).

\[
k ::= [\cdot] \quad \text{Empty Stack} \\
\mid k[K : \tau \rightarrow \tau'] \quad \text{Blocked State} \\
\mid k[\circ K : \tau] \quad \text{Executing State} \\
\mid k[\downarrow K : \tau] \quad \text{Marshalling in State} \\
\mid k[\downarrow K : \tau] \quad \text{Marshalling out State}
\]

The new transition rules $L_K$ differ from the transitions $L$ of Section 4.2.4 in that they include explicit transitions to plug and construct the continuation contexts $K$, operations previously done implicitly through the evaluation contexts $E$. The rule M-R1 list below, for example, enforces the left to right order of MiniML by selecting the left expression $e_1$ as the control and converting the to be evaluated right term $e_2$ into a new continuation.

\[
\langle e_1 e_2, k[\circ K : \tau], \mu, N, s, \alpha \rangle \xrightarrow{\tau} \langle e_1, k[\circ K[\cdot] e_2] : \tau], \mu, N, s, \alpha \rangle \quad \text{(M-R1)}
\]

**Machine Formalisation** The full formalisation of the resulting machine is listed below. Most transition rules are highly similar to their original counterpart, excluding the changes to the state. Note that the rule $M$-Plug plugs a value into a continuation and is thus called at least once for every type of communication. Note also that the marshalling rules of Section 4.2.2 are not repeated within the machine formalisation to save a bit of space.

The formalisation is split into four sections: first the state and its contents are defined, next the internal machine reductions are denoted followed by a compressed listing of the marshalling rules and the interactions between the attacker and machine.
\[ M^K \in \text{State}^K \quad = \quad \text{Ctrl} \times \text{Storage} \times \text{Kont} \times \text{Map} \times \text{Memory} \times \text{Returns} \]

<table>
<thead>
<tr>
<th>Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c ) \in \text{Ctrl}</td>
<td>(:=\ {e} \mid \text{blk} \mid m )</td>
</tr>
<tr>
<td>( e ) \in \text{Expression}</td>
<td>(:=\ {v} \mid {x} \mid (e_1 \ e_2) \mid \langle e^{i \in 1..n} \rangle \mid e_1 \text{ cp } e_2 \mid e_1 \text{ op } e_2 \mid e.i \mid \text{if } e_1 \ e_2 \ e_3 \mid \text{index } e \mid \text{let } x = e_1 \text{ in } e_2 \mid !e \mid \text{ref } e \mid \text{exit } e )</td>
</tr>
<tr>
<td>( v ) \in \text{Value}</td>
<td>(:=\ \text{unit} \mid l_i \mid \overline{n} \mid b \mid (\lambda x: \tau.e) \mid \langle v^{i \in 1..n} \rangle \mid \tau \rightarrow \tau' \text{ Fa} )</td>
</tr>
<tr>
<td>( b ) \in \text{Bool}</td>
<td>(:=\ \text{true} \mid \text{false} )</td>
</tr>
<tr>
<td>( x ) \in \text{Var}</td>
<td>(:=\ \text{a set of identifiers.} )</td>
</tr>
<tr>
<td>( \text{op} ) \in \text{Oper}</td>
<td>(:=\ + \mid - \mid * )</td>
</tr>
<tr>
<td>( \text{cp} \in \text{Comp}</td>
<td>(:=\ &lt;</td>
</tr>
<tr>
<td>( \tau \in \text{Type}</td>
<td>(:=\ \text{Bool} \mid \text{Int} \mid \text{Unit} \mid \tau_1 \rightarrow \tau_2 \mid \text{Ref } \tau )</td>
</tr>
<tr>
<td>( k \in \text{Kont}</td>
<td>(:=\ [] \mid k.K: \tau \rightarrow \tau' \mid k[\circ K: \tau] \mid k[&lt;: K: \tau] )</td>
</tr>
<tr>
<td>( K \in \text{Kin}</td>
<td>(:=\ [\cdot] \mid K[(\cdot): e] \mid K[v[\cdot]] \mid K[(\cdot): \text{op } e_2] \mid K[(v_1 \text{ op } [\cdot])] )</td>
</tr>
<tr>
<td>( n_i \in \text{Name}</td>
<td>(:=\ \text{an enumerable set.} )</td>
</tr>
<tr>
<td>( N \in \text{Map}</td>
<td>(:=\ \text{Name} \rightarrow (\text{Expression} \times \text{Type}) )</td>
</tr>
<tr>
<td>( s \in \text{Memory}</td>
<td>(:=\ \text{a descriptor of low-level memory} )</td>
</tr>
<tr>
<td>( \overline{a} \in \text{Returns}</td>
<td>(:=\ \text{a stack of low-level addresses} )</td>
</tr>
</tbody>
</table>

\[
\langle v, k[\circ K[K'] : \tau], N, s, \overline{a} \rangle \xrightarrow{\tau} \langle K'[v], k[\circ K : \tau], N, s, \overline{a} \rangle \quad \text{(M-Plug)}
\]
\[
\langle (e_1 \ e_2), k[\circ K : \tau], N, s, \overline{a} \rangle \xrightarrow{\tau} \langle e_1, k[\circ K[\cdot]: e_2] : \tau], N \rangle \quad \text{(M-R1)}
\]
\[
\langle (v_1 \ e_2), k[\circ K : \tau], N \rangle \xrightarrow{\tau} \langle e_2, k[\circ K[v_1[\cdot]] : \tau], N, s, \overline{a} \rangle \quad \text{(M-R2)}
\]
\[
\langle (\lambda x: \tau.e) v, k, N, s, \overline{a} \rangle \xrightarrow{\tau} \langle e[v/x] : \tau, k, N, s, \overline{a} \rangle \quad \text{(M-App)}
\]
\[
\langle (\text{if } e_1 \ e_2 \ e_3), k[\circ K : \tau], N, s, \overline{a} \rangle \xrightarrow{\tau} \langle e_1, k[\circ K[\text{if } \cdot e_2 \ e_3] : \tau], N, s, \overline{a} \rangle \quad \text{(M-R3)}
\]
\[
\langle (\text{true } e_2 \ e_3), k, N, s, \overline{a} \rangle \xrightarrow{\tau} \langle e_2, k, N \rangle \quad \text{(M-IFT)}
\]
\[
\langle (\text{false } e_2 \ e_3), k, N \rangle \xrightarrow{\tau} \langle e_3, k, N, s, \overline{a} \rangle \quad \text{(M-IFF)}
\]
\[
\langle (\text{let } x = e_1 \text{ in } e_2), k[\circ K : \tau], N, s, \overline{a} \rangle \xrightarrow{\tau} \langle e_1, k[\circ K[\text{let } x = \cdot \text{ in } e_2] : \tau], N, s, \overline{a} \rangle \quad \text{(M-R4)}
\]
\[
\langle (\text{ref } e), \mu, k[\circ K : \tau], N, s, \overline{a} \rangle \xrightarrow{\tau} \langle e, \mu, k[\circ K[\text{ref } \cdot] : \tau], N, s, \overline{a} \rangle \quad \text{(M-Ra)}
\]

105
\[
\langle \text{ref} v, \mu, k \circ K : \tau, N, s, \overline{a} \rangle \xrightarrow{t} \langle l_i, (\mu, l_i \mapsto v), k \circ K : \tau, N, s, \overline{a} \rangle \quad \text{(M-Ref)}
\]
\[
\langle l_i, \mu, k \circ K : \tau, N, s, \overline{a} \rangle \xrightarrow{t} \langle e, \mu, k \circ K[\langle \cdot \rangle] : \tau, N, s, \overline{a} \rangle \quad \text{(M-Rd)}
\]
\[
\langle \lambda t. \mu, k \circ K : \tau, N, s, \overline{a} \rangle \xrightarrow{t} \langle v, \mu, k \circ K : \tau, N, s, \overline{a} \rangle \quad \text{(M-Der)}
\]

where \( (l_i) = v \)
\[
\langle \text{index} e, \mu, k \circ K : \tau, N, s, \overline{a} \rangle \xrightarrow{t} \langle e, \mu, k \circ K[\text{index} \[ \cdot \]] : \tau, N, s, \overline{a} \rangle \quad \text{(M-Rh)}
\]
\[
\langle \text{index} l_i, \mu, k \circ K : \tau, N, s, \overline{a} \rangle \xrightarrow{t} \langle l_i, \mu, k \circ K : \tau, N, s, \overline{a} \rangle \quad \text{(M-Index)}
\]
\[
\langle \text{exit} e, \mu, k \circ K : \tau, N, s, \overline{a} \rangle \xrightarrow{t} \langle e, \mu, k \circ K[\text{exit} \[ \cdot \]] : \tau, N, s, \overline{a} \rangle \quad \text{(M-Exit)}
\]
\[
\langle \text{exit} v, \mu, k \circ K : \tau, N, s, \overline{a} \rangle \xrightarrow{t} \langle v, \mu, k \circ K[\langle \cdot \rangle] : \tau, N, s, \overline{a} \rangle \quad \text{(M-Rp)}
\]
\[
\langle \langle \lambda t. \mu, k \circ K : \tau, N, s, \overline{a} \rangle \xrightarrow{t} \langle v, \mu, k \circ K : \tau, N, s, \overline{a} \rangle \quad \text{(M-Proj)}
\]
\[
\langle e_1 : \text{e}, \mu, k \circ K : \tau, N, s, \overline{a} \rangle \xrightarrow{t} \langle e_2, \mu, k \circ K : \tau, N, s, \overline{a} \rangle \quad \text{(M-RSq)}
\]
\[\text{unit} ; e_2, \mu, k \circ K : \tau, N, s, \overline{a} \xrightarrow{t} \langle e_2, \mu, k \circ K : \tau, N, s, \overline{a} \rangle \quad \text{(M-Seq)}\]
\[
\langle e_1 : = \text{e}, \mu, k \circ K : \tau, N, s, \overline{a} \rangle \xrightarrow{t} \langle e_1, \mu, k \circ K[\langle \cdot \rangle := e_2] : \tau, N, s, \overline{a} \rangle \quad \text{(M-Rs)}
\]
\[
\langle l_i, \mu, k \circ K[\langle \cdot \rangle := e_2] : \tau, N, s, \overline{a} \rangle \xrightarrow{t} \langle e_2, \mu, k \circ K[l_i := \langle \cdot \rangle] : \tau, N, s, \overline{a} \rangle \quad \text{(M-RS2)}
\]
\[
\langle l_i := v, \mu, k \circ K : \tau, N, s, \overline{a} \rangle \xrightarrow{t} \langle \text{unit} ; (\mu, l_i \mapsto v), k \circ K : \tau, N, s, \overline{a} \rangle \quad \text{(M-Set)}
\]
\[
\langle \text{fix} e, \mu, k \circ K : \tau, N, s, \overline{a} \rangle \xrightarrow{t} \langle e, \mu, k \circ K[\text{fix} \[ \cdot \]] : \tau, N, s, \overline{a} \rangle \quad \text{(M-Rf)}
\]
\[
\langle \text{fix} (\lambda x : \tau. e'), \mu, k \circ K : \tau, N, s, \overline{a} \rangle \xrightarrow{t} \langle e'[\text{fix} (\lambda x : \tau. e')/x], \mu, k \circ K : \tau, N, s, \overline{a} \rangle \quad \text{(M-Fix)}
\]
\[
\langle (e_1 \text{ ope}_2), \mu, k \circ K : \tau, N, s, \overline{a} \rangle \xrightarrow{t} \langle e_1, \mu, k \circ K[[\cdot] \text{ ope}_2] : \tau, N, s, \overline{a} \rangle \quad \text{(M-Ro)}
\]
\[
\langle v, \mu, k \circ K[[\cdot] \text{ ope}_2] : \tau, N, s, \overline{a} \rangle \xrightarrow{t} \langle v, \mu, k \circ K[[\cdot] \text{ ope}_2] : \tau, N, s, \overline{a} \rangle \quad \text{(M-Op)}
\]
\[
\langle (v_1 \text{ op}_v_2), \mu, k \circ K : \tau, N, s, \overline{a} \rangle \xrightarrow{t} \langle v_1, \mu, k \circ K : \tau, N, s, \overline{a} \rangle \quad \text{(M-Op2)}
\]
\[
\langle (e_1 \text{ cpe}_e_2), \mu, k \circ K : \tau, N, s, \overline{a} \rangle \xrightarrow{t} \langle e_1, \mu, k \circ K[[\cdot] \text{ cpe}_e_2] : \tau, N, s, \overline{a} \rangle \quad \text{(M-Rc)}
\]
\[
\langle v, \mu, k \circ K[[\cdot] \text{ cpe}_e_2] : \tau, N, s, \overline{a} \rangle \xrightarrow{t} \langle v, \mu, k \circ K[[\cdot] \text{ cpe}_e_2] : \tau, N, s, \overline{a} \rangle \quad \text{(M-Cp)}
\]
\[
\langle (v_1 \text{ cpe}_v_2), \mu, k \circ K : \tau, N, s, \overline{a} \rangle \xrightarrow{t} \langle b, \mu, k \circ K : \tau, N, s, \overline{a} \rangle \quad \text{(M-Cp2)}
\]
\[
\langle v, \mu, k \circ \langle \cdot \rangle : \tau, N, s, \overline{a} \rangle \xrightarrow{t} \langle v, \mu, k \circ \langle \cdot \rangle : \tau, N, s, \overline{a} \rangle \quad \text{(M-Done)}
\]
\[
\langle m, \mu, k \circ \langle \cdot \rangle : \tau, N, s, \overline{a} \rangle \xrightarrow{t} \langle m', \mu, k \circ N', s, \overline{a} \rangle \quad \text{where} \quad \langle \dot{m} \rangle^N \rightarrow \langle m' \rangle^N
\]
\text{(M-Marshout)}
\]
\[ \langle m, \mu, k \circ K : \tau \rangle, N, s, \overline{a} \xrightarrow{\tau} \langle m', \mu, k \circ K : \tau \rangle, N, s, \overline{a} \]  
where \( \frac{1}{s} \left\| m \right\|_\tau \xrightarrow{\frac{1}{s} \left\| m' \right\|_\tau} \)

(M-Marshin)

\[ \langle wr, \mu, k \circ K : \tau \rangle, N, s, \overline{a} \xrightarrow{\sqrt{\text{E-I}}} \langle blk, \emptyset, [], *, s, \emptyset \rangle \]

(Done)

\[ \langle w, \mu, k \circ [\cdot : \tau], N, s, a_r : \overline{a} \rangle \xrightarrow{\text{ret } a_r.w} \langle blk, \mu, k, N, s, \overline{a} \rangle \]

(M-Ret)

\[ \langle \overline{w}, \mu, k \circ [\cdot : (\tau^{e_1.n}_t) \rangle, N, s, a_r : a_d : \overline{a} \rangle \xrightarrow{\text{write}(a_d.w), \text{ret } a_r.w} \]

(M-RetT)

\[ \langle blk, \mu, k, N, s, \overline{a} \rangle \xrightarrow{s \vdash \text{unprotectedCode}(a_r) \text{ and } s \vdash \text{unprotectedData}(a_d + |\overline{w}|) \}

\[ \langle \overline{w}, \mu, k \circ [\cdot : (\tau^{e_1.n}_t) \rangle, N, s, a_r : a_d : \overline{a} \rangle \xrightarrow{\text{write}(a_d.w), \text{ret } a_r.w} \]

(E-RetT)

\[ \langle blk, \emptyset, [], *, s, \emptyset \rangle \xrightarrow{s \nvdash \text{unprotectedCode}(a_r) \text{ or } s \nvdash \text{unprotectedData}(a_d + |\overline{w}|) \}

\[ \langle a_d, \mu, k \circ [\cdot : (\tau^{e_1.n}_t) \rangle, s, \overline{a} \rangle \xrightarrow{\text{read}(a_d.w)} \]

(M-ReadT)

\[ \langle \overline{w}, \mu, k \circ [\cdot : (\tau^{e_1.n}_t) \rangle, s, \overline{a} \rangle \]

\[ \langle blk, \emptyset, [], *, s, \emptyset \rangle \xrightarrow{s \vdash \text{returnback}(a) \}

(A-Ret)

\[ \langle blk, \mu, k, N, s, \overline{a} \rangle \xrightarrow{\text{setEntryPoint}(a)} \]

(E-AC)

\[ \langle blk, \mu, k, N, s, \overline{a} \rangle \xrightarrow{\text{call } a(w_a, a_d), \text{setEntryPoint}(a)} \]

(A-Deref)

\[ \langle w, \mu, k \circ [\cdot : (\tau^{e_1.n}_t) \rangle \xrightarrow{l_i, \text{Ref } \tau} \langle l_i, \text{Ref } \tau \rangle \text{ and } s \vdash \text{applyEntryPoint}(a) \]

(A-Set)

\[ \langle blk, \mu, k, N, s, \overline{a} \rangle \xrightarrow{\text{call } a(w_a, a_d), \text{applyEntryPoint}(a)} \]

(A-Ref)

\[ \langle (\tau \to \tau_2 \cdot \text{F}_a), \mu, k \circ K : \tau \rangle, N, s, \overline{a} \xrightarrow{\text{call } a_f(w)} \]

(M-Call)

107
Proof Sketch of Property 3. Given a bisimilarity ≈ over LTS^K, we must define a compilation scheme \{\cdot\}_K such that:

\[ \langle M_1, s_1, \alpha \rangle \approx \langle M_2, s_2, \alpha \rangle \iff \{ \langle M_1, s_1, \alpha \rangle \}_K \approx K \{ \langle M_2, s_2, \alpha \rangle \}_K \]

The following compilation scheme \{\cdot\}_K:

\[
\{ \langle \Sigma \circ e : \tau, \mu, N, s, \alpha \rangle \}_K = \langle e, \{ k \}_E [\circ [:] : \tau], \mu, N, s, \alpha \rangle \\
\{ \langle \Sigma \triangleleft m : \tau, \mu, N, s, \alpha \rangle \}_K = \langle m, \{ k \}_E [\triangleleft [:] : \tau], \mu, N, s, \alpha \rangle \\
\{ \langle \Sigma \triangleright m : \tau, \mu, N, s, \alpha \rangle \}_K = \langle m, \{ k \}_E [\triangleright [:] : \tau], \mu, N, s, \alpha \rangle \\
\{ \langle \Sigma, E : \tau \rangle \}_E = \{ \Sigma \}_E [K : \tau] \mid \{ e \}_E = [:] \quad \text{where } K \text{ explicitates } E
\]

compiles states M that are bisimilar in ∼ into states M^K that are bisimilar in ∼ as the new transitions of LTS^K are silent transitions that are ignored by our weak bisimulations.

2. Closure Conversion

In this second transformation the λ-terms of MiniML are converted into closures. As a result, the previously derived LTS^K is transformed into a new LTS^C defined by the following triple.

\[ (M, C, L, \rightarrow^C) \]

The new state M^C is a quintuple:

\[ \langle e^C, \mu, k, N, \alpha \rangle \]

where the new control terms e^C are the original controls e extended with a map of substitutions e_c.

\[ e := \ast | e \cdot (cl / x) \]

The controls e^C thus now consists of closures, marshalling states and a blocked state extended with the map of substitutions.

\[ e^C := cl | m[e] | blk[e] \]

These closures cl are the original MiniML expressions e extended with a map of substitutions e as is the case for the λ_θ-calculus [12]. The closures cl are formally defined as follows.
Closures: \[ \text{cl} ::= e[e] | (\text{cl} \text{ cl}) | \text{if} \text{ cl}_1 \text{ cl}_2 \text{ cl}_3 | \text{cl}_1 \text{ op} \text{ cl}_2 | \text{cl}_1 \text{ cp} \text{ cl}_2 \\
| \text{let x = cl}_1 \text{ in} \text{ cl}_2 | \text{cl}_1;\text{cl}_2 | \text{cl}_1 := \text{ cl}_2 | \text{index cl} | \text{cl}.i \\
| \text{fix cl} | \text{letrec x : } \tau = \text{ cl}_1 \text{ in} \text{ cl}_2 | \text{ref cl} | \langle \text{cl}_i^{\in I..n} \rangle \\
| \text{exit cl} \]

Note that in this closure calculus a \(\lambda\)-term is an expression, while its closure is a value. Note also that the explicit continuations \(K\) now also use closures \(\text{cl}\) instead of controls \(c\).

The new transition rules \(L \rightarrow C\) differ from \(L \rightarrow K\) in that their control element is a closure in all case but the halted and marshalling states. This requires rules that propagate the map of substitutions \(e\) across sub-terms, such as, for example, in the following transition rule (Prop-IF) where \(e\) is propagated across the sub-terms of the term \(\text{if}\).

\[ (\langle \text{if} e_1 e_2 e_3 \rangle[e], \mu, k, N, s, \alpha) \xrightarrow{\tau} (\langle \text{if} e_1[e] e_2[e] e_3[e] \rangle, \mu, k, N, s, \alpha) \quad \text{(Prop-IF)} \]

Having closures as controls of the transition system also requires us to update the reduction rules that use substitution. Function application, for example, is updated to make use of the map of substitutions by splitting it into two new rules M-B and M-V. These rules respectively add a new substitution to the map when performing an application and fetch a substitution from that same map when reducing a standalone variable.

\[ (\langle (\lambda x : \tau . e)[v], \mu, k, N, s, \alpha \rangle \xrightarrow{\tau} (\langle e[e[v'/x]] \rangle, \mu, k, N, s, \alpha) \quad \text{(M-B)} \]

\[ (x[\cdots(\text{cl}/x)\cdots], \mu, k, N, s, \alpha) \xrightarrow{\tau} (\langle \text{cl}, \mu, k, N, s, \alpha \rangle \quad \text{(M-V)} \]

Where \(x[\cdots(\text{cl}/x)\cdots]\), matches the first instance of the variable \(x\) within the substitution map \(e\). Note that a more accurate formal representation of this implementation detail can be achieved by making use of De Bruijn indices [18].

**Machine Formalisation** The full formalisation is listed in what follows. The closure conversion introduces many new propagation rules (Prop-*), that propagate the environment \(e\), adding many additional rules to the machine formalisation. The addition of closures does not, however, introduce new labels as they are masked by the names of the map \(N\).

Again the formalisation is split between defining the state, the internal reductions, the internal marshalling rules and and the observable interactions between the attacker and the machine.
\[ M^C \in \text{State}^C = \text{Ctrl} \times \text{Storage} \times \text{Kont} \times \text{Map} \times \text{Memory} \times \text{Returns} \]

\[
c^C \in \text{Ctrl} ::= \text{cl} | \text{blk}[e] | m[e]
\]

\[
\text{cl} \in \text{Closure} ::= \text{cl}[e] (\text{cl} \text{cl}) | \text{if cl1 cl2 cl3} | \text{let } x = \text{cl1 in cl2}
\]

\[
\text{cl1;cl2} | \text{cl1 op cl2} | \text{cl1 := cl2} | \text{index cl} | \text{fix cl}
\]

\[
\text{letrec } x : \tau = \text{cl1 in cl2} | \text{ref cl} | \text{cl1 cp cl2} | \text{cl.i}
\]

\[
\langle \text{i} \rangle^{i} \in 1..n \rangle \mid \text{exit cl}
\]

\[
e \in \text{Expression} ::= v | x | (e1 e2) | \langle e_i^{i} \in 1..n \rangle | e1 \text{ cp e2}
\]

\[
| e1 \text{ op e2} | e.i | \text{if e1 e2 e3} | \text{index e}
\]

\[
| \text{let } x = e1 \text{ in e2} | !e | \text{ref e} | \text{exit e}
\]

\[
\text{letrec } x : \tau = e1 \text{ in e2} | e1;e2 | \text{fix e} | (\lambda x : \tau.e)
\]

\[
v \in \text{Value} ::= \tau \rightarrow \tau'\text{Fa} | (\lambda x : \tau.e)[e] | \text{b} | \text{unit} | li | \pi | \langle \text{i} \rangle^{i} \in 1..n \rangle
\]

\[
b \in \text{Bool} ::= \text{true} \mid \text{false}
\]

\[
x \in \text{Var} ::= \text{a set of identifiers}.
\]

\[
op \in \text{Oper} ::= \text{+} \mid - \mid *
\]

\[
\text{cp} \in \text{Comp} ::= < \mid > \mid ==
\]

\[
\tau \in \text{Type} ::= \text{Bool} \mid \text{Int} \mid \text{Unit} \mid \tau_1 \rightarrow \tau_2 \mid \text{Ref } \tau \mid \langle \tau_i^{i} \in 1..n \rangle
\]

\[
e \in \text{Env} = \text{e} \cdot (\text{cl}/x)
\]

\[
k \in \text{Kont} ::= [\cdot] | \text{k[K Kl : } \tau \rightarrow \tau' ] | \text{k[K \circ } \tau \mid \text{k(K : } \tau \rangle
\]

\[
\mid \text{k } \uplus \text{K : } \tau \rangle
\]

\[
\text{K } \in \text{Kin} ::= [\cdot] | \text{K}([\cdot] \text{CL}) | \text{K}([v] [\cdot]) | \text{K}([\cdot] \text{op cl2}) | \text{K}([v_1 \text{ op } [\cdot]])
\]

\[
| \text{K}([\cdot] \text{ cp cl2}) | \text{K}([v_1 \text{ cp } [\cdot]]) | \text{K}([v_i^{i} \in 1..j, [\cdot], \text{cl}^{j} [k e j + 1..n]])
\]

\[
| \text{K}([\cdot] \text{ index [i]}) | \text{K}([\text{let } x = [\cdot] \text{ in cl2}) | \text{K}([! [\cdot]])
\]

\[
| \text{K}([\cdot], d) | \text{K}([\text{fix } [\cdot]]) | \text{K}([\cdot] := \text{cl2}) | \text{K}[v := [\cdot])]
\]

\[
| \text{K}([\text{if } [\cdot] : \text{cl2 cl3}) | \text{K}([\text{exit } [\cdot]]) | \text{K}([\text{ref } [\cdot]]) | \text{K}([\cdot]; \text{cl2})]
\]

\[\text{n}i \in \text{Name} \text{ an enumerable set.}\]

\[
\text{N} \in \text{Map} = \text{Name } \rightarrow (\text{Ctrl } \times \text{Type})
\]

\[
s \in \text{Memory} \text{ a descriptor of low-level memory}
\]

\[
\text{\overline{a}} \in \text{Returns} \text{ a stack of low-level addresses}
\]

\[
(v[e], k[\circ K[K'] : \tau] : N, s, \overline{a}) \xrightarrow{s} \langle K[v[e], k[\circ K : \tau] : N, s, \overline{a}]
\]

(M-Plug)

\[
\langle e1 \rangle e2[e], k[\circ K[K'] : \tau] : N, s, \overline{a} \xrightarrow{s} \langle e_1 \rangle e_2[e], k[\circ K[K'] : N, s, \overline{a}]
\]

(Prop-App)

\[
\langle c_1 c_2, k[\circ K[K] : N, s, \overline{a}] \xrightarrow{s} \langle c_1, k[\circ K[\cdot] c_2 : N, s, \overline{a}]
\]

(M-R1)

\[
\langle v_1[e] c_2, k[\circ K[K'] : N, s, \overline{a}] \xrightarrow{s} \langle c_2, k[\circ K[v_1[e][\cdot] : \tau] : N, s, \overline{a}]
\]

(M-R2)

\[
\langle \lambda x : \tau.e[e][v[e]] : k[\circ K[K'] : N, s, \overline{a}] \xrightarrow{s} e[e \cdot (v[e]/x)], k, N, s, \overline{a}
\]

(M-B)

\[
\langle x[e], \mu, k, N, s, \overline{a} \xrightarrow{s} \langle \text{cl, } \mu, k, N, s, \overline{a} \rangle \text{ where e(x) = cl}
\]

(M-V)

\[
\langle \text{if e_1 e_2 e_3[e][k[\circ K[K'] : N, s, \overline{a}] \xrightarrow{s}
\]

(Prop-If)
\[
\langle \text{if } c_1 \text{ and } c_2 \text{ and } c_3, k \circ K : \tau, N, s, \bar{a} \rangle \xrightarrow{r} \\
\langle c_1, k \circ K \langle \text{if } [:] \text{ and } c_2 \text{ and } c_3 \rangle : \tau, N, s, \bar{a} \rangle
\]
(M-R3)

\[
\langle \text{if } e_1 \text{ and } c_2 \text{ and } c_3, k, N, s, \bar{a} \rangle \xrightarrow{r} \langle c_2, k, N, s, \bar{a} \rangle
\]
(M-IFT)

\[
\langle \text{if } e_1 \text{ and } c_2 \text{ and } c_3, k, N, s, \bar{a} \rangle \xrightarrow{r} \langle c_3, k, N, s, \bar{a} \rangle
\]
(M-IFF)

\[
\langle \text{let } x = e_1 \text{ in } e_2, k \circ K : \tau, N, s, \bar{a} \rangle \xrightarrow{r} \\
\langle \text{let } x = e_1 \text{ in } e_2, k \circ K : \tau, N, s, \bar{a} \rangle
\]
(Prop-Let)

\[
\langle \text{let } x = c_1 \text{ in } c_2, k \circ K : \tau, N, s, \bar{a} \rangle \xrightarrow{r} \\
\langle c_1, k \circ K \langle \text{let } x = [:] \text{ in } c_2 \rangle : \tau, N, s, \bar{a} \rangle
\]
(M-R4)

\[
\langle \text{let } x = v[e] \text{ in } e_1, k, N, s, \bar{a} \rangle \xrightarrow{r} \langle e_1 \cdot (v[e]/x), k, N, s, \bar{a} \rangle
\]
(M-Let)

\[
\langle \text{ref } e, \mu, k \circ K : \tau, N, s, \bar{a} \rangle \xrightarrow{r} \langle \text{ref } e, \mu, k \circ K : \tau, N, s, \bar{a} \rangle
\]
(Prop-Ref)

\[
\langle \text{ref } c, \mu, k \circ K : \tau, N, s, \bar{a} \rangle \xrightarrow{r} \langle c, \mu, k \circ K \langle \text{ref } [:] \rangle : \tau, N, s, \bar{a} \rangle
\]
(M-Ra)

\[
\langle \text{ref } v[e], \mu, k \circ K : \tau, N, s, \bar{a} \rangle \xrightarrow{r} \langle l_i, (\mu, l_i \rightarrow v[e]), k \circ K : \tau, N, s, \bar{a} \rangle
\]
(M-Ref)

\[
\langle \text{let } e, \mu, k \circ K : \tau, N, s, \bar{a} \rangle \xrightarrow{r} \langle \text{let } e, \mu, k \circ K : \tau, N, s, \bar{a} \rangle
\]
(Prop-Der)

\[
\langle \text{let } c, \mu, k \circ K : \tau, N, \bar{a} \rangle \xrightarrow{r} \langle c, \mu, k \circ K \langle \text{let } [:] \rangle : \tau, N, \bar{a} \rangle
\]
(M-Der)

\[
\langle \text{index } e, \mu, k \circ K : \tau, N, s, \bar{a} \rangle \xrightarrow{r} \\
\langle \text{index } e, \mu, k \circ K : \tau, N, s, \bar{a} \rangle
\]
(Prop-I)

\[
\langle \text{index } c, \mu, k \circ K : \tau, N, s, \bar{a} \rangle \xrightarrow{r} \\
\langle c, \mu, k \circ K \langle \text{index } [:] \rangle : \tau, N, s, \bar{a} \rangle
\]
(M-Ri)

\[
\langle \text{index } l_i[e], \mu, k \circ K : \tau, N, s, \bar{a} \rangle \xrightarrow{r} \langle l_i, (\mu, l_i \rightarrow v[e]), k \circ K : \tau, N, s, \bar{a} \rangle
\]
(M-Index)

\[
\langle \text{let } e_1 ; e_2, \mu, k \circ K : \tau, N, s, \bar{a} \rangle \xrightarrow{r} \\
\langle \text{let } e_1 ; e_2, \mu, k \circ K : \tau, N, s, \bar{a} \rangle
\]
(Prop-Seq)

\[
\langle c_1 \text{ and } c_2, \mu, k \circ K : \tau, N, s, \bar{a} \rangle \xrightarrow{r} \\
\langle c_1, \mu, k \circ K \langle [:] \text{ and } c_2 \text{ and } \rangle : \tau, N, s, \bar{a} \rangle
\]
(M-RSq)

\[
\langle \text{unit } e, c_2, \mu, k \circ K : \tau, N, s, \bar{a} \rangle \xrightarrow{r} \langle c_2, \mu, k \circ K : \tau, N, s, \bar{a} \rangle
\]
(M-Seq)

\[
\langle \text{let } e_1 := e_2, \mu, k \circ K : \tau, N, s, \bar{a} \rangle \xrightarrow{r} \\
\langle \text{let } e_1 := e_2, \mu, k \circ K : \tau, N, s, \bar{a} \rangle
\]
(Prop-Set)

\[
\langle c_1 := c_2, \mu, k \circ K \langle [:] \text{ := } c_2 \rangle : \tau, N, s, \bar{a} \rangle \xrightarrow{r} \\
\langle c_1, \mu, k \circ K \langle [:] \text{ := } c_2 \rangle : \tau, N, s, \bar{a} \rangle
\]
(M-Rs)

\[
\langle l_i[e], \mu, k \circ K \langle [:] \text{ := } c_2 \rangle : \tau, N, s, \bar{a} \rangle \xrightarrow{r} \\
\langle c_2, \mu, k \circ K \langle l_i[e] := [:] \rangle : \tau, N, s, \bar{a} \rangle
\]
(M-Rs2)

\[
\langle l_i[e] := v[e'], \mu, k \circ K : \tau, N, s, \bar{a} \rangle \xrightarrow{r} \\
\langle \text{let } e, (\mu, l_i \rightarrow v[e']), k \circ K : \tau, N, s, \bar{a} \rangle
\]
(M-Set)

\[
\langle \text{fix } e, \mu, k \circ K : \tau, N, s, \bar{a} \rangle \xrightarrow{r} \langle \text{fix } e, \mu, k \circ K : \tau, N, s, \bar{a} \rangle
\]
(Prop-F)

\[
\langle \text{fix } c, \mu, k \circ K : \tau, N, s, \bar{a} \rangle \xrightarrow{r} \langle c, \mu, k \circ K \langle \text{fix } [:] \rangle : \tau, N, s, \bar{a} \rangle
\]
(M-Rf)
\[\langle \text{fix}(\lambda x : \tau'.e)[e], \mu, k \circ K : \tau], N, s, \overline{a} \rangle \stackrel{r}{\rightarrow} \langle e[e \cdot (\text{fix}(\lambda x : \tau'.e)[e/x]), \mu, k \circ K : \tau], N, s, \overline{a} \rangle\]  
(M-Fix)

\[\langle e_1 \text{ op}_{e_2}[e], \mu, k \circ K : \tau], N, s, \overline{a} \rangle \stackrel{r}{\rightarrow} \langle (e_1[e \text{ op}_{e_2}[e]), \mu, k \circ K : \tau], N, s, \overline{a} \rangle\]  
(Prop-O)

\[\langle \text{cl}_1 \text{ op}_{\text{cl}_2}[e], \mu, k \circ K : \tau], N, s, \overline{a} \rangle \stackrel{r}{\rightarrow} \langle (\text{cl}_1[e \text{ op}_{\text{cl}_2}[e]), \mu, k \circ K : \tau], N, s, \overline{a} \rangle\]  
(M-Ro)

\[\langle v[e], \mu, k \circ [K[([.] \text{ op}_{\text{cl}_2}[]) : \tau]], N, s, \overline{a} \rangle \stackrel{r}{\rightarrow} \langle (v[e], \mu, k \circ [K[([.] \text{ op}_{\text{cl}_2}[)]) : \tau], N, s, \overline{a} \rangle\]  
(M-Op)

\[\langle (v_1[e \text{ op}_{v_2}[e']], \mu, k \circ K : \tau], N, s, \overline{a} \rangle \stackrel{r}{\rightarrow} \langle (v'[e], \mu, k \circ K : \tau], N, s, \overline{a} \rangle\]  
(M-Op2)

\[\langle \langle e_1 \text{ cp}_{e_2}[e], \mu, k \circ K : \tau], N, s, \overline{a} \rangle \stackrel{r}{\rightarrow} \langle (\langle e_1[e \text{ cp}_{e_2}[e]), \mu, k \circ K : \tau], N, s, \overline{a} \rangle\]  
(Prop-C)

\[\langle \text{cl}_1 \text{ cp}_{\text{cl}_2}[e], \mu, k \circ K : \tau], N, s, \overline{a} \rangle \stackrel{r}{\rightarrow} \langle (\text{cl}_1[e \text{ cp}_{\text{cl}_2}[e]), \mu, k \circ K : \tau], N, s, \overline{a} \rangle\]  
(M-Rc)

\[\langle v[e], \mu, k \circ K[[.] \text{ cp}_{\text{cl}_2}[]) : \tau], N, s, \overline{a} \rangle \stackrel{r}{\rightarrow} \langle (v[e], \mu, k \circ K[[.] \text{ cp}_{\text{cl}_2}[]) : \tau], N, s, \overline{a} \rangle\]  
(M-Cp)

\[\langle (v_1[e \text{ cp}_{v_2}[e']], \mu, k \circ K : \tau], N, s, \overline{a} \rangle \stackrel{r}{\rightarrow} \langle (b[e], \mu, k \circ K : \tau], N, s, \overline{a} \rangle\]  
(M-Cp2)

\[\langle \langle \text{exit}[e][e], \mu, k \circ K : \tau], N, s, \overline{a} \rangle \stackrel{r}{\rightarrow} \langle (\text{exit}[e][e]), \mu, k \circ K : \tau], N, s, \overline{a} \rangle\]  
(Prop-Ex)

\[\langle \langle \text{exit}[cl], \mu, k \circ K : \tau], N, s, \overline{a} \rangle \stackrel{r}{\rightarrow} \langle \text{cl}[u] \mu, k \circ K[([.] \text{ exit}[]) : \tau], N, s, \overline{a} \rangle\]  
(M-Re)

\[\langle \langle \text{exit}[v[e]], \mu, k \circ K[([.] \text{ exit}[v]) : \tau], N, s, \overline{a} \rangle \stackrel{r}{\rightarrow} \langle \langle v[0], \mu, k \circ [K[([.] \text{ exit}[v]) : \tau], N, s, \overline{a} \rangle\]  
(M-Exit)

\[\langle \langle e.i][e], \mu, k \circ K : \tau], N, s, \overline{a} \rangle \stackrel{r}{\rightarrow} \langle (e[i][e], \mu, k \circ K : \tau], N, s, \overline{a} \rangle\]  
(Prop-Proj)

\[\langle \langle \text{cl}[i], \mu, k \circ K : \tau], N, s, \overline{a} \rangle \stackrel{r}{\rightarrow} \langle \text{cl}[i] \mu, k \circ K[([.] \text{ cl}[i]) : \tau], N, s, \overline{a} \rangle\]  
(M-Rproj)

\[\langle (\text{cl}[i_{\leq n}]), .i], \mu, k \circ K : \tau], N, s, \overline{a} \rangle \stackrel{r}{\rightarrow} \langle (\text{cl}[i] \mu, k \circ K[([.] \text{ cl}[i_{\leq n}]) : \tau], N, s, \overline{a} \rangle\]  
(M-Proj)

\[\langle v[e], \mu, k \circ [K[([.] : \tau], N, s, \overline{a} \rangle \stackrel{r}{\rightarrow} \langle (v[e], \mu, k \circ [K[([.] : \tau], N, s, \overline{a} \rangle\]  
(M-Done)

\[\langle m[e], \mu, k \circ [K[([.] : \tau], N, s, \overline{a} \rangle \stackrel{r}{\rightarrow} \langle m'[e], \mu, k \circ K[([.] : \tau], N, s, \overline{a} \rangle\]  
(M-Marshout)

\[\langle m[e], \mu, k \circ [K[([.] : \tau], N, s, \overline{a} \rangle \stackrel{r}{\rightarrow} \langle m'[e], \mu, k \circ K[([.] : \tau], N, s, \overline{a} \rangle\]  
(M-Marshin)

\[\langle \text{wr}[e], \mu, k \circ [K[([.] : \tau], N, s, \overline{a} \rangle \stackrel{r}{\rightarrow} \langle \text{blk}[*], \theta, [K[([.] : \tau], N, s, \overline{a} \rangle\]  
(E-I)

\[\langle \text{blk}[*], \theta, [K[([.] : \tau], N, s, \overline{a} \rangle \stackrel{r}{\rightarrow} \langle \text{blk}[*], \theta, [K[([.] : \tau], N, s, \overline{a} \rangle\]  
(Done)

\[\langle \text{wr}[e], \mu, k \circ [K[([.] : \tau], N, s, \overline{a} \rangle \stackrel{r}{\rightarrow} \langle \text{blk}[*], \theta, [K[([.] : \tau], N, s, \overline{a} \rangle\]  
(M-Ret)

\[\langle \text{wr}[e], \mu, k \circ [K[([.] : \tau], N, s, \overline{a} \rangle \stackrel{r}{\rightarrow} \langle \text{blk}[*], \theta, [K[([.] : \tau], N, s, \overline{a} \rangle\]  
(M-RetT)

\[s \vdash \text{ unprotectedCode}(a_r)\text{ and }s \vdash \text{ unprotectedData}(a_d + |w|)\]  
\(\langle \text{wr}[e], \mu, k \circ [K[([.] : \tau], N, s, \overline{a} \rangle \stackrel{r}{\rightarrow} \langle \text{blk}[*], \theta, [K[([.] : \tau], N, s, \overline{a} \rangle\]  
(E-RetT)

\[s \vdash \text{ unprotectedCode}(a_r)\text{ or }s \vdash \text{ unprotectedData}(a_d + |w|)\]  

---

\[112\]
\[ \langle a_d | e | \mu, k | \langle \cdot : (\tau_1^{\ell_1_{\tau_1}}, \ldots, \tau_n^{\ell_n_{\tau_n}}) \rangle, s, \overline{a} \rangle \xrightarrow{\text{read}(a_d, \overline{w})} \] 
(M-ReadT)

\[ \langle w[e], \mu, k | \langle \cdot : (\tau_1^{\ell_1_{\tau_1}}, \ldots, \tau_n^{\ell_n_{\tau_n}}) \rangle, s, \overline{a} \rangle \] 
where \( \frac{\ell_1}{} N \rightarrow \frac{\ell_1}{} N \)
(E-ReadT)

\[ \langle a_d | e | \mu, k | \langle \cdot : (\tau_1^{\ell_1_{\tau_1}}, \ldots, \tau_n^{\ell_n_{\tau_n}}) \rangle, s, \overline{a} \rangle \xrightarrow{\text{blk}([\cdot], \emptyset, [\cdot], \cdot, s, \emptyset)} \] 
(E-Ret)

\[ \langle p[e], \mu, k, N, s, \overline{a} \rangle \xrightarrow{\text{blk}([\cdot], \emptyset, [\cdot], \cdot, s, \emptyset)} \] 
where \( s \vdash \text{returnback}(a) \) (E-AR)

\[ \langle p[e], \mu, k, N, s, \overline{a} \rangle \xrightarrow{\text{blk}([\cdot], \emptyset, [\cdot], \cdot, s, \emptyset)} \] 
where \( s \not\vdash \text{returnback}(a) \) (E-AC)

\[ \langle \text{call}(w[a_d], d) | e | \mu, k, N, s, \overline{a} \rangle \xrightarrow{\text{call}(w[a_d], d)} \] 
(A-Deref)

\[ \langle \text{call}(w[a_d], d), \mu, k | \text{ref}[\cdot] : (\tau_1^{\ell_1_{\tau_1}}, \ldots, \tau_n^{\ell_n_{\tau_n}}) \rangle \] 
where \( N(w_n) = (l_i[e'], \text{Ref} \tau) \) and \( s \vdash \text{derefEntryPoint}(a) \) (A-Apply)

\[ \langle \text{call}(w[a_d], d), \mu, k | \text{set}[\cdot] : (\tau_1^{\ell_1_{\tau_1}}, \ldots, \tau_n^{\ell_n_{\tau_n}}) \rangle \] 
where \( N(w_n) = (l_i[e'], \text{Ref} \tau) \) (A-Set)

\[ \langle \text{call}(w[a_d], d), \mu, k | \text{ref}[\cdot] : (\tau_1^{\ell_1_{\tau_1}}, \ldots, \tau_n^{\ell_n_{\tau_n}}) \rangle \] 
where \( N(w_n) = (l_i[e'], \text{Ref} \tau) \) and \( s \vdash \text{setEntryPoint}(a) \) (A-Ref)

\[ \langle \langle \tau_1^{\ell_1_{\tau_1}}, \ldots, \tau_n^{\ell_n_{\tau_n}} \rangle | e | \mu, k | \text{set}[\cdot] : (\tau_1^{\ell_1_{\tau_1}}, \ldots, \tau_n^{\ell_n_{\tau_n}}) \rangle \xrightarrow{\text{call}(w[a_d])} \] 
where \( N(w_n) = (l_i[e'], \text{Ref} \tau) \) and \( s \vdash \text{refEntryPoint}(a) \) (M-Call)

\[ \langle \langle \tau_1^{\ell_1_{\tau_1}}, \ldots, \tau_n^{\ell_n_{\tau_n}} \rangle | e | \mu, k | \text{set}[\cdot] : (\tau_1^{\ell_1_{\tau_1}}, \ldots, \tau_n^{\ell_n_{\tau_n}}) \rangle \xrightarrow{\text{call}(w[a_d])} \] 
where \( N(w_n) = (l_i[e'], \text{Ref} \tau) \) and \( s \vdash \text{unprotectedData}(a_d + |\overline{w}|) \) (M-CallW)

\[ \langle \langle \tau_1^{\ell_1_{\tau_1}}, \ldots, \tau_n^{\ell_n_{\tau_n}} \rangle | e | \mu, k | \text{set}[\cdot] : (\tau_1^{\ell_1_{\tau_1}}, \ldots, \tau_n^{\ell_n_{\tau_n}}) \rangle \xrightarrow{\text{call}(w[a_d])} \] 
where \( N(w_n) = (l_i[e'], \text{Ref} \tau) \) and \( s \not\vdash \text{unprotectedData}(a_d + |\overline{w}|) \) (E-CallW)

---

**Proof Sketch of Property 3.** Given a bisimilarity relation \( \approx^C \) over LTS\(^C\), we must define a compilation scheme \( \{ \cdot \}_C \) such that:

\[ \langle M_1, s_1, \overline{a_1} \rangle \approx^l \langle M_2, s_2, \overline{a_2} \rangle \iff \{ \langle M_1, s_1, \overline{a_1} \rangle \}_K \approx^C \{ \langle M_2, s_2, \overline{a_2} \rangle \}_K \]

113
The explicit substitutions introduced by the closure conversion do not interfere with the contextual equivalences of MiniML, as this internal implementation detail is never observed by the IoE-level attacker. The following compilation scheme \( \{ \cdot \}_C \):

\[
\{ ⟨c, μ, k, N, s, a⟩ \}_C = ⟨c[\star], μ, k, N, s, a⟩
\]

compiles states of \( M^K \) that are bisimilar in \( \approx^K \), and are thus derived from bisimilar states in \( \approx^l \), into states \( M^C \) that are bisimilar in \( \approx^C \) as the additional transitions of \( \text{LTS}^C \) are silent transitions \( \tau \rightarrow \) that are ignored by the weak bisimulations.

### 3. Refocusing and Transition Compression

The previous two transformations introduced various additional reduction steps into the semantics of the original LTS. In this third step, the previously derived \( \text{LTS}^C \) is transformed into a new \( \text{LTS}^R \) defined by the following triple.

\[
( M^C, L, \xrightarrow{L \rightarrow R})
\]

The state is unchanged from the previous \( \text{LTS}^C \), only the transition rules are modified. The new transition rules \( \xrightarrow{L \rightarrow R} \) are a refocused and compressed version of \( \xrightarrow{L \rightarrow C} \), obtained by following the methodology of Biernacka [12]. For example, M-B introduced in the previous transformation is optimized into the following:

\[
⟨v[e], μ, k \circ K[(\lambda x : τ'.e)[e'][\tau]] : N, s, a⟩ \xrightarrow{\tau} \quad \text{(M-B-O)}
\]

whereas the previous rule required the value to be plugged into the context before a reduction step could take place, this rule directly applies the value stored within continuation.

#### Machine Formalisation

The full formalisation is listed in the pages that follow. In this updated machine, the propagation (Prop-*) and plug rule (M-Plug) are removed. Instead they are replaced by smarter reduction rules that transition immediately based on the contents of the continuations.

Again the formalisation is split between defining the state, the internal reductions, the internal marshalling rules and and the observable interactions between the attacker and the machine.
\[ M^C \in \text{State}^R = \text{Ctrl} \times \text{Storage} \times \text{Kont} \times \text{Map} \times \text{Memory} \times \text{Returns} \]

\[ c^C \in \text{Ctrl} ::= \text{cl} | \text{blk}[e] | m[e] \]
\[ \text{cl} \in \text{Closure} ::= c[e] (\text{cl cl}) | \text{if cl}_1 \text{ cl}_2 \text{ cl}_3 \mid \text{let } x = \text{cl}_1 \text{ in } \text{cl}_2 \]
\[ \text{cl}_1 ; \text{cl}_2 \mid \text{cl}_1 \text{ op cl}_2 \mid \text{cl}_1 := \text{cl}_2 \mid \text{index cl} \mid \text{fix cl} \]
\[ \text{letrec } x : \tau = \text{cl}_1 \text{ in } \text{cl}_2 \mid \text{ref cl} \mid \text{cl}_1 \text{ cp cl}_2 \mid \text{cl}_i \]
\[ \langle \text{cl}_i^{\epsilon_1.\ldots.,\epsilon_n} \rangle \mid \text{exit cl} \]
\[ e \in \text{Express} ::= v \mid x \mid (e_1 \ e_2) \mid \langle e_i^{\epsilon_1.\ldots.,\epsilon_n} \rangle \mid e_1 \text{ cp e}_2 \]
\[ e_1 \text{ op e}_2 \mid e_i \mid \text{if e}_1 \ e_2 \ e_3 \mid \text{index e} \]
\[ \text{let } x = e_1 \text{ in } e_2 \mid \langle e \rangle \mid \text{ref e} \mid \text{exit e} \]
\[ \text{letrec } x : \tau = e_1 \text{ in } e_2 \mid e_1 ; e_2 \mid \text{fix e} \mid (\lambda x : \tau . e) \]
\[ v \in \text{Value} ::= \tau \rightarrow \tau \text{ Fu} \mid (\lambda x : \tau . e)[e] \mid b \mid \text{unit } l_i \pi \mid \langle v_i^{\epsilon_1.\ldots.,\epsilon_n} \rangle \]
\[ b \in \text{Bool} ::= \text{true} \mid \text{false} \]
\[ x \in \text{Var} \quad \text{a set of identifiers} \]
\[ \text{op} \in \text{Oper} ::= + \mid - \mid * \]
\[ \text{cp} \in \text{Comp} ::= < \mid \mid === \]
\[ \tau \in \text{Type} ::= \text{Bool} \mid \text{Int} \mid \text{Unit} \mid \tau_1 \rightarrow \tau_2 \mid \text{Ref } \tau \mid \langle \tau_i^{\epsilon_1.\ldots.,\epsilon_n} \rangle \]
\[ e \in \text{Env} = \ast \mid e \cdot (\text{cl}/x) \]
\[ k \in \text{Kont} ::= [] \mid k[K : \tau \rightarrow \tau'] \mid k[\circ K : \tau] \mid k[\triangleleft K : \tau] \]
\[ \mid k[\triangleright K : \tau] \]
\[ K \in \text{Kin} ::= [] \mid K([[] \text{ cl}]) \mid K[v []] \mid K([[] \text{ op cl}_2]) \mid K([v_1 \text{ op } []]) \]
\[ K([[] \text{ cp cl}_2]) \mid K([v_1 \text{ op } []]) \mid K([\langle v_i^{\epsilon_1.\ldots.,\epsilon_n} \rangle \text{ cl}_{k+1}]) \]
\[ K(\text{index } []) \mid K([\text{let } x = [] \text{ in } \text{cl}_2]) \mid K([!![]]) \]
\[ K([[]] i) \mid K([\text{fix } []]) \mid K([[] := \text{cl}_2]) \mid K([v := [], i]) \]
\[ K([\text{if } [] \text{ cl}_2 \text{ cl}_3]) \mid K([\text{exit } []]) \mid K([\text{ref } []]) \mid K([[] ; \text{cl}_2]) \]
\[ n_i \in \text{Name} \quad \text{an enumerable set} \]
\[ N \in \text{Map} = \text{Name} \rightarrow (\text{Ctrl} \times \text{Type}) \]
\[ s \in \text{Memory} \quad \text{a descriptor of low-level memory} \]
\[ \alpha \in \text{Returns} \quad \text{a stack of low-level addresses} \]

\[
\langle e_1 e_2 \rangle^e, \mu, k[\circ K : \tau], N, s, \alpha \xrightarrow{\text{M-V}} \langle e_1 e_2 \rangle^e, \mu, k[\circ K[\cdot e_2[e]] : \tau], N, s, \alpha \]

\[
\langle v e_1 \rangle^e, \mu, k[\circ K[\cdot e'[e]] : \tau], N, s, \alpha \xrightarrow{\text{M-R1}} \langle e'[e'], \mu, k[\circ K[\cdot e] : \tau], N, s, \alpha \]

\[
\langle v e_1 \rangle^e, \mu, k[\circ K[\cdot e'[e]/x)], \mu, k[\circ K : \tau], N, s, \alpha \xrightarrow{\text{M-B}}
\]

\[
\langle v e_1 \rangle^e, \mu, k[\circ K[\cdot e'[e]/x)], \mu, k[\circ K : \tau], N, s, \alpha \xrightarrow{\text{M-R3}}
\]

\[
\langle e_1 e_2 e_3 \rangle^e, \mu, k[\circ K : \tau], N, s, \alpha \xrightarrow{\text{M-R3}} \langle e_1 e_2 e_3 \rangle^e, \mu, k[\circ K[\cdot e_2 \circ e_3[e]] : \tau], N, s, \alpha
\]
⟨true[e], µ, k o K![if [·] e2[e'] e3[e']]] : τ], N, s, a⟩ →
⟨e2[e'], µ, k o K : τ], N, s, a⟩ (M-IfT)

⟨false[e], µ, k o K![if [·] e2[e'] e3[e']]] : τ], N, s, a⟩ →
⟨e3[e'], µ, k o K : τ], N, s, a⟩ (M-IfF)

⟨(let x = e1 in e2)[e], µ, k o K : τ], N, s, a⟩ →
⟨e1[e], µ, k o K[(let x = [·] in e2)[e]] : τ], N, s, a⟩ (M-R4)

⟨v[e], µ, k o K[(let x = [·] in e2[e'])] : τ], N, s, a⟩ →
⟨e2[e'], (v[e']/x)], µ, k o K : τ], N, s, a⟩ (M-Let)

⟨(refe)[e], µ, k o K : τ], N, s, a⟩ → ⟨e[e], µ, k o K[(ref[·]) : τ], N, s, a⟩ (M-Ra)

⟨v[e], µ, k o K[(ref[·]) : τ]], N, s, a⟩ → ⟨l[e], µ, k o K : τ], N, s, a⟩ (M-Ref)

⟨(le)[e], µ, k o K : τ], N, s, a⟩ → ⟨e[e], µ, k o K[(le[·]) : τ], N, s, a⟩ (M-Rd)

⟨l[e], µ, k o K[(le[·])] : τ], N, s, a⟩ → ⟨v[e'], µ, k o K : τ], N, s, a⟩ (M-Der)

where µ(l1) = v[e']

⟨(index)[e], µ, k o K : τ], N, s, a⟩ →
⟨e[e], µ, k o K[(index[·])] : τ], N, s, a⟩ (M-Ri)

⟨l[e], µ, k o K[(index[·])] : τ], N, s, a⟩ → ⟨l[e], µ, k o K : τ], N, s, a⟩ (M-Index)

⟨(exit)[e], µ, k o K : τ], N, s, a⟩ →
⟨e[e], µ, k o K[(exit[·])] : τ], N, s, a⟩ (M-Re)

⟨v[e], µ, k o K[(exit[·])] : τ], N, s, a⟩ → ⟨l[e], µ, k o K : τ], N, s, a⟩ (M-Exit)

⟨e1 ; e2[e], µ, k o K : τ], N, s, a⟩ →
⟨e1[e], µ, k o K[(e1 ; e2)[e]] : τ], N, s, a⟩ (M-RSq)

⟨unit[e], µ, k o K[(e1 ; e2)[e']]] : τ], N, s, a⟩ →
⟨e2[e'], µ, k o K : τ], N, s, a⟩ (M-Seq)

⟨e1 := e2[e], µ, k o K : τ], N, s, a⟩ →
⟨e1[e], µ, k o K[(e1 := e2)[e]] : τ], N, s, a⟩ (M-Rs)

⟨l[e], µ, k o K[(e1 := e2)[e']] : τ], N, s, a⟩ →
⟨e2[e'], µ, k o K[(l[e] := [e']) : τ], N, s, a]⟩ (M-Set)

⟨v[e], µ, k o K[(l[e] := [e'])]] : τ], N, s, a⟩ →
⟨unit[e], µ, l, i → v[e]], k o K : τ], N, s, a⟩ (M-Set2)

⟨(fix)[e], µ, k o K : τ], N, s, a⟩ → ⟨e[e], µ, k o K[(fix[·]) : τ], N, s, a]⟩ (M-Rf)

⟨(λx : τ', e')[e'], µ, k o K[(fix[·]) : τ], N, s, a]⟩ →
⟨e'[e' · ([fix(λx : τ', e')][x])] · µ, k o K : τ], N, s, a]⟩ (M-Fix)

⟨(e1 op e2)[e], µ, k o K : τ], N, s, a]⟩ →
⟨e1[e], µ, k o K[(e1 op e2)[e]] : τ], N, s, a]⟩ (M-Ro)

⟨v[e], µ, k o K[(e1 op e2)[e']] : τ], N, s, a]⟩ →
⟨e2[e'], µ, k o K[(vop[·]) : τ], N, s, a]⟩ (M-Op)

⟨v[e], µ, k o K[(v1 op[·])] : τ], N, s, a]⟩ →
⟨(v'[e'], µ, k o K : τ], N, s, a]⟩ (M-Op2)

⟨(e1 cp e2)[e], µ, k o K : τ], N, s, a]⟩ →
⟨e1[e], µ, k o K[(e1 cp e2)[e]] : τ], N, s, a]⟩ (M-Rc)
\begin{align*}
\langle v[e], \mu, k \circ K[((] cp e_2[e']) : \tau] \rangle, N, s, \overline{a} \xrightarrow{\tau} & \quad (M-Cp) \\
\langle e_2[e'], \mu, k \circ K([v cp []) : \tau], s, N, \overline{a} \rangle \quad (M-Cp2)
\end{align*}

\begin{align*}
\langle (e.i)[e], \mu, k \circ K : \tau], N, s, \overline{a} \rangle \xrightarrow{\tau} & \langle e[e], \mu, k \circ K[((] : \tau] : \tau], N, s, \overline{a} \rangle \quad (M-Rproj)
\langle (c_i^{\leq 1..n}) \rangle, \mu, k \circ K : \tau], N, s, \overline{a} \rangle \xrightarrow{\tau} & \langle c_i, \mu, k \circ K : \tau], N, s, \overline{a} \rangle \quad (M-Proj)
\langle v[e], \mu, k \circ [\vdash : \tau], N, s, \overline{a} \rangle \xrightarrow{\tau} & \langle v[e], \mu, k \circ [\vdash : \tau], N, s, \overline{a} \rangle \quad (M-Done)
\end{align*}

\begin{align*}
\langle m[e], \mu, k \circ [\vdash : \tau], N, s, \overline{a} \rangle \xrightarrow{\tau} & \langle m'[e], \mu, k, N', s, \overline{a} \rangle \quad \text{where } \frac{\Downarrow m[N]}{\Downarrow m'[N']}
\langle m[e], \mu, k \circ \langle : \tau], N, s, \overline{a} \rangle \xrightarrow{\tau} & \langle m'[e], \mu, k \circ [\vdash : \tau], N, s, \overline{a} \rangle \quad \text{where } \frac{\Downarrow m[N]}{\Downarrow m'[N']}
\end{align*}

\begin{align*}
\langle \text{wr}[e], \mu, k \circ \langle : \tau], N, s, \overline{a} \rangle \xrightarrow{\text{E-I}} & \langle \text{blk}[\star], \emptyset, [], \star, s, \theta \rangle \quad \text{(E-I)}
\langle \text{blk}[\star], \emptyset, [], \star, s, \theta \rangle \xrightarrow{\text{E-RetT}} & \langle \text{blk}[\star], \emptyset, [], \star, s, \theta \rangle \quad \text{(Done)}
\langle w[e], \mu, k \circ [\vdash : \tau], N, s, a_r : \overline{a} \rangle \xrightarrow{\text{ret } a_r, \overline{a}} & \langle \text{blk}[\star], \mu, k, N, s, \overline{a} \rangle \quad \text{(M-Ret)}
\langle \text{blk}[\star], \mu, k, N, s, \overline{a} \rangle \xrightarrow{\text{write}(a_d, \overline{w}), \text{ret } a_r} & \langle \text{blk}[\star], \mu, k, N, s, \overline{a} \rangle \quad \text{(M-RetT)}
\end{align*}

\begin{align*}
\langle a_d[e], \mu, k \circ [\vdash : \tau], N, s, \overline{a} \rangle \xrightarrow{\text{read}(a_d, \overline{w})} & \langle \text{blk}[\star], \emptyset, [], \star, s, \theta \rangle \quad \text{(M-ReadT)}
\langle \text{blk}[\star], \emptyset, [], \star, s, \theta \rangle \xrightarrow{\text{E-RetT}} & \langle \text{blk}[\star], \emptyset, [], \star, s, \theta \rangle \quad \text{(E-RetT)}
\langle \text{blk}[\star], \emptyset, [], \star, s, \theta \rangle \xrightarrow{\text{A-Ret}} & \langle \text{blk}[\star], \emptyset, [], \star, s, \theta \rangle \quad \text{(A-Ret)}
\end{align*}

\begin{align*}
\langle \text{blk}[\star], \emptyset, [], \star, s, \theta \rangle \xrightarrow{\text{E-AR}} & \langle \text{blk}[\star], \emptyset, [], \star, s, \theta \rangle \quad \text{(E-AR)}
\langle \text{blk}[\star], \emptyset, [], \star, s, \theta \rangle \xrightarrow{\text{E-AC}} & \langle \text{blk}[\star], \emptyset, [], \star, s, \theta \rangle \quad \text{(E-AC)}
\end{align*}

\begin{align*}
\langle \text{blk}[\star], \emptyset, [], \star, s, \theta \rangle \xrightarrow{\text{A-Deref}} & \langle \text{blk}[\star], \emptyset, [], \star, s, \theta \rangle \quad \text{(A-Deref)}
\end{align*}
where the state \(\text{LTS}\) of \(\text{AM}\) is the following quintuple.

\[
\langle c, e, k, N, \overline{a} \rangle
\]

The control \(c\) and substitution map \(e\) are obtained by unfolding the closures \(cl\) of \(\text{LTS}^R\) and by removing the map of substitutions of the loE-level words and halted state. Note that even after the unfolding, closures remain values used to encode \(\lambda\)-terms: the application of closures cannot be unfolded.

Another result of the unfolding is that the map of substitutions no longer stores closures but values.

4. Unfolding Closures

The previously derived \(\text{LTS}^R\) is transformed into a CESK machine defined by the triple

\[
\langle M^{AM}, L, \rightarrow^{AM} \rangle
\]

where the state \(M^{AM}\) is the following quintuple.

\[
\langle e, e, k, N, \overline{a} \rangle
\]
\[ e ::= * \mid e(v/x). \]

**Machine Formalisation** The formalisation of the machine is listed below. The most notable change here, besides the separate environment substate, is the change in the continuations. The continuations now store an explicit environment \( e \), to return to when control is returned to the continuation.

Because \( \lambda \)-terms are still expressions and the closure of a \( \lambda \)-term a value, we include a new rule (M-PrL) that propagates the map of substitutions \( e \) to make a closure out of a \( \lambda \)-term.

Again the formalisation is split between a definition the machine state, the internal reduction rules, the internal marshalling rules and and the observable interactions between the attacker and the machine.

\[
M \in \text{State}^{AM} = \text{Ctrl} \times \text{Env} \times \text{Storage} \times \text{Kont} \times \text{Map} \times \text{Memory} \times \text{Returns}
\]

\[
e^C \in \text{Ctrl} ::= e \mid \text{blk} \mid m
\]

\[
e \in \text{Express} ::= v \mid x \mid (e_1 \ e_2) \mid \langle e_i^{i \in 1..n} \rangle \mid e_1 \ \text{cp} \ e_2
\]

\[
| e_1 \ \text{op} \ e_2 \mid e \ i \mid \text{if} \ e_1 \ e_2 \ e_3 \mid \text{index} \ e
\]

\[
| \text{let} \ x = e_1 \ \text{in} \ e_2 \mid !e \mid \text{ref} \ e \mid \text{exit} \ e
\]

\[
| \text{letrec} \ x : \tau = e_1 \ \text{in} \ e_2 \mid e_1 ; e_2 \mid \text{fix} \ e \mid (\lambda x : \tau.e)
\]

\[
v \in \text{Value} ::= \tau \rightarrow \tau' Fp \mid (\lambda x : \tau.e)[e] \mid b \mid \text{unit} \mid l_i \mid \bar{n} \mid \langle v_i^{i \in 1..n} \rangle
\]

\[
b \in \text{Bool} ::= \text{true} \mid \text{false}
\]

\[
x \in \text{Var} = \text{a set of identifiers}.
\]

\[
op \in \text{Oper} ::= + \mid - \mid *
\]

\[
cp \in \text{Comp} ::= < \mid > \mid ==
\]

\[
\tau \in \text{Type} ::= \text{Bool} \mid \text{Int} \mid \text{Unit} \mid \tau_1 \rightarrow \tau_2 \mid \text{Ref} \tau \mid \langle \tau_i^{i \in 1..n} \rangle
\]

\[
e \in \text{Env} ::= * \mid e \cdot (v/x)
\]

\[
k \in \text{Kont} ::= [:] \mid k[K : \tau \rightarrow \tau'] \mid k[\circ \ K : \tau] \mid k[\leftarrow K : \tau]
\]

\[
| k[\times \ K : \tau]
\]

\[
K \in \text{Kin} ::= [:] \mid K[(\cdot : e)[e]] \mid K[v \cdot] \mid K[(\cdot : \text{op} e)[e]] \mid K[(v_1 \text{op} \cdot)[e]]
\]

\[
| K[(\cdot : \text{cp} e)[e]] \mid K[(v_1 \text{cp} \cdot)[e]] \mid K[(v_1^{i \in 1..j} : \cdot, e_1^{i \in k \in 1..n})[e]]
\]

\[
| K[(\text{index}[:]]) \mid K[(\text{let} \ x = [:] \ \text{in} \ e)[e]] \mid K[(:: : e_2)[e]]
\]

\[
| K[(:: i)] \mid K[(\text{fix} [:])]) \mid K[(:: : := e_2)[e]] \mid K[(v :: [:])]
\]

\[
| K[(\cdot : e_2 e_3)[e]] \mid K[(\text{exit}[:])]) \mid K[(\text{ref}[:])]) \mid K[(\text{!}[:])])
\]

\[
n_1 \in \text{Name} = \text{an enumerable set}.
\]

\[
\text{N} \in \text{Map} = \text{Name} \rightarrow (\text{Ctrl} \times \text{Type})
\]

\[
s \in \text{Memory} = \text{a descriptor of low-level memory}
\]

\[
\bar{\pi} \in \text{Returns} = \text{a stack of low-level addresses}
\]

119
\( \langle x, e, \mu, k, N, s, \overline{a} \rangle \xrightarrow{\tau} \langle v, e, \mu, k, N, s, \overline{a} \rangle \) where \( e(x) = v \) (M-V)

\( \langle \lambda x : \tau.e \rangle, e, \mu, k, N, s, \overline{a} \xrightarrow{\tau} \langle \lambda x : \tau.e \rangle[e], e, \mu, k, N, s, \overline{a} \) (M-PrL)

\( \langle e_1 e_2 \rangle, e, \mu, k, O K : \tau, N, s, \overline{a} \xrightarrow{\tau} \)

\( \langle e_1, e_2, e_3 \rangle, e, \mu, k, O K : \tau, N, s, \overline{a} \xrightarrow{\tau} \langle e, e', \mu, k \circ O K[v : \tau], N, s, \overline{a} \rangle \) (M-R2)

\( \langle v, e, \mu, k \circ O K(\lambda x : \tau.e)[e'] : \tau, N, s, \overline{a} \xrightarrow{\tau} \)

\( \langle e, e', \mu, k \circ O K[v : \tau], N, s, \overline{a} \rangle \) (M-B)

\( \langle \text{let } x = e_1 \text{ in } e_2, e, \mu, k \circ O K : \tau, N, s, \overline{a} \xrightarrow{\tau} \)

\( \langle e, e', \mu, k \circ O K : \tau, N, s, \overline{a} \xrightarrow{\tau} \langle e, e', (v/x), \mu, k \circ O K : \tau, N, s, \overline{a} \rangle \) (M-Let)

\( \langle \text{let } x = [ ] \text{ in } e_2, e, \mu, k \circ O K : \tau, N, s, \overline{a} \xrightarrow{\tau} \)

\( \langle e, e', \mu, k \circ O K : \tau, N, s, \overline{a} \xrightarrow{\tau} \langle v, e, \mu, k \circ O K : \tau, N, s, \overline{a} \rangle \) (M-R4)

\( \langle v, e, \mu, k \circ O K([\text{ref} : ])[e] : \tau, N, s, \overline{a} \xrightarrow{\tau} \)

\( \langle l_i, e, (\mu, l_i \mapsto v), k \circ O K : \tau, N, s, \overline{a} \rangle \) (M-Der)

\( \langle i.e, e, \mu, k \circ O K : \tau, N, s, \overline{a} \xrightarrow{\tau} \langle e, e, \mu, k \circ O K[l_i : ] : \tau, N, s, \overline{a} \rangle \) (M-Rd)

\( \langle l_i, e, \mu, k \circ O K([i : ])[e] : \tau, N, s, \overline{a} \xrightarrow{\tau} \langle v, e, \mu, k \circ O K : \tau, N, s, \overline{a} \rangle \) (M-Index)

\( \langle \lambda x : \tau.e \rangle, e, \mu, k, O K : \tau, N, s, \overline{a} \xrightarrow{\tau} \langle \lambda x : \tau.e \rangle[e], e, \mu, k, O K : \tau, N, s, \overline{a} \rangle \) (M-Rsq)

\( \langle \text{let } x = [ ] \text{ in } e_2, e, \mu, k \circ O K : \tau, N, s, \overline{a} \xrightarrow{\tau} \langle e, e', (v/x), \mu, k \circ O K : \tau, N, s, \overline{a} \rangle \) (M-Set2)
\[ \langle \text{fix}e \rangle, e, \mu, k[ \circ K : [\tau], N, s, \overline{a} ] \xrightarrow{r} \langle e, e, \mu, k[ \circ K[\langle \text{fix} \rangle] : [\tau], N, s, \overline{a} \rangle \quad \text{(M-Rf)} \]

\[ \langle \lambda x : [\tau']e'[x], e, \mu, k[ \circ K[\langle \text{fix} \rangle] : [\tau], N, s, \overline{a} \rangle \xrightarrow{r} \langle e'[x], \langle e'[x] \circ (\text{fix})(\lambda x : [\tau']e'[x]) \rangle, \mu, k[ \circ K : [\tau], N, s, \overline{a} \rangle \quad \text{(M-Fix)} \]

\[ \langle e_1 \text{op}_2, e, \mu, k[ \circ K : [\tau], N, s, \overline{a} \rangle \xrightarrow{r} \langle e_1, e, \mu, k[ \circ K[[\text{op}_2]e[ : [\tau], N, s, \overline{a} \rangle \quad \text{(M-Ro)} \]

\[ \langle v, e, \mu, k[ \circ K[[\text{op}_2]e'[ : [\tau], N, s, \overline{a} \rangle \xrightarrow{r} \langle v', e, \mu, k[ \circ K : [\tau], N, s, \overline{a} \rangle \quad \text{(M-Op)} \]

\[ \langle e_1 \text{cp}_2, e, \mu, k[ \circ K : [\tau], N, s, \overline{a} \rangle \xrightarrow{r} \langle e_1, e, \mu, k[ \circ K[[\text{cp}_2]e[ : [\tau], N, s, \overline{a} \rangle \quad \text{(M-Cp)} \]

\[ \langle v, e, \mu, k[ \circ K[[\text{cp}_2]e'[ : [\tau], N, s, \overline{a} \rangle \xrightarrow{r} \langle v', e, \mu, k[ \circ K : [\tau], N, s, \overline{a} \rangle \quad \text{(M-Cp2)} \]

\[ \langle (e.i), e, \mu, k[ \circ K : [\tau], N, s, \overline{a} \rangle \xrightarrow{r} \langle e, e, \mu, k[ \circ K[[\text{cp}_2]e'[ : [\tau], N, s, \overline{a} \rangle \quad \text{(M-Rproj)} \]

\[ \langle v_i[i \in [1..n]], e, \mu, k[ \circ K[[\text{cp}_2]e'[ : [\tau], N, s, \overline{a} \rangle \xrightarrow{r} \langle v_i, e, \mu, k[ \circ K : [\tau], N, s, \overline{a} \rangle \quad \text{(M-Proj)} \]

\[ \langle m, e, \mu, k[ \circ [\cdot] : [\tau], N, s, \overline{a} \rangle \xrightarrow{r} \langle m', e, \mu, k, N', s, \overline{a} \rangle \quad \text{where } \frac{\| m \|^N}{\| m' \|^N'} \quad \text{(M-Marshout)} \]

\[ \langle m, e, \mu, k[ \circ [\cdot] : [\tau], N, s, \overline{a} \rangle \xrightarrow{r} \langle m', e, \mu, k, N', s, \overline{a} \rangle \quad \text{where } \frac{\| m \|^N}{\| m' \|^N'} \quad \text{(M-Marshin)} \]

\[ \langle \text{wr}, e, \mu, k[ \circ K : [\tau], N, s, \overline{a} \rangle \xrightarrow{r} \langle \text{blk}, \ast, \emptyset, [\cdot], \ast, s, \emptyset \rangle \quad \text{(E-I)} \]

\[ \langle \text{blk}, \ast, \emptyset, [\cdot], \ast, s, \emptyset \rangle \xrightarrow{r} \langle \text{blk}, \ast, \emptyset, [\cdot], \ast, s, \emptyset \rangle \quad \text{(Done)} \]

\[ \langle w, e, \mu, k[ \circ [\cdot] : [\tau], N, s, a_r : \overline{a} \rangle \xrightarrow{r} \langle \text{blk}[e], \mu, k, N, s, \overline{a} \rangle \quad \text{(M-Ret)} \]

\[ \langle \text{write}(a_d, w), \text{ret } a_r \rangle \xrightarrow{r} \langle \text{blk}, e, \mu, k, N, s, \overline{a} \rangle \quad \text{(M-RetT)} \]

\[ s \vdash \text{unprotectedCode}(a_r) \text{ and } s \vdash \text{unprotectedData}(a_d + |\overline{w}|) \]

\[ \langle \text{write}(a_d, w), \text{ret } a_r \rangle \xrightarrow{r} \langle \text{blk}, e, \mu, k, N, s, \overline{a} \rangle \quad \text{(E-RetT)} \]

\[ s \not\vdash \text{unprotectedCode}(a_r) \text{ or } s \not\vdash \text{unprotectedData}(a_d + |\overline{w}|) \]

\[ \langle a_d, e, \mu, k[ \circ [\cdot] : [\tau_i[i \in [1..n]], s, \overline{a} \rangle \xrightarrow{r} \langle \text{read}(a_d, w), \text{blk}, e, \mu, k, \circ [\cdot] : [\tau_i[i \in [1..n]], s, \overline{a} \rangle \text{ where } \frac{\| a_d \|^N}{\| a_d \|^N} \quad \text{(M-ReadT)} \]

\[ \langle \text{write}(a_d, w), \text{ret } a_r \rangle \xrightarrow{r} \langle \text{blk}, e, \mu, k, \circ [\cdot] : [\tau_i[i \in [1..n]], s, \overline{a} \rangle \text{ where } \frac{\| a_d \|^N}{\| a_d \|^N} \quad \text{(E-ReadT)} \]
(blk, e, μ, k, N, s, a) \xrightarrow{ret a(\tau)} \langle w, e, \mu, k|<:\tau], N, s, a \rangle \quad \text{(A-Ret)}

where s \vdash \text{returnback}(a)

(\textbf{Proof Sketch of Property 3.}) Given a bisimilarity relation \(\approx^{AM}\) over the CESK machine (The CESK machine is also an LTS), we must define a compilation scheme \{\cdot\}_{AM} such that:

\[
\langle M_1, s_1, \overline{a_1} \rangle \approx^I \langle M_2, s_2, \overline{a_2} \rangle \iff \{\{\langle M_1, s_1, \overline{a_1} \rangle\}_C\}_{AM} \approx^{AM} \{\{\langle M_2, s_2, \overline{a_2} \rangle\}_C\}_{AM}
\]

The following compilation scheme \{\cdot\}_{AM}:

\[
\{\langle e[e], \mu, k, N, a \rangle\}_{AM} = \langle e, \mu, k, N, a \rangle
\]

\[
\{\langle w[e], \mu, k, N, a \rangle\}_{AM} = \langle w, e, \mu, k, N, a \rangle
\]

\[
\{\langle blk[e], \mu, k, N, a \rangle\}_{AM} = \langle blk, e, \mu, k, N, a \rangle
\]

\]

122
compiles states $M^R$ that are bisimilar in $\approx^R$ into states $M^{AM}$ that are bisimilar in $\approx^{AM}$ as the transitions are unmodified except for the changed structure of the continuations and the added (M-PrL) rule that is a silent transition.

5.3.2 Step 4. Implementation

The derived CESK machine has been implemented in C (available online\textsuperscript{2}) and deployed to the Fides implementation of PMA [80]. As explained previously in Section 6.5, Fides implements PMA through use of a hypervisor that runs two virtual machines: one that handles the secure memory module and one that handles the outside memory. This approach produces a lot of overhead every time control flow switches between the abstract machine and the loE-level context.

\textbf{Figure 5.2.} The CESK machine resides within the protected memory. The loE-level attacker can only interact with the program running on the machine through the entry points.

Our C implementation of the CESK machine is compiled into the protected memory of the PMA mechanism, as illustrated in Figure 5.2. The machine state $M^{AM}$ as well as the run-time stack and heap are placed in the protected data section. This restricts access to the state to the transition rules $\xrightarrow{L}^{AM}$ of the CESK that reside in the protected code section. The entry points defined in the set of entry points $\bar{ep}$ become the entry points to the abstract machine. Their implementation is simple, they simply defer control to the appropriate transition rule in the code section.

\textsuperscript{2}https://github.com/sylvarant/secure-abstract-machine

123
Note that in our current implementation the MiniML program to be executed is loaded into the protected memory module at compile time. A possible extension to this work is thus to extend the implemented CESK with the secure authentication capabilities of Fides to enable trusted third parties to upload MiniML programs to the CESK machine.

**Performance**

To get an indication of the performance overhead of our secure implementation, we have benchmarked the overhead produced by our implementation for three scenarios. In the first scenario (*Application*) the loE-level attacker applies a secure MiniML function to a boolean value. In the second scenario (*Callback*) the attacker applies a higher-order secure function to an attacker function, triggering a callback. In the third scenario (*Read*) the attacker dereferences a shared location.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>CESK</th>
<th>CESK + PMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>0.28 µs</td>
<td>16.71 µs</td>
</tr>
<tr>
<td>Callback</td>
<td>0.33 µs</td>
<td>40.92 µs</td>
</tr>
<tr>
<td>Read</td>
<td>0.24 µs</td>
<td>16.88 µs</td>
</tr>
</tbody>
</table>

*Table 5.5. The performance of the secure abstract machine for three security relevant scenarios, without PMA and with PMA enabled.*

The tests were performed on a Dell Latitude with a 2.67 GHz Intel Core i5 and 4GB of DDR3 RAM. As listed in Table 5.5, the Callback scenario incurs high overheads due to it involving more than one transition between the secure and insecure memory. However, in real-world programs the interactions between the abstract machine and the outside context will only represent a small part of the computations, limiting the impact of the secure abstract machine on overall performance.

### 5.3.3 Validation

As explained in Section 5.2.5, each step of the methodology specifies a property whose proof assures the correctness of the applied step. Combined these properties form a chain of bi-implications (Figure 5.1), that ensure that the contextual equivalences of the source language are preserved and reflected in obtained abstract machine implementation.

The chain of bi-implications denoted in Figure 5.3, establishes the fact that the contextual equivalence of MiniML is preserved and reflected in the derived CESK machine through four links.

The first link establishes that the first step of the methodology: securely developing a FFI to the loE-level attacker, was done correctly for our MiniML+ FFI formalisation by means of the proof for Theorem 5 in Section 3.5.3.
Theorem 5

\[ \text{MiniML} : \quad e_1 \simeq e_2 \]

Theorem 6

\[ \text{MiniML}^+ : \quad \{ e_1 \}^+ \simeq^+ \{ e_2 \}^+ \]

Property 3

\[ \langle \{ e_1 \}^+, \emptyset \rangle \approx AM \langle \{ e_2 \}^+, \emptyset \rangle \]

Fides [80]

\[ \text{A+I:} \quad \{ \langle \{ e_1 \}^+, \emptyset \rangle \}_{\text{CESK}} A+I \simeq a \{ \langle \{ e_2 \}^+, \emptyset \rangle \}_{\text{CESK}} A+I \]

Figure 5.3. Our derived CESK machine reflects and preserves the equivalences of MiniML through a 4 link long chain of bi-implications.

The second link establishes that the second step of the methodology: deriving a LTS formalisation of the previously derived FFI, was done correctly for the LTS of Section 4.2 by means of the proof for Theorem 6 in Section 4.3.

The third link establishes that the third step of the methodology: deriving the abstract machine formalisation, was done correctly in Section 5.3.1 through the proof sketches of Property 3 given in that section. Note that the translation scheme \{\cdot\}_{\text{CESK}} combines all translation schemes of Section 5.3.1.

\[ \{\cdot\}_{\text{CESK}} = \{\{\cdot\}_{K}\}_{AM} \]

The fourth and final link establishes that the fourth step of the methodology: implementing the abstract machine formalisation to run on top of the PMA mechanism, excluding human coding errors, was done correctly. This follows from the fact that we employed the secure compiler of the Fides Architecture to compile our C implementation to the protected memory of PMA.

5.4 Summary

This chapter presented the implementation of a secure CESK machine for MiniML. The CESK machine is made secure by applying the IoE-level memory isolation mechanism (PMA) and by following a methodology that applies Biernacka et al.’s syntactic correspondence to the secure foreign function interface of Chapter refchffi and Chapter 4. A concatenation of formal properties for each step of the methodology ensures that the result is secure.

There are different directions for future work. One is to investigate different abstract machine implementations and what changes (if any) must be done to the methodology to scale to them. Another direction is the integration of a
secure abstract machine with runtime aspects of advanced programming languages such as, for example, garbage collection. Two challenges arise in this setting: implementing secure garbage collection and proving that this garbage collection does not introduce security leaks.
6. A Secure Compiler for ML Modules

This chapter presents a compilation scheme that compiles ModuleML (Section 2.2) into A+I: untyped assembly language running on a machine model enhanced with PMA (Section 2.7). The compilation scheme compiles an input ModuleML module to the protected memory of PMA in a way that protects the result from low-level attackers while at the same time preserving all of its functionality.

This chapter is not the first work to securely compile to untyped assembly extended with PMA. Previous work on secure compilation by Patrignani et al. [66] has securely compiled an object-oriented language to PMA. The secure compilation scheme introduced in this chapter differs from that work in the following three ways. Firstly, the secure compilation scheme of Patrignani et al. is limited in its usefulness as a real world compilation scheme in that it does not accept any arguments from the attacker outside of basic values, such as integers and booleans, and shared object identities. In this work we develop a more realistic compiler that accepts attacker defined functions, locations and modules.

Secondly, the abstractions of functional languages are more challenging than those of imperative object-oriented languages. In a functional language such as ModuleML functions are, for example, higher-order and thus cannot be compiled into a straight-forward sequence of calls and returns. In our work we address these challenges through by using the interaction counting masking mechanism of Chapter 3.

Lastly, the inclusion of functors, higher-order functions mapping modules to modules, in ModuleML presents a novel secure compilation challenge. The modules created through functors are not analogous to objects created through constructors from a secure compilation standpoint. Whereas every object produced by a constructor is of the same type and thus subject to the same type checks and security constraints, functors can produce modules of different types that require different type checks and security constraints. In this work we investigate all of the security challenges introduced by functors and develop an efficient method of encoding the required checks.

Our compilation scheme is proven to reflect contextual equivalence. As is common in secure compilation works that target a realistic low-level target language [66], we assume that the preservation of contextual equivalence holds. Preservation coincides with compiler correctness, it establishes that the secure compiler is a correct ModuleML compiler. While we have tested our implementation of the compilation scheme intensely, we consider formally verifying the implementation of the compiler a separate research subject.
First this chapter details ModuleML contextual equivalence and its relation to security properties (Section 6.1). Next to better explain the secure compilation scheme, this chapter introduces a pattern referred to as the Secure Abstract Data Type pattern (Section 6.2). This pattern bundles together some of the techniques applied in previous secure compilation and full abstraction works. Subsequently it details the compilation scheme (Section 6.3) and proves that it reflects contextual equivalence (Section 6.4). Finally we present the implementation and performance of the compilation scheme (Section 6.5).

6.1 Contextual Equivalence for ModuleML

The secure compilation scheme aims to reflect ModuleML contextual equivalence in the target language $A+I$. As defined in Definition 3, a ModuleML context $C : M' \rightarrow M$ is a well-typed program $P$ of type $M$ with a single hole $\cdot$ that is to be filled with a module $M$ of type $M'$. Two ModuleML modules $M_1$ and $M_2$ are contextually equivalent if and only if there is no context $C$ that can distinguish them. Contextual equivalence for ModuleML is formalised as follows.

**Definition 16 (Contextual Equivalence for ModuleML)**

\[
M_1 \simeq^{MM} M_2 \overset{def}{=} \forall C : M' \rightarrow M. \ C[M_1]\uparrow \iff C[M_2]\uparrow
\]

where $\uparrow$ indicates divergence.

6.1.1 Examples of Contextual Equivalence

The following two ModuleML modules $M_1$ and $M_2$ are, for example, not contextually equivalent as they are distinguishable by the denoted context $C$.

| module $M$ = struct | open $M$
| val $v_1$ = ref $\bar{\top}$ | (if !(M.$v_1$) == $\bar{0}$) then $\Omega$
| end | else true)

The Module $M_1$

The Context $C$

Where $\Omega$ is a diverging term and the open $M$ statement includes either the module $M_1$ or $M_2$ into $C$, implementing the hole of the context.

Contextual equivalence is a means of capturing security properties such as confidentiality and and integrity. We reconsider the examples of Section 2.3.2 for the modules of ModuleML.
**Confidentiality**

Consider the following two contextually equivalent ModuleML modules $M_1$ and $M_2$ that uphold the signature $S$.

\[
\begin{align*}
\text{module } M &= \text{struct} \\
&\text{type } T = \text{Ref } \text{Int} \\
&\text{val } v_1 = \text{ref } \overline{1} \\
&\text{end} : S \\
\end{align*}
\]

$M_1$

\[
\begin{align*}
\text{module } M &= \text{struct} \\
&\text{type } T = \text{Ref } \text{Int} \\
&\text{val } v_1 = \text{ref } 0 \\
&\text{end} : S \\
\end{align*}
\]

$M_2$

The Module $M_1$  

The Module $M_2$

The Signature $S$

As in the previous example, the different value bindings $v_1$ compute to a location storing different numerical values. However, the two modules are contextually equivalent as they are ascribed the same signature $S$ that abstracts the result of $v_1$ with an abstract type $T$. There exists no context $C$ that can bypass the abstract type $T$ and observe the difference in implementation. In this example contextual equivalence thus captures the confidentiality properties of ModuleML modules.

**Integrity**

Consider the following two contextually equivalent ModuleML modules $M_1$ and $M_2$ that uphold the signature $S$.

\[
\begin{align*}
\text{module } M &= \text{struct} \\
&\text{val } v_p = \text{ref } 0 \\
&\text{val } v_1 (f : \text{Int } \rightarrow \text{Int}) = v_p := \overline{1}; \\
&\quad (f \overline{2}); \\
&\quad 0 \\
&\text{end} : S \\
\end{align*}
\]

$M_1$

\[
\begin{align*}
\text{module } M &= \text{struct} \\
&\text{val } v_p = \text{ref } 0 \\
&\text{val } v_1 (f : \text{Int } \rightarrow \text{Int}) = v_p := \overline{1}; \\
&\quad (f \overline{2}); \\
&\quad \text{if } !v_p == \overline{1} \text{ then } \overline{0} \text{ else } \overline{1} \\
&\text{end} : S \\
\end{align*}
\]

$M_2$

The Module $M_1$  

The Module $M_2$

The Signature $S$

In this example both modules define a private value binding $v_p$ that is not a member of the signature $S$. In the public value binding $v_1$ both modules assign the location of $v_p$ to one, and apply the higher-order argument $f$ to the number
2. The only difference between both modules is the integrity check performed by M₂ that checks that the location in v_p was not modified while control was diverted to the outside argument.

Despite this difference in implementation, the two modules are contextually equivalent as there exists no context C that can provide an argument f capable of modifying the hidden value binding v_p. We say that in this example contextual equivalence captures the integrity properties of the MiniML modules.

6.2 The Secure Abstract Data Type Pattern

An A+I context must be able to perform the operations of ModuleML on the securely compiled ModuleML module. Each of these operations is different, but poses a similar secure compilation challenge: how do we enable the A+I context to perform the relevant operations without exposing the implementation details of the abstraction? In this chapter we introduce the Secure Abstract Data Type (ADT) pattern as a general approach to addressing this challenge. This pattern bundles together the individual techniques applied in certain secure compilation [66] and full abstraction results [54].

An ADT defines both the values of a data type as well as the functions that apply to it, relying on static typing rules to hide the implementation details of the data type. The Secure ADT pattern, in contrast, protects the implementation details of a source language abstraction τ without relying on static typing rules. As illustrated in Figure 6.1, it does this by inserting an ADT like interface between the actual implementation of the abstraction and the target language context. Concretely a secure ADT has the following elements: a secured type Sec[τ], an interface that defines the operations applicable to
the protected type, marshalling rules that handle the transitions between the
different representations for \( \tau \), and additional run-time checks if required.

**Secured Type**
The Secure ADT pattern states that values of the type \( \tau \), the type of the ab-
straction that the Secure ADT aims to secure, must be *isolated* and that value
that adhere to the type \( \tau \) thus cannot be shared directly with the attacker. In-
stead these values can be, for example, shared securely by encrypting them or
by providing a reference object, an object that refers to the original value (as
is the case for the masks of Section 3.4.1). The type of these securely shared
instances is denoted as follows.

\[
\text{Sec}[\tau]
\]

Every time the secure ADT must return a value of type \( \tau \) it will thus share an
instance of type \( \text{Sec}[\tau] \).

The Secure ADT pattern considers not only the secure sharing of values of
type \( \tau \), but also input from the possibly malicious target language context. The
type of inputs deriving from target language contexts are denoted as

\[
\text{Ins}[\tau]
\]

where \( \tau \) denotes the source language type that the input is expected to conform
to. Note that input from the target language can be of any possible type \( \tau \) not
just of the type \( \tau \) that the secure ADT pattern secures.

**Interface**
As illustrated in Figure 6.1, the interface defines a series of functions \((v_i)\) that
provide the outside context with the functionality of ModuleML. These func-
tions take as arguments some sequence of securely shared values and target
language input and return a securely shared or a target language value.

**Marshalling**
The Secure ADT pattern introduces *type-directed* marshalling functions to
handle the transitions between the values of the source language type \( \tau \), which
are the securely compiled values, the values of type \( \text{Sec}[\tau] \), which are the se-
curely shared instances, and the values of type \( \text{Ins}[\tau] \) which are input by the
outside context. A function of the following type signature:

\[
\text{Marshall}^\tau_{\sigma} : \tau \rightarrow \text{Sec}[\tau]
\]

converts values into their secured instances. A function of the type signature:

\[
\text{Marshall}^\tau_{\sigma} : \text{Sec}[\tau] \rightarrow \tau
\]

converts the secured instances back into the original value. Note that this
function performs an implicit run-time type check. It fails when given an
input that does not correspond to a securely shared value of type \( \tau \).
Certain secure compilation schemes, such as the one considered in this work, also specify a third type of marshalling function.

\[ \text{Marshall}^c_\tau : \text{Ins}[\tau] \rightarrow \tau \]

Such a marshalling function converts values from the target language context into values of the secured type \( \tau \), by converting the input value into the correct representation and by wrapping the result with type checks. Note that if the input is of type \( \text{Ins}[\tau'] \), where \( \tau' \neq \tau \), then the input will only be marshalled if a marshalling function of the following type signature is defined.

\[ \text{Marshall}^c_{\tau'} : \text{Ins}[\tau'] \rightarrow \tau'. \]

**Run-time checks**

The marshalling rules verify that the input provided by the outside target language context and the output shared with the outside context conform to the typing rules of the source language. This, however, is sometimes not enough to protect the abstractions of the source language. Certain security relevant language properties such as, for example, control-flow integrity, are not always explicitly captured by the typing system. Enforcing these properties must thus be done through additional run-time security checks.

### 6.3 A Secure Compiler for ModuleML

The secure compilation scheme for ModuleML is a type directed compilation scheme that compiles a standalone ModuleML module and its signature to a protected module (Figure 6.2) separating it from the low-level A+I context in the unprotected memory. This protected module is always of a fixed size, ensuring that the attacker cannot observe the size of the source program.

The secure compilation scheme applies the Secure ADT pattern both to the abstractions of MiniML as well as to the overall program. The entry points of the protected module implement an ADT-like interface to the A+I context. The abstractions of ModuleML are isolated by placing all code and data into the data and code sections of the protected module. The protected data section also includes a heap and stack that can only be accessed by the securely compiled program. This ensures that the run-time memory of the compiled program is also inaccessible to the low-level A+I context.

The inner workings of how ModuleML is compiled to assembly is of little relevance to this result of this chapter. Instead this section focuses on the security relevant aspects of the compilation scheme. This section details how we apply the Secure ADT pattern of Section 6.2 to securely compile abstract types (Section 6.3.2), structures and signatures (Section 6.3.3), functions (Section 6.3.4), locations (Section 6.3.5) and functors (Section 6.3.6). Basic types such as integers or tuples are not compiled using the Secure ADT pattern, but must still be marshalled when interacting with the A+I context (Section 6.3.1).
The compilation scheme and its implementation are limited in that we do not consider the security consequences of certain technical low-level challenges such as: integer overflows, stack overflows and out of memory exceptions.

6.3.1 Booleans, Integers and Tuples

The securely compiled module shares and inputs not only abstractions such as functions, but also basic ModuleML values: booleans, integers and tuples. Booleans and integers are exchanged with the A+I context using their respective A+I representation. The two marshalling functions for integers, for example, have as their type signatures:

1. \( \text{Marshal}_{\text{c}}^{\text{Int}} : \text{Ins}[\text{Int}] \rightarrow \text{Int} \): the marshalling function that converts A+I integers into ModuleML integers.
2. \( \text{Marshal}_{0}^{\text{Int}} : \text{Int} \rightarrow \text{Ins}[\text{Int}] \): the marshalling function that converts ModuleML integers to A+I integers.

The marshalling functions for booleans and unit are analogous.

Marshalling tuples is slightly different. When marshalling, for example, a pair \( \langle v_1, v_2 \rangle \) the marshalling functions for tuples marshall each value separately. Marshalling \textit{out} the pair:

\[ \langle T, \emptyset \rangle \]

for example, will thus produce a value of type

\[ \langle \text{Ins}[\text{int}], \text{Ins}[\text{int}] \rangle \]
while marshalling out the pair
\[ ((\lambda x : \tau.e), (\lambda x : \tau.e')) \]
will produce a value of the following type.
\[ \langle \text{Sec}[\tau \rightarrow \tau'], \text{Sec}[\tau \rightarrow \tau''] \rangle \]

### 6.3.2 Abstract types

Abstract types are, as the name indicates, abstract in that associated values are unobservable to an ModuleML context. Consider, for example, the following module \( A \) that conforms to the signature \( S \). This signature defines an abstract type \( T \) that abstracts the results of the value bindings \( v_1 \) and \( v_2 \).

```plaintext
module A : S =
  struct
    type T = Bool
    val v1 = true
    val v2 = v1
  end

signature S = sig
  type T
  val v1 : T
  val v2 : T
end
```

A malicious A+I context should not be able to observe that \( A.v_1 \) and \( A.v_2 \) both return the value \( \text{true} \).

To achieve this our compilation scheme applies the Secure ADT pattern to compile values of the abstract type. Instead of directly sharing the value of abstract type \( T \) with the A+I context, a secured instance of type \( \text{Sec}[T] \) is shared instead. These secured instances are implemented as indices to a map \( \mathcal{A} \). This map works \( \mathcal{A} \) in a manner similar to the binding map \( N \) of our secure foreign function interface (Chapter 3). The map \( \mathcal{A} \) maps natural numbers to values and their types in a deterministic manner, simply denumerating its entries. Note that like the \( N \) the map \( \mathcal{A} \) is not a set: it may map different numbers to duplicate elements.

![Diagram](#)

*Figure 6.3.* We use request counting to obscure the value of an abstract type.

As illustrated in Figure 6.3, the marshalling functions \( \text{Marshall}_0^T \) and \( \text{Marshall}_1^T \) are implemented by extending the map \( \mathcal{A} \) and looking up an index.
in $\mathcal{A}$ respectively. Every time a value of an abstract type is returned to the A+I context, the securely compiled module will thus share a new index $i$ to the map $\mathcal{A}$ that corresponds to the number of requests that the A+I context has made to abstract types. Note that each member of a sequence of abstractly typed values (Section 6.3.1) will count as a separate request.

We have shown previously by means of fully abstract translation proofs (Chapters 3 and 4) that these request counting indices do not reveal any information to the outside context (in this case the A+I context) other than the number of times the outside context has requested a value of an abstract type. This is information that the context of any source language with state can reproduce and thus does not harm full abstraction properties. In the case of ModuleML, a ModuleML context $C$ can count its interactions with the protected module by making use of references (a detail that returns in the proof of Section 6.4).

6.3.3 Structures and Signatures

Our compiler compiles both ModuleML structures and signatures into records stored within the data section of the protected memory. As dictated by the Secure ADT pattern these records are not exposed directly to the A+I context. Instead, the compilation scheme defines an ADT-like interface of entry points to the protected memory that provide access to the value and structure bindings exposed by the module’s signature.

Note that, as in previous work [66], these entry points are sorted alphabetically to obscure their implementation order. The compiler also includes a load entry point that evaluates each of the expressions defined within a structure. Our compilation scheme defines marshalling rules that both share secure structures as well as convert in structures created by the A+I context.

The Load Entry Point

As is the case in most ML module implementations, the value bindings of ModuleML map names to expressions not values. These expressions must be reduced to values before the value bindings of a structure can be queried. Our compiler, however, compiles a standalone ModuleML module not a full program, it thus does not have any control over when or if the expressions are evaluated. Instead our compilation scheme provides the A+I context the ability to load the module through a load entry point.

This load entry point takes no arguments and executes each of the expressions defined throughout the compiled module, storing the result in the appropriate record. Because it is up to the low-level context to invoke this load entry point, a malicious A+I context may attempt to query bindings before the module is loaded or attempt to load the module multiple times. To prevent this, the compiler introduces an additional run-time check in the form of a global flag

135
\( L_f \) that encodes whether or not the module has been loaded. What follows is a pseudo code implementation of the load entry point.

1. Check the flag \( L_f \). Abort if set.
2. For each and every value binding \( v_i \) in the compiled module:
   a) Evaluate the expression \( e \).
   b) Store the result in the appropriate record.
3. Set \( L_f \).

**Value Binding Entry Points**

For each value binding \( v_i \) declared within the signature of a structure, the compilation scheme creates an entry point with signature:

(a) \( v_i : \text{Sec}[\tau] \): when \( \tau \), the type of the value that the expression bound to \( v_i \) reduces to, is an abstraction that must be secured.
(b) \( v_i : \text{Ins}[\tau] \): when \( \tau \) is a basic type such as \textbf{Int}.

Both types of entry points are implemented as follows.

1. Check the flag \( L_f \). Abort if not set.
2. Fetch the value \( v \) and its type \( \tau \) from the data section.
3. Return \( \text{Marshall}^{\tau}_{v}(v) \)

**Entry Points to Structures**

For each module binding \( M_i \) to a structure \( s \) with signature \( S \) that is declared within the signature of the outer structure, our compiler creates an entry point of type:

\[ M_i : \text{Sec}[S] \]

that takes no arguments and returns a marshalled instance of the structure of the secured type \( \text{Sec}[S] \), as follows.

1. Check the flag \( L_f \). Abort if not set.
2. Return \( \text{Marshall}^{S}_{o}(s) \)

**Marshalling Secure structures In and Out**

As dictated by the Secure ADT pattern, structures are not shared directly but instead marshalled \textit{out} using the following type directed function.

\[ \text{Marshall}^{S}_{o} : S \rightarrow \text{Sec}[S] \]

This function converts a structure of signature \( S \) into a secured instance \( \text{Sec}[S] \). This secure instance is a record that contains an index \( i \) to a map \( \mathcal{M} \) and references to the entry points of each value/module binding in \( S \). The references
to the entry points are included to inform the A+I context of the functionality that the structure provides, simplifying interoperation. Like map $\mathcal{A}$ of Section 6.3.2, the map $\mathcal{M}$ maps numbers to structures and their signatures.

The indices of this map $\mathcal{M}$ enable the marshalling in function

$$Marshall_i^S : \text{Sec}[S] \rightarrow S$$

to retrieve the original structure and its signature from $\mathcal{M}$. Note that this marshalling in function performs an implicit type check as the marshalling function fails whenever the retrieved signature is not a subtype of $S$.

**Marshalling in A+I Defined Structures**

Our compilation scheme enables the A+I context to supply its own structures as arguments to the functors of Section 6.3.6. These structures are marshalled in by a function

$$Marshall_i^S : \text{Ins}[S] \rightarrow S$$

that iterates through the components of the expected signature $S$, querying the A+I context’s structure for the names of the bindings, marshalling in the results or aborting if a name isn’t found. When a value binding is marshalled in it is marshalled in using the type appropriate function. When a module binding is marshalled the marshalling function recurses. The marshalling is thus an immediate process, as marshalling in a lazy, on demand, manner would expose to the A+I context how and when its structure is used, something a ModuleML context cannot observe.

Note that this marshalling function performs a sub-type check:

$$\text{Ins}[S] <: S$$

as it only verifies the bindings defined by the expected signature $S$.

**6.3.4 Higher-Order Functions**

To compile the $\lambda$-terms of ModuleML the compilation scheme uses closure conversion [75] to eliminate free variables by using an explicit environment that stores bindings between variables and values. As is required by the Secure ADT pattern, these closures are not made available to the A+I context but are instead shared as secured instance of type:

$$\text{Sec}[\tau_1 \rightarrow \tau_2]$$

implemented as indices to a map $\mathcal{C}$ that maps numbers to closures and their types. As was the case for the indices of Section 6.3.2, these numbers simply denumerate the requests made by the A+I context. The marshalling functions

1. $Marshall_0^{\tau_1 \rightarrow \tau_2}$: marshalls out closures.
2. $Marshall_i^{\tau_1 \rightarrow \tau_2}$: marshalls in shared closures.

are thus implemented as extending the map $\mathcal{C}$ and looking up the closure and its type in $\mathcal{C}$ respectively.
The Closure Application Entry Point

As is required by the Secure ADT pattern we enable the A+I context to apply closures that have been shared with it, through a single polymorphic entry point of type:

\[ \text{appl} : \text{Sec}[\tau_1 \rightarrow \tau_2] \rightarrow (\text{Ins}[\tau_1] \lor \text{Sec}[\tau_1]) \rightarrow (\text{Ins}[\tau_2] \lor \text{Sec}[\tau_2]), \]

where the result is of the type \( \text{Ins}[\tau_2] \) if \( \tau_2 \) is a basic type and \( \text{Sec}[\tau_2] \) otherwise. This entry point takes as its arguments an index \( i \) to the map \( \mathcal{C} \) and a value \( v \) of the appropriate representation for type \( \tau_1 \). The entry point is implemented as follows.

1. Check the flag \( L_f \). Abort if not set.
2. \( c = \text{Marshall}_{\tau_1 \rightarrow \tau_2}(i) \)
3. Depending on the representation of \( v \):
   a) If \( \text{Sec}[\tau_1] \): \( r = \text{Marshall}_{\tau_1}(v) \)
   b) If \( \text{Ins}[\tau_1] \): \( r = \text{Marshall}_{\tau_1}(v) \)
4. Apply \( c \) to \( v \), store the result in \( r' \).
5. Return \( \text{Marshall}_{\tau_2}(r') \)

Note that the marshalling rules of 3(a) and 3(b) implement the typing rule for function applications, by ensuring that the input value \( v \) is of type \( \tau_1 \).

Marshalling In A+I functions

Our compilation scheme enables the A+I context to supply its own functions as arguments to the securely compiled entry points that accept an argument of type \( \text{Ins}[\tau_1 \rightarrow \tau_2] \). These A+I functions are marshalled by a function

\[ \text{Marshall}_{\tau_1 \rightarrow \tau_2} : \text{Ins}[\tau_1 \rightarrow \tau_2] \rightarrow (\tau_1 \rightarrow \tau_2) \]

that takes in a reference to the A+I function \( f \) and wraps that function into a new ModuleML function that performs the following steps, whenever the A+I function \( f \) is applied to a ModuleML value \( v \) within the securely compiled module.

1. \( a = \text{Marshall}_{\tau_1}(v) \)
2. Apply \( f \) to \( a \). Store the result in \( r \).
3. Return \( \text{Marshall}_{\tau_2}(r) \)

Note that the type marshalling rule on line 3 ensures that the result conforms to the typing rules of ModuleML.

An example

Consider, for example, the following two implementations of the ModuleML value bindings \( v_1 \). Both implementations apply two lambda functions to the
higher-order argument $g$: an outside function that takes two other functions as arguments. Implementation (a), however provides as arguments two functions $f_1$ and $f_2$, whereas, implementation (b) applies $g$ twice to $f_1$.

\[
\text{val } v_1:((\text{int} \to \text{int}) \to (\text{int} \to \text{int})) \to \text{int} = (g \ f_1) \ f_2
\]

Implementation (a)

\[
\text{val } v_1:((\text{int} \to \text{int}) \to (\text{int} \to \text{int})) \to \text{int} = (g \ f_1) \ f_1
\]

Implementation (b)

Assume for this example that the lambda functions $f_1$ and $f_2$ are two contextually equivalent functions, making both implementations of $v_1$ contextually equivalent.

The threat to compiler security in this example is that the low-level context could supply a function $g$ that could distinguish between both implementations of $v_1$, by observing that implementation (b) applies the argument to $f_1$ twice in contrast to implementation (a). Our compilation scheme prevents this by enforcing that every time a closure is exposed to the A+I context it is shared through a fresh index $i$. When the attacker inputs its own function $g$ that function will be, as noted earlier, marshalled into a wrapper function that will securely marshall out any argument $v$ from within the ModuleML module by using the $Marshall_{\phi}^v$ function. The attacker will thus observe two fresh indices in its interactions with both implementations.

6.3.5 Locations

As is the case in most commonly used ML variants [51], memory locations do not explicitly appear in the syntax used by programmers. This in contrast to the locations of MiniML used in chapters 3, 4 and 5.

Locations are thus not directly observable to a ModuleML context, leading to many equivalences. Consider, for example, the following two contextually equivalent implementations of the value binding $v_1$.

\[
\text{val } v_1 = (\text{let } x = (\text{ref } \text{true}) \text{ in } \text{let } y = (\text{ref } \text{true}) \text{ in } y)
\]

Implementation (a)

\[
\text{val } v_1 = (\text{let } x = (\text{ref } \text{true}) \text{ in } \text{let } y = (\text{ref } \text{true}) \text{ in } x)
\]

Implementation (b)

No ModuleML context can observe that the implementation (a) differs from implementation (b), in that it returns the second location it created, stored within variable $y$, and not the first location stored within the variable $x$.

Again our compilation scheme applies the Secure ADT pattern to protect ModuleML’s locations and the operations available on them. Locations are shared with the A+I context in the same manner as higher-order functions.
(Section 6.3.4) and abstract types (Section 6.3.2): as indices into a map $\mathcal{L}$ that maps numbers to locations and their types. As was the case previously, these numbers simply denumerate the requests made by the A+I context for access to ModuleML locations. The marshalling functions

1. $\text{Marshall}^{\text{Ref }}\tau$: marshalls out locations.
2. $\text{Marshall}^{\text{Ref }}\tau$: marshalls in shared locations.

are thus implemented by extending the map $\mathcal{L}$ and looking up an index in $\mathcal{L}$ respectively.

**Write and Read Entry Points**

To enable the low-level A+I context to write and read to shared locations in the same way that a ModuleML context can, we introduce a single polymorphic write location entry point of type:

$$\text{write} : \text{Sec}[\text{Ref} \tau] \to (\text{Ins}[\tau] \lor \text{Sec}[\tau]) \to \text{Unit}$$

and a single polymorphic read location entry point of the following type.

$$\text{read} : \text{Sec}[\text{Ref} \tau] \to (\text{Ins}[\tau] \lor \text{Sec}[\tau])$$

The write location entry point takes two arguments: an index $i$ to the map $\mathcal{L}$ and a value $v$ of the appropriate representation for type $\tau$. It securely writes $v$ to the appropriate location, as follows.

1. Check the flag $L_f$. Abort if not set.
2. $l = \text{Marshall}^{\text{Ref }}\tau(i)$.
3. Depending on the representation of $v$:
   a) If $\text{Ins}[\tau]$: $r = \text{Marshall}^{\text{Ref }\tau}(v)$
   b) If $\text{Sec}[\tau]$: $r = \text{Marshall}^{\text{Ref }\tau}(v)$
4. Write $r$ to $l$.

Note again, that the marshalling rules 3(a) and 3(b) implement the assign location typing rule, by ensuring that the input value $v$ is of type $\tau$.

The implementation of the read location entry point is straight-forward: it retrieves the location from $\mathcal{L}$, dereferences it and marshalls the value.

**Marshalling In A+I locations**

A ModuleML context can allocate new locations and share them with the ModuleML module embedded within the context’s hole. We thus enable the A+I context to supply its own locations as arguments to entry points that accept an argument of type $\text{Ins}[\text{Ref }\tau_1]$.

As specified by the Secure ADT pattern the locations provided by the A+I context are marshalled by a function

$$\text{Marshall}^{\text{Ref }}\tau : \text{Ins}[\text{Ref }\tau] \to \text{Ref }\tau$$

140
that takes in a location $l_f$ of the A+l context and wraps it with two functions. The first function enables a ModuleML expression to read the foreign location, the second function enables an ModuleML expression to write to the foreign location. The implementation of the latter is analogous to the implementation of the write entry point (Section 6.3.4). The implementation of the former is simply:

$$Marshall^c_l(l_f)$$

where $l_f$ denotes the dereference of the A+l location $l_f$.

### 6.3.6 Functors

As noted earlier (Section 2.2.1), a ModuleML functor is a higher-order function that maps modules (structures or functors) to modules. Consider the following example.

**Argument Signature $S_a$**

```
signature $S_a$ =
type U
val $v_1$: Int → Int
val $v_2$: U
des
```

**Result Signature $S_r$**

```
signature $S_r$ =
type T
val fd: Int → Int
val $F_1$: Functor(X:$S_a$)→$S_a$
val $M_1$: sig
   val $v_1$: T
des
```

The Functor $F$ and An Application

Module $F$ is a functor that maps a structure that conforms to signature $S_a$, to a new structure that consists of: a value binding $fd$, that applies the argument’s value binding $v_1$ to an argument $y$, and an inner functor $F_1$ and an inner structure $M_1$ that copies the argument. This new structure is ascribed with the signature $S_r$ which seals the value binding $M_1.v_1$ with the abstract type $T$.

When compiling functors the compiler operates in two modes. The first mode considers the static functor applications within the compiled module, such as, for example, the application of $F$ to an example module $M_i$ in the above listing. Compiling these applications is straightforward, the compiler
performs the application and compiles the result in the same way that it compiles any other module.

The second mode considers those functors that are part of the interface to the A+I context. In this case we must securely compile functors into run-time constructs. As is dictated by the Secure ADT pattern we do not share these run-time representations directly with the A+I context. Instead the run-time representations of functors are shared as indices into a map \( \mathcal{F} \) that maps numbers to functors and their types. These numbers (again) denumerate the requests by the A+I contexts for access to ModuleML functors. The marshalling functions

1. \( \text{Marshall}_o^{\text{Functor}}(X_i:S)\rightarrow S' \) : marshalls out functors.

where \( \text{Functor}(X_i:S)\rightarrow S' \) is the expected type of the functor, are implemented by extending the map \( \mathcal{F} \) and looking up an index in \( \mathcal{F} \) and confirming the type respectively.

Compiling Run-Time Functors

Functors are compiled into run-time constructs in a manner similar to the way in which \( \lambda \)-terms are compiled to closures. The functor body is compiled into a function that takes as its arguments a module and an environment of module bindings and returns a new module that conforms to the specification of the functor body.

In addition to being compiled into a function, every functor is also compiled into a tree structure of the accessible bindings, that defines a unique stamp \( \Sigma_i \) for each non-leaf node (Figure 6.4). These stamps \( \Sigma_i \) are used by the entry points for these bindings to authenticate the modules and inner modules that result from the A+I context applying the compiled functor.

\[
\begin{align*}
\text{module } F &\text{: functor } (A : S_a) \rightarrow \\
&\text{sig} \\
&\text{type } T \\
&\text{val } fd : \text{Int} \rightarrow \text{Int} \\
&\text{module } F_1 : \text{Functor}(X : S) \rightarrow \\
&(\text{sig} \\
&\text{type } U \\
&\text{val } v_1 : \text{Int} \\
&\text{val } v_s : U \\
&\text{end}) \\
&\text{module } M_1 : \\
&(\text{sig} \\
&\text{val } v_1 : T \\
&\text{end}) \\
&\text{end}
\end{align*}
\]

\( \Sigma_1 \)

\( \Sigma_2 \)

\( \Sigma_3 \)

\( F \)

\( F_1 \)

\( M_1 \)

\( v_1 \)

\( v_y \)

\( v_1 \)

\( v_1 \)

\( F_1 \)

\( M_1 \)

Figure 6.4. The secure compiler compiles the signature of \( F \) into a tree of unique stamps \( \Sigma_i \), that enable the functor entry points to identify their arguments.
The module that results from applying a run-time functor is stored as a record that incorporates the resulting module as well as additional run-time data. Additionally the record stores a stamp $\Sigma_i$ that identifies the functor that produced it, a module binding environment $e$, which includes the argument to the functor, and environment of abstract type identifiers $e_t$. The latter is required to keep track of the abstract types that are created by functors that seal their result, such as our example functor $F$ which seals $M_1.v_1$, as they generate a new abstract type each time they are applied.

**Functor Application Entry Point**

To enable the low-level A+I context to apply secure functors to modules in the same way that an ModuleML context can, we introduce a single polymorphic functor application entry point into the protected memory that is of the type:

$$\text{fappl} : \text{Sec}[\text{Functor}(X_i : S) \to S'] \to (\text{Ins}[S] \lor \text{Sec}[S]) \to \text{Sec}[S'].$$  

The first argument to this entry point is an index to the map $\mathcal{F}$, the second argument $m$ is a shared module or a module defined by the A+I context that conforms to the signature $S$. The entry point securely applies the appropriate functor $f$ with associated stamp $\Sigma_f$ to the argument $a$, as long as $a$ conforms to the signature $S$, as follows.

1. Check the flag $L_f$. Abort if not set. 
2. $f = \text{Marshall}_{i}^{\text{Functor}(X_i : S) \to S'}(i)$. 
3. Depending on the representation of $m$:
   a) If $\text{Ins}[S]$: $a = \text{Marshall}_{C}^{S}(a)$
   b) If $\text{Sec}[S]$: $a = \text{Marshall}_{I}^{S}(a)$
4. Apply $f$ to $a$. Store the result in $r$.
5. Stamp $r$ with $\Sigma_f$.
6. Return $\text{Marshall}_{o}^{S'}(r)$.

Note that as specified in Section 6.3.3, the marshalling rules of 3(a) and 3(b) perform the sub-typing check required by the functor application rule.

**Functor Entry Points**

Besides enabling the A+I context to apply a functor, the secure compilation scheme also outputs entry points that enable the A+I context to gain access to the functor as well as interact with the result of the functor application. The entry points to functor bindings that are not embedded within another functor have a type:

$$M_i : \text{Sec}[\text{Functor}(X_i : S) \to S']$$

and marshall out the associated functor through an index to a map $\mathcal{F}$. 

143
The entry points to the bindings of structures that are defined within the body of a functor, differ from the previously detailed entry points for value, structure and functor bindings in that they take an argument: an index $i$ to the map $\mathcal{M}$. As detailed in the previous paragraph, the functor application entry point marshalls out its result through the marshalling function $\text{Marshall}^S_{\mathcal{M}}$, which as explained in Section 6.3.3, stores the result into the structure requests counting map $\mathcal{M}$.

Our compilation scheme compiles a value binding $v_i$ of a structure with a signature $S$ defined within the body of a functor $F_i$, into an entry point of type:

$$v_i : \text{Sec}[S] \rightarrow \text{Sec}[\tau]$$

if the value binding has a type that must be secured, or

$$v_i : \text{Sec}[S] \rightarrow \text{Ins}[\tau]$$

when the value binding $v_i$ returns a value of a type that must not be secured (Section 6.3.3). Likewise, the compilation scheme compiles a structure binding $M_i$ of type $S_i$ defined within that same structure of $F_i$ into an entry point of type:

$$M_i : \text{Sec}[S] \rightarrow \text{Sec}[S_i]$$

and compiles a functor binding $M_f$ of type $\text{Functor}(X_i : S_f) \rightarrow S'_f$ within that structure into a functor binding entry point of the following type.

$$M_i : \text{Sec}[S] \rightarrow \text{Sec}[\text{Functor}(X_i : S) \rightarrow S'_f].$$

The implementations of all of these entry points extend the previously discussed entry point implementations (Sections 6.3.2, 6.3.3, 6.3.4, 6.3.5) in that their result is not statically defined but depends on the structure associated with the input index $i$. The entry points will thus look up index $i$ in $\mathcal{M}$ and check that the retrieved structure is stamped with the correct stamp $\Sigma_i$, as follows.

1. $d = \text{Marshall}^S_{\mathcal{M}}(i)$.
2. Check that stamp of $d = \Sigma_i$. If not Abort.

To illustrate the necessity of this stamp check, we reconsider the example functor $F$ introduced at the beginning of this section. This functor is assigned the stamp $\Sigma_1$ (Figure 6.4) and each of its bindings $F.fd$, $F.F_1$ and $F.M_1$, check that the structure associated with input index $i$ is stamped by $\Sigma_1$. If they did not do so the A+I context could, for example, violate the typing rules of ModuleML (Section 2.2.1) by passing a structure created using $F$ to the bindings of the following functor $F_b$.  

144
<table>
<thead>
<tr>
<th>Argument Signature $S_b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>signature $S_b = \text{sig}$</td>
</tr>
<tr>
<td>type U</td>
</tr>
<tr>
<td>val $v_1 : \text{Int} \rightarrow \text{Int}$</td>
</tr>
<tr>
<td>val $v_s : \text{Int}$</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

A Leaky Functor $F_b$

While both $F_b$ and $F$ produce a structure with signature $S_r$, the argument of $F_b$ conforms to the signature $S_b$ not the signature $S_a$, which seals the binding $v_s$ whereas $S_b$ does not. Without the stamp checking mechanism the A+I context could break the abstractions of ModuleML by passing a module produced by applying $F$ to the entry point for $F_b,F_1$ as the implementation of $F_b,F_1$ exposes the value binding $A.v_s$, as highlighted in gray in the listing for $F_b$.

The entry points for $F,F_1$ and $F,M_1$ stamp their result with stamps $\Sigma_2$ and $\Sigma_3$ respectively. This further specialization of the stamps within the inner modules is necessary to prevent similar attacks.

### Marshalling In A+I Functors

Our compilation scheme enables the A+I context to supply its own functors as arguments to the functor application entry point. These foreign functors are marshalled into ModuleML functors by a function:

$$\text{Marshall}_{c}^F(X : S) \rightarrow S'$$

that takes in a reference to an A+I function $f$ and wraps that function into a new function that performs the following steps, whenever the foreign functor is applied to a ModuleML module $M$, within the securely compiled module.

1. $a = \text{Marshall}_c^F(M)$
2. Apply $f$ to $a$. Store the result in $r$.
3. Return $\text{Marshall}_c^{S'}(r)$

### 6.4 Compiler Reflection

To establish that our compilation scheme is a secure, we prove that the compilation scheme reflects the contextual equivalences of ModuleML in the output assembly code. Meaning that if two ModuleML modules are indistinguishable...
to any possible ModuleML context, the assembly codes output by the compiler are indistinguishable to possibly malicious A+I code.

Denote the result of compiling the module $M$ down to A+I as $M\downarrow$. Compiler reflection is formally expressed as:

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

To prove this statement we will prove the contra-positive:

$$M_1\downarrow \not\simeq^{MM} M_2\downarrow \Rightarrow M_1 \not\simeq M_2$$

this contra-positive can be stated as, whenever an A+I context can distinguish between two compiled modules, there exists a ModuleML context that can distinguish between the original modules. As detailed in Section 2.7 we do not directly reason about contextual equivalence for A+I programs but instead rely on trace equivalence. As such we can redefine compiler reflection as:

**Theorem 7 (Module Differentation)** Any two ModuleML modules $M_1$ and $M_2$ whose compilation results produce two different low-level traces $\alpha_1$ and $\alpha_2$ are not contextually equivalent.

$$\text{Tr}(M_1\downarrow) \neq \text{Tr}(M_2\downarrow) \Rightarrow M_1 \not\simeq M_2$$

In order to prove that $M_1$ and $M_2$ are not contextually equivalent, it suffices to show that there exists an ModuleML context $C$ that can distinguish between $M_1$ or $M_2$. As per Definition 16 this context should diverge for one of the modules but not for the other.

To prove Theorem 7 we adopt the established proof technique [66, 42] of developing an algorithm that given two modules $M_1$ and $M_2$ and their differing low level traces $\alpha_1$ and $\alpha_2$ can produce a witness context $C$ that can distinguish between $M_1$ and $M_2$. We have implemented exactly such a witness building algorithm in Ocaml.¹ The algorithm analyses the labels of the low-level traces $\alpha_1$ and $\alpha_2$ that detail the interactions between an unknown A+I context (it’s a black box) and the modules $M_1$ and $M_2$.

For the algorithm to be correct, it must detect the first two labels $\alpha$ in the traces that differ. From Section 2.7, we have that those labels can be: call, ret, write, read and termination $\sqrt{\,}$. Because we have assumed that the execution starts outside of the compiled module (Section 6.3.3), the actions that appear at even-numbered positions in a trace will be calls or returnbacks from the A+I context, whereas actions that appear at odd-numbered positions will be returns or callbacks generated by the ModuleML modules $M_1$ or $M_2$. Assuming the first differing labels are at position $i$, the algorithm produces an ModuleML module that will replicate the first $i-1$ labels of the traces and at the $i$-th step will diverge for $M_1$ and terminate for $M_2$, distinguishing them as required.

¹https://github.com/sylvarant/moduleml-witness-algorithm
Both in the implementation and in the examples that follow the low-level traces are adapted to be more understandable. For example, if the entry point of a value binding $v_1$ is located at 0x234, the low-level label call 0x234 will be written as call $v_1$. This abstraction is safe, as it does not introduce additional information, it merely makes the traces more readable.

The algorithm starts with the following witness context:

```plaintext
struct
open M
val $v_c$ = ref 0
val $v_d$ = ($\lambda$ x : Bool =
  letrec div : Bool $\to$ Bool =
    (fun x : Bool = (div x)) in (div x))
val $v_m$ = unit
end ;; $v_m$
```

The open $M$ statement includes the modules $M_1$ and $M_2$ as a module $M$, thus implementing the hole of the witness. The value binding $v_m$ is of type Unit and serves as the starting point of the witness. The witness needs to keep track of the index that it replicates its, this is done by updating the value binding $v_c$, which holds a reference to the count. When the witness is able to differentiate two actions of the input modules it must diverge in one case and terminate in the other. Divergence is implemented through a diverging boolean function in the value binding $v_d$.

As the algorithm iterates through the traces it expands the witness as needed, using the signature obtained from type checking as its knowledge base for the input modules. In what follows we provide three examples. The first example illustrates how the algorithm builds a witness context that observes traces of different length. The second example illustrates how the algorithm builds a witness context that reproduces the actions of traces that apply functions of the assembly context to closures. The third example illustrates how the algorithm builds a witness context that reproduces the actions of traces that apply structures of the assembly context to secure functors. For more examples, such as of traces that return different locations we defer to the implementation.

**Different Trace Length**

Consider the following trace $\alpha_1$:

```
call $v_1$? · ret 1! · ✓
```

where $v_1$ is a value binding of the modules to be distinguished. The second trace $\alpha_2$ is similar except that it does not return from the call made by the witness.

```
call $v_1$?
```

The input modules uphold the following signature:
The algorithm reproduces action 0 of the trace by having the witness call \( v_1 \) from the starting point \( v_m \) and store the result in a variable \( x \). Next the algorithm has the witness terminate by means of the `exit` statement. This is all the witness must do in this case. The fact that the second trace \( \alpha_2 \) does not produce a response, implies that it is diverging and will thus force the witness to diverge when it imports \( M_2 \) but not when it imports \( M_1 \). The witness thus distinguishes between them as required by Definition 16. The produced witness is listed below:

```ml
struct open M
  val \( v_c \) = ref 0
  val \( v_m \) = incr_v_c;
  (let x = M.v_1 in exit unit)
end ;; v_m
```

**Different CallBack Argument**

Consider the trace \( \alpha_1 \):

\[
call v_1? \cdot ret 1! \cdot call apply(1,0x12)? \cdot call 0x12(4)! \cdot ret 6? \cdot call 0x12(4)!
\]

where `apply` is the closure application entry point and `0x12` is the address of an assembly context. The second trace \( \alpha_2 \) is identical except for the last action which is `call 0x12(5)`! The input modules are typed as:

```
sig v_1 : Int \to Int \to Int end
```

The algorithm reproduces action 0 of the trace by having the witness call \( v_1 \) from the starting point \( v_m \) and store the result in a variable \( x \). Internally the algorithm also rebuilds the map \( N \) associating the index 1 with the variable \( x \) and the type of \( v_1 \). Note that, as discussed in Section 6.3.4, the index of the first action is always 1. To reproduce action 2 of the trace the algorithm must infer what the address `0x12` points to. By looking up the index 1 in \( N \) it the algorithms deduces that the argument must be function within the witness of type: \( \text{Int} \to \text{Int} \).

Given no other information the algorithm will start by constructing a value binding \( v_{0x12} \) to a \( \lambda \)-term that returns 0 (the default value for \( \text{Int} \)) and reproduces action 2 by applying \( x \) to \( v_{0x12} \). When it reads action 3 the algorithm switches from implementing the value binding \( v_m \) to implementing the value binding \( v_{0x12} \) and simply has it return 5 when the witness step count is 2 to reproduce action 4. The final actions differ in that the traces call \( v_{0x12} \) with different arguments. In this case the witness simply diverges if the argument is 4 otherwise it exits with value 0. The resulting witness is listed below:
struct
open M
val v_c = ref 0
val v_d = (\x : Bool =
letrec div : Bool \rightarrow Bool =
(fun x : Bool = (div x)) in (div x))
val v_{0,12} = (fun y:Int =
(if (v_c == 2) then incr_v_c; 5
else (if (v_c == 3) then (if (y == 4) (v_d true)
else (exit 0)) else 0))
val v_m = incr_v_c;
(let x = M.v_1 in incr_v_c; (x v_{0,12}))
end ;; v_m

Where incr_v_c is short for
\[ v_c := (!v_c + 1) \]
and M is the module that fills the hole. Note that within the main function \( V_m \) we do not check the current step count as the secure compiler does not produce code that calls the main function of the context.

**Different Return From a Dynamic Module**

Consider the following trace \( \alpha_1 \):

\begin{center}
\texttt{call X_1? \cdot ret 1! \cdot call applyf(1,0xa)? \cdot read(0x8a,R_a) \cdot write(0xa,R_r) \cdot ret 1! \cdot call V_1(1)? \cdot ret 0!}
\end{center}

where \( \texttt{applyf} \) is the functor application entry point, and \( R_a \) is a record of a structure of the assembly context and \( R_r \) is a record returned by the modules as discussed in Section 6.3.6. The second trace \( \alpha_2 \) is identical except for the last action which is \texttt{ret 1!}. The input modules are typed as:

\begin{center}
\texttt{sig module F_1 : Functor(X_i : sig val v_1 : Int end) \rightarrow (sig v_1 : Bool)}
\end{center}

The algorithm deduces from the signature that \( F_1 \) is a functor. The algorithm rebuilds the map \( \mathcal{F} \) internally by associating \( F_1 \) with the index 1 and has the witness increase the step count by one. Note that it follows from Section 6.3.6 that action 1 will always return the same index for modules of the same signature, if that were not the case we would not be able to replicate a low-level call to a functor. To reproduce action 2 the algorithm deduces by looking up index 1 in \( \mathcal{F} \) that it must build a module \( X_{0,ax} \) that corresponds to the signature of \( X_i \) and apply it to \( F_1 \). Like functions modules are first constructed with default values based on the types and then expanded upon later.

The algorithm reproduces action 2 by applying \( X_{0,ax} \) to \( F_1 \) and taking another empty step. To reproduce action 4 the algorithm calls the value binding \( V_1 \) on
the result of the previous application. The final action of the traces \( \overline{a_1} \) and \( \overline{a_2} \) differs in that they return different booleans. The witness thus simply diverges as it did in the previous example. The resulting witness is listed below:

```ocaml
struct
open M
val v_c = ref 0
val v_d = (\x : Bool =
  letrec div : Bool -> Bool =
    (fun x : Bool = (div x)) in (div x))
module X_a = struct val v_1 = 0 end
module X_r = M.X_1(X_1)
val v_m = incr_v_c; incr_v_c;
  let x = X_r.v_1 in incr_v_c; (if (x == false)
    then (v_d true) else (exit unit))
end ;; v_m
```

### 6.5 Implementation and Experimental Results

We have developed a compiler\(^2\) that compiles ModuleML modules using either the secure compilation scheme detailed in this chapter or through a *naive* compilation scheme that features none of the security checks. The compiler targets the Fides implementation of PMA [80].

Fides implements PMA through use of a hypervisor that runs two virtual machines: one that handles the secure memory module and one handles the outside memory. One consequence of this architecture is that, as the low-level context interacts with the compiled module, the Fides hypervisor will be forced to switch between the two virtual machines for each call and callback between the context and the module, an expensive operation.

The security checks described in this chapter are only triggered when execution crosses the boundary between protected and unprotected memory. We have benchmarked five scenarios (included with the source code of the compiler) that involve boundary crossings. In the first scenario (*Value*) the A+I context retrieves a value binding by calling the appropriate entry point. In the second scenario (*Closure Application*) the A+I context applies a secure closure to another secure closure using the closure application entry point. In the third scenario (*Callback*) the attacker applies a secure closure to a function of the A+I context. In the next scenario (*Functor Application*) the A+I context applies a functor to a module of the A+I context using the functor application entry point. In the final scenario (*Dynamic Value*) the A+I context accesses the value binding of a structure that results from applying a functor at run-time. We have timed the average performance of each of these five scenarios, as given in Table 6.1

\(^2\)https://github.com/sylvarant/secure-ml-compiler
<table>
<thead>
<tr>
<th>Operation</th>
<th>Unprotected</th>
<th>Fides</th>
<th>Fides &amp; Secured</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value Binding</td>
<td>0.18µs</td>
<td>17.59µs</td>
<td>17.86µs</td>
</tr>
<tr>
<td>Closure Application</td>
<td>0.32µs</td>
<td>17.68µs</td>
<td>18.09µs</td>
</tr>
<tr>
<td>Callback</td>
<td>0.31µs</td>
<td>36.59µs</td>
<td>36.97µs</td>
</tr>
<tr>
<td>Functor Application</td>
<td>0.57µs</td>
<td>37.14µs</td>
<td>106.50µs</td>
</tr>
<tr>
<td>Dynamic Value Binding</td>
<td>0.26µs</td>
<td>17.73µs</td>
<td>18.41µs</td>
</tr>
</tbody>
</table>

Table 6.1. The average execution time of code produced by our compiler for five security relevant scenarios.

This table details the average execution time without protection, with Fides and with Fides as well as the security checks introduced by the compilation scheme. The tests scenarios were performed between a hundred thousand and ten million times depending on the performance impact of the scenario, on a Dell Latitude with a 2.67 GHz Intel Core i5 and 4GB of DDR3 RAM.

The difference between rows Unprotected and Fides shows the high overhead of the Fides architecture. It is especially notable in the call back and functor application scenarios which transition between the protected and unprotected memory twice. The security checks of the functor application scenario have by far the biggest performance impact. This is due to the fact that this scenario involves both the dynamic type checking of the structure input by the A+I context as well as the creation of a new module, two computationally intensive operations. This functor application operation is, however, unlikely to be frequently invoked by the average program.

The additional performance impact of the security checks in the other scenarios is small, peaking at about 4% when securing the value binding of a dynamically obtained structure.

6.6 Summary

This chapter presented a secure compiler for ModuleML: a light-weight ML language with higher-order functions, references and a module system. This secure compiler, compiles ModuleML to untyped assembly code enhanced with PMA in a way that reflects the equivalences of MiniML. To simplify the compilation scheme, we introduced the secure abstract data type pattern that captures many of the intricacies of securely compiling a functional programming language in a general manner. The reflection property of the compilation scheme was proven by implementing a witness building algorithm.
7. Related Work

This chapter presents related work. Firstly, it discusses alternative low-level security architectures (Section 7.1). Then it surveys related results for secure foreign function interfaces (Section 7.2), high-level attacker models (Section 7.3), abstract machine derivation (Section 7.4) and secure compilers (Section 7.5). Finally it discusses similar applications of bisimulations (Section 7.6).

7.1 Low-Level Protection Mechanisms

In this section we discuss the different versions of PMA (Section 7.1.1), as well as two alternative mechanisms: sandboxing (Section 7.1.2) and address space layout randomization (Section 7.1.3).

7.1.1 Protected Module Architecture (PMA) Implementations

PMA is the low-level memory protection mechanism used to secure functional programs in this work (Section 2.6). Our current implementations of the secure compiler and the abstract machine we make use of Fides [80], a hypervisor based approach to PMA. Previous versions of both software artifacts ran on Sancus [64], a modified MSP430 implementation of the PMA infrastructure. Future versions of our artifacts will be compatible with Intel SGX, a commercially supported implementation of PMA [57].

Other possible targets are Salus, Flickr and Mondriaan. Salus is an Linux kernel-based implementation of PMA [10]. Flicker is another software-based PMA implementation that can executes designated snippets of code in isolation, ensuring the security of important information [56]. Mondriaan is a PMA implementation with fine-grained protection schemes aimed at developing less monolithic operating systems [87].

7.1.2 Sandboxing

A commonly used alternative to PMA is sandboxing [34, 89, 26]. A sandbox is a restricted environment where code that is not trusted is placed and monitored, using both static and dynamic analysis, to detect any possible malicious behaviour. A sandbox is in some sense the dual of PMA, in contrast of placing secure code in a safe memory space, a sandbox places untrusted code in a monitored environment. Many of the protection techniques we have developed should thus be transferable to sandboxed programs.
7.1.3 Address Space Layout Randomization (ASLR)

Address Space Layout Randomization (ASLR) is a memory protect technique that introduces randomness into the addresses used by an executable [83, 33]. Under ASLR an executable is divided into segments whose order is randomised by the dynamic linker.

ASLR is used to hinder code reuse attacks by removing the attackers ability to locate code gadgets that perform a desired set of operations. It also prevents attackers from using knowledge about the location of certain data to access that data in subsequent runs of a program. Successful random accesses are still possible, their probability is however very low.

7.2 Secure Foreign Function Interfaces

In this section we cover competing foreign function interface designs (Section 7.2.1) as well as alternative methods for securely sharing data structures (Section 7.2.2).

7.2.1 Foreign Function Interfaces

Techniques that ensure secure interoperation between languages with different levels or kinds of abstraction, have been developed before. Furr and Foster address the complications that arise when OCaml interoperates with C, by developing a multi-language type system that embeds OCaml types in C and vice-versa [29]. They, however, assume that the C code is not an attacker and will thus not circumvent their typing system.

Tan et al. tackle the issues that arise when Java interoperates with C through SafeJNI [82], a framework that ensures type safety through the Java foreign function interface. Their system however, requires both static and dynamic checks on the C code that Java interoperates with. Our secure foreign function interface design, on the other hand, does not require any static checks on the attacker.

Matthews’ and Findler’s multi-language semantics [55] provide a technique for specifying operational semantics that allows two languages to interoperate in a way that preserves termination and type safety. In their work however, they aim to abstract away low-level details and instead focus on semantic properties. Our foreign function interface design, in contrast, explicitly lifts low-level interoperation details into the operational semantics.

Gampe et al. present a technique that establishes the non-interference properties of two interoperating languages with different security typing mechanisms [30]. They do not, however, consider any attacker model.

Zdancewic et al. present a multi-agent calculus that treats the different modules that make up a program as different principals, each with a different
view of the environment [90]. Their work, however, models the different views each agents sees through typing.

7.2.2 Secure Sharing

In both our secure foreign function interface as well as our secure compilation scheme we make use of a masking system which counts the interactions between the secure and the attacker (Sections 3.4.1,6.3.2), to securely share security relevant terms and values. Alternatively, we could have applied the encryption mechanism of Matthews et al. [54], that encrypts all security relevant constructs before sharing them with possibly malicious code. Their encryption mechanism does, however, require many more computational resources.

The masking mechanism of Patrignani et al. [66], however, could not be used to replace our masking mechanism as their mechanism identifies equal objects/constructs and assigns them the same mask. Applying that approach for masking the security relevant constructs of MiniML and ModuleML, would provide the low-level attacker with information not available to contexts, breaking our security results.

The marshalling rules of our secure foreign function interface design are reminiscent of Finne et al.’s H/Direct FFI [25]. Our marshalling rules, however, mask security relevant values with denumerable names, whereas in H/Direct security relevant terms are shared as is.

7.3 High-Level Attacker Models

The attacker model of Chapter 4 is based on the insights of Mitchell [60], Kennedy [45] and Wand [85] concerning the full abstraction and security consequences of reflection operators in programming languages.

Alternative high-level attacker models are Jagadeesan et al.’s attacker language $\lambda_{\mu}$proberef with low-level memory access operators [40] or the erasure function approach of several non-interference works [52]. The former is not a suitable attacker model for PMA as the protected memory of PMA is inaccessible to the low-level attacker, even via random access. The latter does not lend itself to low-level attackers as it only considers the labeling mechanism of non-interference.

A more low-level attacker model is the typed assembly language developed by Morrisett et al. [61]. The drawback of this attacker model, is the typing constraint applied to it. As long as there is only a very small number of real-world compilers and processors that target and execute this typed assembly language, the typed nature of the attacker will remain an unrealistic constraint.
7.4 Abstract Machine Derivation and Security

This section surveys both the existing techniques for deriving abstract machines (Section 7.4.1) and similar proofs of source language transformations (Section 7.4.2).

7.4.1 Abstract Machine Derivation Techniques

Step 3 of our methodology for implementing secure abstract machines (Section 5.2.3) requires the user to derive an abstract machine from a labelled transition system over a foreign function interface. When deriving our secure CESK$^+$ machine for MiniML we make use of the syntactic correspondence of Biernacka and Danvy [12] between Curien’s $\lambda_p$-calculus and CESK machines. Other syntactic correspondences have focused on calculi with lazy evaluation and calculi with objects [16] and targeted other abstract machine types such as the Spineless Tagless G-machine [73].

An alternative approach to syntactic correspondence could be to adapt Ager et al.’s functional correspondence [4] between evaluators and abstract machines. In contrast to syntactic correspondence, functional correspondence derives new specifications from evaluators instead of the source language. This approach has been applied to multiple different languages and abstract machines, including Landin’s SECD machine [17] and multi-level languages [39].

7.4.2 Proofs of Language Transformations

The validation of our methodology for deriving secure abstract machines (Section 5.2.5) relies on chain of bi-implications between bisimulations over the source language and each result of the transformations used to derive the abstract machine formalisation in Step 3 (Section 5.2.3). This chain is similar to the chain of compilation steps found in the work of Morrisett et al. [62].

While the correctness of that chain of compilation steps was not proven in the original work, Ahmed and Blume have subsequently proven that two of the transformations used in that chain compilations steps are fully abstract translations. Ahmed and Blume have proven that a typed continuation-passing-style translation from the simply-typed $\lambda$-calculus to System F is a fully abstract translation [7] and that a typed closure conversion, similar to the closure conversion of Section 5.3.1, from (and to) System F is fully abstract [6].

7.5 Secure Compilation

In this section we cover competing secure compilation schemes (Section 7.5.1) as well as closely related work on verified compilation schemes (Section 7.5.2).
7.5.1 Fully Abstract Compilation

Secure (fully abstract) compilation was first introduced by Abadi [1] as a criticism of the way in which Java was translated into the Java bytecode language. Secure compilation schemes have since been introduced for many different source and target languages.

Closely related to our work in Chapter 6 is the secure compilation scheme for ML to JavaScript by Fournet et al. [27]. Their definition of ML, however, does not feature a module system. Their Javascript attacker model is also slightly more high-level than our untyped assembly contexts with low-level code execution privileges.

Another related compilation scheme is the secure compilation scheme for the $\lambda$-hashref-calculus to a machine model with address space layout randomisation by Jagadeesan et al. [40]. Like the ModuleML source language of our secure compiler the $\lambda$-hashref-calculus features dynamic memory allocation. In contrast to ModuleML, locations in $\lambda$-hashref are observable through a hash operation similar to how locations are observable in MiniML through the index construct. As noted in Section 7.3 their attacker model differs from our low-level attacker as well as PMA prevents low-level attackers from accessing memory addresses.

The compilation scheme of Jagadeesan et al. [40] is an extension prior work by of Abadi and Goroden [2], whose compilation scheme compiles the simply-typed $\lambda$-calculus extended with an abstract memory to a $\lambda$-calculus extended with a concrete memory. Abadi and Gordon prove that given a sufficiently large memory space that employs ASLR, their compilation scheme can achieve probabilistic full abstraction.

Agten et al. were the first to present a fully abstract compilation scheme that uses PMA to preserve confidentiality and integrity properties of their source languages [5]. The barebones object oriented source language of their compilation scheme was later expanded by Patrignani et al. to include dynamic memory allocation, exceptions, inner classes and cross package inheritance [69]. Their results differ from ours, however, in that their compilation scheme limits the input of the attacker to basic values and does not consider the security challenges inherent to functional programming languages such as nameless functions and higher order modules.

7.5.2 Verified Compilation

Verified compilation is a broad research topic that aims to provide compilers that are proven to be correct [13, 50], meaning that the compilers are proven to output programs that uphold the semantics of the original source language program. The resulting compilers thus come with proofs for the preservation property that we have assumed to hold, in our formal proof of compiler security (Section 6.4).
Related to this work is a verified compositional compiler for an ML language that features references and recursive types, to assembly by Hur and Dreyer [38]. Their compiler preserves the equivalences of ML programs for well-behaved assembly contexts, but does not consider the threats posed by possibly malicious contexts.

This well-behaved context assumption is not an unusual constraint for verified compilation results. Most established verified compilation results, hold only under a closed world assumption: that the compiler compiles whole programs and that every context derives from the compiler thus cannot misbehave. This is in contrast to our secure compilation scheme, where we consider the context to be arbitrary low-level attackers. An exception to this limitation of verified compilation work is the work by Perconti et. al. on verifying a compiler that does not compile whole programs [70]. That work, however, targets the typed assembly language of Morrisett et al. [61], where malicious contexts are well-typed assembly programs, in contrast to the untyped assembly language A+I that this thesis considers.

7.6 Bisimulation

Our notions of bisimulation over MiniML, MiniML+ and the abstract machines that we derive from MiniML+ are inspired by previous works on applicative bisimulation (Section 7.6.1). A possible alternative to our applicative bisimulation is environmental bisimulation (Section 7.6.2) and fully abstract trace semantics (Section 7.6.3).

7.6.1 Applicative Bisimulation

Bisimulation has been applied to functional programming languages before, most notably by Abramsky in his work on an applicative bisimulation for the lazy $\lambda$-calculus [3]. The notions of applicative bisimulation for MiniML and MiniML+ are based on the applicative bisimulation for the vref-calculus by Jeffrey and Rathke [41], the applicative bisimulation for ML by Fournet et al. [27], the fully abstract trace semantics for the $\lambda_\mu$-hashref-calculus by Jagadeesan [40] and the trace semantics for general references by Laird [46]. The labels of our labelled transition systems do, however, differ from the labels used by Laird and Fournet et al. as they do not explicitly denote a store of the shared locations in the labels of the traces.

Our proof of congruence for the bisimulation over MiniML+ relies on Gordon’s proof of congruence for FPC [35]. That proof is itself an adaption of Howe’s proof method for bisimulation [37].
7.6.2 Environmental Bisimulation

An alternative to the applicative bisimulations for MiniML and MiniML+ is environmental bisimulation [77]. The advantage of environmental bisimulation is that it can capture more abstract language constructs than the applicative bisimulation that we make use of. Using environmental bisimulation as a semantics for MiniML, for example, has the advantage of not needing the index operation to be included in MiniML, as an environmental bisimulation can capture the locations of a functional programming language without requiring an explicit denotation of the locations in the labels of a labelled transition system. Similarly, the global and denumerable names of MiniML+ are a lot more explicit than the local seals captured by the environmental bisimulation of Sumii and Pierce [81].

While environmental bisimulation can capture more abstract language constructs, they are of limited use in our work as they, in contrast to our applicative bisimulation, do not provide us with a clear formalism to reason about the observations and actions of an attacker.

7.6.3 Fully Abstract Trace Semantics

Trace semantics was first introduced as a means for studying the behaviour of concurrent CSP [28] and has since been adopted as a formalism for capturing the behaviour of functional programming languages [46, 40, 2, 32], object-oriented languages [86, 43] and the low-level assembly $\text{A+I}$ [67, 68] that we make use of.

The notions of bisimulation devised in this work adapt several insights from both Jagadeesan et al.’s fully abstract trace semantics for the $\lambda\mu$-hashref-calculus [40] and Laird’s fully abstract trace semantics for a simple $\lambda$-calculus with references [46].
8. Conclusions and Future Work

Software is playing an increasingly prominent role in all areas of society. As the importance and complexity of software systems increases, the importance of securing the software systems increases as well. In this thesis we have taken aim at the security threats inherent in multi-component software, more specifically the security issues that arise when a low-level component interacts with a component written in a high-level functional programming language.

Important prior research has come up with ways to protect programs in functional programming languages from low-level components through static approaches that enforce various limitations on low-level components. These limitations on low-level components, however, make the attacker model considered, less relevant. In this work, instead we constrain the attacker model as little as possible, assuming that it is an in-process attacker model that has gained the ability to execute arbitrary code in the address space of the functional program. This is an important attacker model, as many real-world exploits are often triggered by the injection or loading of untrusted binary code into a program’s address space.

In this thesis we have developed run-time protection techniques for functional programs against this attacker model by making use of the protected module architecture to isolate the code and data of the functional programs. We introduced these run-time techniques for three important areas: foreign function interfaces, abstract machines and compilation.

In Chapter 3 we introduced a secure foreign function interface between a functional program in the protected memory of PMA and the low-level attacker. To achieve this we introduced reference objects, modelled as names, that mask the security important terms and values as well as a novel split formalisation of language interoperation that uses stacks of (evaluation) contexts to track the interaction. This foreign function interface was proven to be secure by a proof that relates the bisimilarity relations over the source language and the FFI.

In Chapter 4 we addressed the main weakness of the foreign function interface of Chapter 3, the high-level attacker model that it uses to simplify the formalisation. We proved that this high-level attacker captures all threats that the low-level attacker poses, through a novel prove technique that consists of adapting a previous fully abstract trace semantics over the low-level attacker to the bisimilarity relation over the FFI.

In Chapter 5 we derived and implemented an abstract machine from the formalisation of our secure foreign function interface using a 4 step methodology.
The beauty of this methodology is that it comes with formal properties, whose proof ensures us that the derived machine is indeed secure.

Finally in Chapter 6 we introduced and implemented a compilation scheme that compiles a functional programming language with higher-order modules to the protected memory of the PMA. A combination of reference objects, run-time type checks and functor stapling ensure that two contextually equivalent modules compiled by our compilation scheme are indistinguishable to low-level attackers. This was proven by developing and implementing an algorithm that can produce a high-level context for every scenario where the low-level attacker distinguishes between two compiled modules.

8.1 Future Work

There are several directions for future work. One is to extend our work on foreign function interfaces, abstract machines and secure compilers by investigating support for some of the functional programming language features, that where not considered in this work, such as parametric polymorphism, general functional references and concurrency. The main challenge to this direction of future work are the applicative bisimilarities that we used to establish our fully abstract translation results. Our bisimilarity relations are not capable of capturing high-level abstractions such as parametric polymorphism as they cannot be captured in a label. Extending our results to these high-level abstractions would thus require alternative proof techniques such as logical relations or environmental bisimulations.

Another possible extension that would enrich our results in all three of the areas that we introduced run-time protection techniques, is garbage collection. A common type of garbage collection is tracing, which is done by scanning the entirety of the heap for objects that have no references pointing to them. Hence, such a garbage collector must have to have access to all of the available memory addresses, the addresses of the attacker memory and those of the protected module. This could prove to be difficult to implement as most low-level languages do not provide any support for garbage collection. The garbage collector may also inadvertently leak information about the allocation status of the protected memory to the attacker, breaking our formal security properties.
References


161


[49] Xavier Leroy. Applicative functors and fully transparent higher-order modules.


[67] Marco Patrignani and Dave Clarke. Fully Abstract Trace Semantics of


Appendix A.
Type Soundness for MiniML⁺

A.1 Completeness of the Dynamic Type checks

Lemma 1 states that:

Given \( \Gamma \vdash N \) we have that either:

1. \( \llbracket v \rrbracket_\tau \xrightarrow{\ast} \llbracket v \rrbracket_\tau \) where \( \Gamma \vdash v : \tau \)
2. \( \llbracket v \rrbracket_\tau \xrightarrow{\ast} \llbracket \text{wr} \rrbracket_\tau \)

Proof:
The proof proceeds by induction on the structure of \( \tau \).

- \( \tau = \text{Bool} \)
In this case there are two sub-cases.
  - \( v = b \)
    In this sub-case only the transition rule Marshall-In-Bool applies.
    \[ \llbracket b \rrbracket_{\text{Bool}} \xrightarrow{\ast} \llbracket b \rrbracket_{\text{Bool}} \]
    As per the Type-Boolean rule (Section 2.1) we have that thesis (1) holds:
    \[ \Gamma \vdash b : \text{Bool} \]
  - \( v \neq b \)
    In this sub-case only the transition rule Marshall-In-Bool-Error applies.
    \[ \llbracket v \rrbracket_{\text{Bool}} \xrightarrow{\ast} \llbracket \text{wr} \rrbracket_{\text{Bool}} \]
    In this case thesis (2) holds.

- \( \tau = \text{Int} \)
  Similar to the \( \tau = \text{Bool} \) case.

- \( \tau = \text{Unit} \)
  Similar to the \( \tau = \text{Bool} \) case.

- \( \tau = \text{Ref} \tau \)
  In this case two sub-cases apply.
In this sub-case there are again two sub-cases.

* \( n_j \in N \)

Here the transition rule *Marshall-In-Location* applies.

\[
\|n_j\|_{N_{\text{Ref}}}^{\tau} \rightarrow \|l\|_{N_{\text{Ref}}}^{\tau} \quad \text{where } N(n_j) = (l, \text{Ref } \tau)
\]

From the rules (*Type-Map*) and (*Type-Location*) we have that thesis (1) holds:

\[
\Gamma \vdash l : \text{Ref } \tau
\]

* \( n_j \notin N \)

In this sub-case only the transition rule *Marshall-In-Location-Error* applies.

\[
\|n_j\|_{N_{\text{Ref}}}^{\tau} \rightarrow \|\text{wr}\|_{N_{\text{Ref}}}^{\tau}
\]

In this case thesis (2) holds.

\( v = n_j^1 \)

In this sub-case there are again two sub-cases.

* \( n_j^1 \in N \)

Here the transition rule *Marshall-In-Location-Error* applies.

\[
\|n_j^1\|_{N_{\text{Ref}}}^{\tau} \rightarrow \|\text{wr}\|_{N_{\text{Ref}}}^{\tau}
\]

In this case thesis (2) holds.

\( v \neq n_j^1 \)

In this sub-case only the transition rule *Marshall-In-Location-Error* applies.

\[
\|v\|_{N_{\text{Ref}}}^{\tau} \rightarrow \|\text{wr}\|_{N_{\text{Ref}}}^{\tau}
\]

In this case thesis (2) holds.

\( \tau = \tau \rightarrow \tau' \)

In this case three sub-cases apply.

\( v = n_i^f \)

In this sub-case there are again two sub-cases.

* \( n_i^f \in N \)

Here the transition rule *Marshall-In-Lambda* applies.

\[
\|n_i^f\|_{\tau \rightarrow \tau'}^{\tau} \rightarrow \|((\lambda x : \tau. e))\|_{\tau \rightarrow \tau'}^{\tau'} \quad \text{where } N(n_i^f) = ((\lambda x : \tau. e), \tau \rightarrow \tau')
\]

From the rules (*Type-Map*) and (*Type-Lambda*) we have that thesis (1) holds:

\[
\Gamma \vdash (\lambda x : \tau. e) : \tau \rightarrow \tau'
\]

* \( n_i^f \notin N \)

In this sub-case only the transition rule *Marshall-In-Lambda-Error* applies.

\[
\|n_i^f\|_{\tau \rightarrow \tau'}^{\tau} \rightarrow \|\text{wr}\|_{\tau \rightarrow \tau'}^{\tau'}
\]

Thesis (2) holds.
\[- v = \lambda x. e \]

Here the transition rule \textit{Marshall-In-Foreign-Lambda} applies.

\[
\Vdash (\lambda x. e)_{\tau \rightarrow \tau'}^N \Rightarrow \Vdash^N \text{F}(\lambda x. e)_{\tau \rightarrow \tau'}^N
\]

From the \textit{Type-Foreign-Lambda} rule we conclude that thesis (1) holds.

\[
\Gamma \vdash \tau \rightarrow \tau' \text{F}(\lambda x. e) : \tau \rightarrow \tau'
\]

\[- v \neq n^i \text{ and } v \neq \lambda x. e \]

In this sub-case only the transition rule \textit{Marshall-In-Lambda-Error} applies.

\[
\Vdash v_{\tau \rightarrow \tau'} N \Rightarrow \Vdash \text{wr}_{\tau \rightarrow \tau'} N
\]

Thesis (2) holds.

\[\bullet \tau = \langle \tau^i_{1..n} \rangle\]

In this case two sub-cases apply.

\[- v = \langle v^i_{1..n} \rangle\]

In this sub-case two different reduction rules can apply.

* The rule \textit{Marshall-In-Tuple}:

\[
\Vdash \langle v^i_{1..n} \rangle_{\tau^i_{1..n} \rightarrow \tau^i_{1..n}}^N \Rightarrow \Vdash \langle v^i_{1..n} \rangle_{\tau^i_{1..n} \rightarrow \tau^i_{1..n}}^N
\]

We assume that the thesis holds for all subderivations and thus conclude that thesis (1) holds:

\[
\Gamma \vdash \langle v^i_{1..n} \rangle : \langle \tau^i_{1..n} \rangle
\]

* The rule \textit{Marshall-In-Tuple-Error}:

\[
\Vdash \langle v^i_{1..n} \rangle_{\tau^i_{1..n} \rightarrow \tau^i_{1..n}}^N \Rightarrow \Vdash \text{wr}_{\tau^i_{1..n} \rightarrow \tau^i_{1..n}}^N
\]

Thesis (2) holds.

\[- v \neq \langle v^i_{1..n} \rangle\]

In this sub-case only the transition rule \textit{Marshall-In-Tuple-Error2} applies.

\[
\Vdash v_{\tau^i_{1..n} \rightarrow \tau^i_{1..n}} N \Rightarrow \Vdash \text{wr}_{\tau^i_{1..n} \rightarrow \tau^i_{1..n}}^N
\]

Thesis (2) holds.

\[
\square
\]

\section{A.2 Type Preservation for MiniML$^+$}

Theorem 1, states that:

Given $\Gamma \vdash A || M$ and $A || M \Rightarrow A' || M'$ we have that: $\Gamma \vdash A' || M'$. 

169
Proof:
The proof proceeds by induction on the derivation of $\Gamma \vdash M$.

- $\Gamma \vdash N; \mu \vdash \Sigma \circ t : \tau$

  In this case there are three sub cases.

  - Internal reductions (Internal MiniML):
    
    $$\text{A} \parallel N; \mu \vdash \Sigma \circ e : \tau \rightarrow \text{A} \parallel N; \mu' \vdash \Sigma \circ e' : \tau$$

    By the fact that the internal reductions of the MiniML terms within MiniML$^+$ preserve the semantics of MiniML, we conclude the thesis.

  - Insecure Call (M-Call):
    
    $$\mu \parallel C, C \parallel N; \mu \vdash \Sigma \circ E[(\tau_1 \rightarrow \tau_2 \text{F}(\lambda x.e) v)] : \tau \rightarrow \mu \parallel C, C[\{e[\cdot]\}] \parallel N; \mu \vdash \Sigma, E : \tau_2 \rightarrow \tau \triangleright v : \tau_1$$

    It follows from the subsumed type rules of MiniML that given an application $(\lambda x : \tau_1.t) v$ where the argument type of the $\lambda$-term is a type $\tau_1$ then the argument will be of type $\tau_1$ as well. Similarly it follows from the type rules of MiniML that if the return type of the $\lambda$-term is $\tau_2$ then the result of the application is of type $\tau_2$. The open context $E$ thus type checks for an input of type $\tau_2$.

    As per rule Type-Marshalling-Out and Type-Stack we must prove that:

    $$\Gamma, x : \tau_2 \vdash E[x] : \tau$$

    It holds from the fact that MiniML$^+$ preserves the semantics of MiniML.

  - Setting up the marshalling out state (Setup):
    
    $$\text{A} \parallel N; \mu \vdash \Sigma \circ v : \tau \rightarrow \text{A} \parallel N; \mu \vdash \Sigma \triangleright v : \tau$$

    In this transition no terms and types change, we conclude the thesis.

- $\Gamma \vdash N; \mu \vdash \Sigma \triangleright m : \tau$

  In this case there are two sub cases.

  - Marshalling out reductions (Marshall-Out-*):
    
    $$\text{A} \parallel N; \mu \vdash \Sigma \triangleright m : \tau \rightarrow \text{A} \parallel N; \mu \vdash \Sigma \triangleright m' : \tau$$

    The marshalling out states are always well-typed as the reduction rules preform the type checks dynamically when outside input comes in (Lemma 1): enforcing the thesis.

    $$\Gamma \vdash N; \mu \vdash \Sigma \triangleright m' : \tau$$
– Share the result of the marshalling process (Share):

\[ \mu \vdash C, C \triangleright N; \mu \vdash \Sigma \triangleright v : \tau \Rightarrow \mu \vdash C \circ C[v] \triangleright N; \mu \vdash \Sigma \]

It follows from the assumption that \( N, \mu \) and \( \Sigma \) are type sound and thus we conclude that the thesis holds.

\[ \Gamma \vdash N; \mu \vdash \Sigma \]

• \( \Gamma \vdash N; \mu \vdash \Sigma \triangleleft m : \tau \)

In this case there are three sub cases.

– Marshalling reductions (\( Marshall-In-* \)):

\[ A \triangleright N; \mu \vdash \Sigma \triangleleft m : \tau \Rightarrow A \triangleright N; \mu \vdash \Sigma \triangleleft m' : \tau \]

The marshalling in rules preform the type checks dynamically: enforcing the thesis as proven in Lemma 1.

\[ \Gamma \vdash N; \mu \vdash \Sigma \triangleleft m' : \tau \]

– Plugging the stack (\( Plug \)):

\[ A \triangleright N; \mu \vdash \Sigma \triangleright E : \tau \rightarrow \tau' \triangleleft v : \tau \rightarrow A \triangleright N, \mu \vdash \Sigma \circ E[v] : \tau' \]

The open evaluation context \( E \) is type checked similar to how a \( \lambda \)-term is type checked. As is the case for \( \lambda \)-term application, context plugging thus holds.

\[ \Gamma \vdash N, \mu \vdash \Sigma \circ E[v] : \tau' \]

– Incorrect attacker input (\( Type-Error-In \)):

\[ \mu \vdash C, C \triangleright N; \mu \vdash \Sigma \triangleright wr : \tau \Rightarrow \mu \vdash C, C[wr] \triangleright \ast; \emptyset \vdash \epsilon \]

The type rules type check only the secure state and must thus only consider the empty state. By the rules \( Type-Passive-State \), \( Type-Store-Empty \), \( Type-Map-Empty \), \( Type-Stack-Empty \), we have that the thesis holds.

\[ \Gamma \vdash \ast; \emptyset \vdash \epsilon \]

• \( \Gamma \vdash N; \mu \vdash \Sigma \): There are 6 sub cases.

– Attacker input (\( Input \)):

\[ \mu \vdash C \circ v \triangleright N; \mu \vdash \Sigma \triangleright E : \tau \rightarrow \tau' \rightarrow \mu \vdash C \circ N; \mu \vdash \Sigma \triangleleft v : \tau \]

The type rule \( Type-Marshall-In \) does not restrict the input value \( v \), that is left to the run-time type checks. We conclude that the thesis holds.

\[ \Gamma \vdash N; \mu \vdash \Sigma \triangleleft v : \tau \]
– Calling secure functions (A-Call):

\[
\begin{align*}
\mu \models C \circ E[\text{call } n_f^i v] & \mid N; \mu \models \Sigma \\
\mu \models C \circ v & \mid N; \mu \models \Sigma, (e[\cdot]) : \tau \rightarrow \tau'
\end{align*}
\]

where \( N(n_f^i) = (e, \tau \rightarrow \tau') \)

It follows from the typing rule for the map \( N \) that:

\[ \Gamma \vdash e : \tau \rightarrow \tau' \]

where \( e \) is a \( \lambda \)-term, as in MiniML. The newly constructed context applies the \( \lambda \)-term to a hole. The hole must thus be filled in by an argument of type \( \tau \). The result will be of type: \( \tau' \). We conclude by the typing rule \( \text{Type-Passive-State} \) that the thesis holds.

– A-Set:

\[
\begin{align*}
\mu \models C \circ E[\text{set } n_l^i v] & \mid N; \mu \models \Sigma \\
\mu \models C \circ v & \mid N; \mu \models \Sigma, (e[\cdot]) : \tau \rightarrow \text{Unit}
\end{align*}
\]

where \( N(n_l^i) = (e, \text{Ref } \tau) \)

It follows from the typing rules for the map \( N \) that:

\[ \Gamma \vdash e : \text{Ref } \tau \]

as in MiniML. The newly constructed context assigns the location to a hole. The hole must thus be filled in by an argument of type \( \tau \) as per the typing rules of MiniML. An assignment reduces to \text{unit} and is thus of type \text{Unit}. We conclude by the typing rule \( \text{Type-Passive-State} \) that the thesis holds.

– A-Ref:

\[
\begin{align*}
\mu \models C \circ E[\text{ref } v] & \mid N; \mu \models \Sigma \\
\mu \models C \circ v & \mid N; \mu \models \Sigma, (\text{ref } [\cdot]) : \tau \rightarrow \text{Ref } \tau
\end{align*}
\]

In this case the attacker promises an argument of type \( \tau \) to the reference operation. This promise is checked by the marshalling rules. It follows from the MiniML typing rules that a term \( \Gamma \vdash \text{ref } e \) types to \( \text{Ref } \tau' \) where \( \tau' \) is the type of \( t \). We conclude by the typing rule \( \text{Type-Passive-State} \) that the thesis holds.

\[ \Gamma \vdash e : \text{Ref } \tau \]

– A-Der:

\[
\begin{align*}
\mu \models C \circ E[\text{deref } n_l^i] & \mid N; \mu \models \Sigma \\
\mu \models C \circ v & \mid N; \mu \models \Sigma, (\text{ref } [\cdot]) : \tau \rightarrow \text{Ref } \tau
\end{align*}
\]

It follows from the typing rule for the map \( N \) that
Γ ⊢ l_i : Ref τ
as in MiniML. It also follows from the typing rules of MiniML that
Γ ⊢ !l_1 : τ. We conclude that the thesis holds.

Γ ⊢ N; µ ⊩ Σ o !l : τ

– A-Wr*:

\[ A \| N; µ \vdash Σ \rightarrow µ \vdash C o C[wr] \| \star;θ \vdash ε \]

As was the case for the error case in the marshalling in case, we
note that the type rules type check only the secure state and thus the
empty state. By the rules Type-Passive-State, Type-Store-Empty,
Type-Map-Empty, Type-Stack-Empty, we conclude that the thesis
holds.

Γ ⊢ \star;θ \vdash ε

\[ □ \]

A.3 Type Progress for MiniML+

Theorem 2 states that: Given \( A \| \Gamma \vdash M \) then either

1. \( A \| M \rightarrow A' \| M' \)
2. \( A \| M \rightarrow A \| M' \)
3. \( A \| M \rightarrow A' \| M \)
4. \( A \| M \rightarrow µ\vdash wr \| \star;θ \vdash ε \)
5. \( µ\vdash e \| M \not\rightarrow A' \| M' \)
6. \( µ\vdash C o e \| N, µ \vdash e \not\rightarrow A' \| M' \)

Proof:

By induction on a derivation of \( Γ \vdash M \).

- \( Γ \vdash N; µ \vdash Σ o e : τ \)

There are two sub-cases.

- Foreign application: \( e = (τ \rightarrow τ_2 F(λx.t) v) \)

There are two sub cases depending on the form of the attacker state: \( A \).

* \( A = µ\vdash C, C \)

In this case the reduction rule M-Call applies.

* \( A = µ\vdash ε \)

In this case no reduction is possible, as in case (5) of the thesis.
Otherwise, it follows from the fact that MiniML\(^+\) preserves the semantics of MiniML that the term \(e\) is either a value \(v\) in which case the rule Setup applies, or else there exists a \(e'\) such that:

\[
\Gamma \vdash A \parallel N; \mu \vdash \Sigma \circ e : \tau \rightarrow A \parallel N; \mu' \vdash \Sigma \circ e' : \tau
\]

- \(\Gamma \vdash N; \mu \vdash \Sigma \triangleright m : \tau\)
  There are two sub-cases.
  - Marshalling done: \(m = v\)
    In this case there are two sub cases depending on the form of the attacker state: \(A\).
    
    \[\begin{align*}
    & * \ A = \mu \models \overline{C}, C \\
    & \text{In this case the reduction rule } Share \text{ applies.}
    \end{align*}\]
    
    \[\begin{align*}
    & * \ A = \mu \models e \\
    & \text{In this case no reduction is possible, as in case (5) of the thesis.}
    \end{align*}\]
  - Otherwise by the completeness of the marshalling rules, we have that there exists a \(m'\) such that:
    
    \[
    \Gamma \vdash A \parallel N; \mu \vdash \Sigma \triangleright m : \tau \rightarrow A \parallel N; \mu \vdash \Sigma \triangleright m' : \tau
    \]

- \(\Gamma \vdash N; \mu \vdash \Sigma \triangleright m : \tau\)
  There are three sub-cases.
  - Marshalling completed \(m = v\)
    In this case the reduction rule Plug applies.
  - Incorrect input detected: \(m = wr\)
    If the input from the attacker does not conform to the type \(\tau\) the marshalling term will reduce to \(wr\) (Lemma 1). As per reduction rule Type-Error-In we have that case (5) of the thesis.
    - Otherwise by the completeness of the marshalling rules, we have that there exists a \(m'\) such that:
      
      \[
      \Gamma \vdash A \parallel N; \mu \vdash \Sigma \triangleright m : \tau \rightarrow A \parallel N; \mu \vdash \Sigma \triangleright m' : \tau
      \]

- \(\Gamma \vdash N; \mu \vdash \Sigma\)
  In this case there are 6 sub-cases depending on the form of the attacker state \(A\).
  
  - \(A = \mu \models \overline{C} \circ v\)
    In this case the reduction rule Input applies if the secure call stack is not empty, otherwise case (6) of the thesis applies.
  - \(A = \mu \models \overline{C} \circ E[call\ n^f_v]\)
    There are two sub-cases in this sub-case.
    
    \[* \ n^f_v \in N\]
    \text{The rule A-Call applies.}
Otherwise by A-WrC we have that the program will reduce to
the error state as in case (4).

- \( A = \mu \models \overline{C} \circ \mathcal{E}\left[\text{set } n_1 \_ \text{v}\right] \)
  Similar to the \( A = \mu \models \overline{C} \circ \mathcal{E}\left[\text{call } n_1 \_ \text{v}\right] \) sub-case.

- \( A = \mu \models \overline{C} \circ \mathcal{E}\left[\text{deref } n_1 \_ \right] \)
  Similar to the \( A = \mu \models \overline{C} \circ \mathcal{E}\left[\text{call } n_1 \_ \text{v}\right] \) sub-case.

- \( A = \mu \models \overline{C} \circ \mathcal{E}\left[\text{fref} \_ \text{v}\right] \)
  In this case the MiniML\(^+\) reduction rule A-Ref applies.

- Otherwise, case (3) of the thesis applies.

\( \square \)
Appendix B.
A Fully Abstract Translation Scheme for MiniML

B.1 Completeness of $\approx^M$

Lemma 2 states that:

$$
\mu_1 \mid e_1 \approx^M \mu_2 \mid e_2 \Rightarrow \emptyset; \mu_1 \mid e_1 : \tau \approx^M \emptyset; \mu_2 \mid e_2 : \tau \text{ where } \Gamma \vdash e_1 : \tau \text{ and } \Gamma \vdash e_2 : \tau
$$

Proof:

Proving that MiniML contextual equivalence implies bisimilarity is done by showing that the contextual equivalence relation is itself a bisimulation.

Assume that:

$$
\mu_1 \mid e_1 \approx^M \mu_2 \mid e_2.
$$

Because bisimilarity is symmetrical, we divide the proof into two parts:

1. Assume that: $\emptyset; \mu_2 \mid e_2 : \tau \Rightarrow^X K'_2; \mu'_2 \mid e'_2 : \tau'$. We now establish that there exists a $K'_2; \mu'_2 \mid e'_2 : \tau'$ such that:

   a) $\emptyset; \mu_2 \mid e_2 : \tau \Rightarrow^X K'_2; \mu'_2 \mid e'_2 : \tau'$

   b) $\mu'_1 \mid e'_1 \approx^M \mu'_2 \mid e'_2$

The proof proceeds by case analysis on the label $\gamma$. For every label that decodes an action of an attacker we establish the label can be encoded as a MiniML context $C$. Note that we simplify the proof by only establishing that there exists a context that can distinguish between $e_1$ and $e_2$, to comply with the definition of contextual equivalence the context must also diverge in one case.

- true: From the LTS transition O-T it follows that $e_1$ reduces to the value true. It follows from $\mu_1 \mid e_1 \approx^M \mu_2 \mid e_2$ that $e_2$ also reduces to true (and thus produces a label true) as otherwise the context:

  $$
  C = ([\_] == \text{true})
  $$

  can distinguish between $e_1$ and $e_2$. It thus follows from O-T that $e'_1 = e'_2 = \text{true}$. It also follows from O-T that the shared location stores $K'_1 = K_1$ and $K'_2 = K_2$, as the shared location stores are unchanged by the labelled transition. It also follows from this that $\mu'_1$ and $\mu'_2$ contextually equivalent as any additional locations are not shared and any changes to already shared locations can be observed by contexts:

  $$
  \mu'_1 \mid e'_1 = \mu'_2 \mid e'_2.
  $$

176
Both theses thus hold.

- **false**: Similar to the true case.
- **unit**: Similar to the true case.
- **\(\bar{n}\)**: Similar to the true case.
- **\(l_i\)**: From the LTS transition O-L it follows that \(e_1\) reduces to \(l_i\). It follows from \(\mu_1 \mid e_1 \simeq^M \mu_2 \mid e_2\) that \(e_2\) also reduces to a location \(l_i\) (and thus produces the same label \(l_i\)) as otherwise the context:

\[
C = ([\cdot]; \bar{l_i})
\]

can distinguish between \(e_1\) and \(te2\). The shared location stores \(K\) are updated with the same location \(l_i\). Both theses thus hold.

- **\(@v\)**: From the contextual equivalence between \(e_1\) and \(e_2\) it follows that \(e_1\) and \(e_2\) are both function types and thus both reduce to \(\lambda\)-terms \(v_1\) and \(v_2\). Given that \(\gamma = @v\) it now follows from the LTS that \(e'_1 = (v_1 \, v)\) and \(e'_2 = (v_2 \, v)\). The label \(@v\) can thus be encoded as the context:

\[
C = ([\cdot] \, v)
\]

because contextual equivalence is closed under contexts we can now conclude that:

\[
\mu'_1 \mid (v_1 \, v) \simeq^M \mu'_2 \mid (v_2 \, v).
\]

- **\(.i\)**: From the contextual equivalence between \(e_1\) and \(e_2\) it follows that both \(e_1\) and \(e_2\) are both tuple types of the same length \(n\) and thus both reduce to tuples of length \(n\):

\[
\langle v_i \in 1..n \rangle \quad \text{and} \quad \langle v'_i \in 1..n \rangle,
\]

where \(\forall i \in 1..n. \, v_i \simeq v'_i\). Otherwise depending on the type of a member \(v_i\) the following context could distinguish them:

- \(v_i = \bar{n} \Rightarrow C = ([\cdot], i == \bar{n})\)
- \(v_i = b \Rightarrow C = ([\cdot], i == b)\)
- \(v_i = l_j \Rightarrow C = (\text{index}([\cdot], i == \bar{j})\)
- \(v_i = (\lambda x : \tau. \, t) \Rightarrow C = (\text{index}([\cdot], i @ v)\)
- \(v_i = \langle v_j \in 1..n \rangle \Rightarrow C = (\text{index}([\cdot], i \, j...)\)

The label \(.i\) can be encoded as the context \(C = ([\cdot], i)\), because contextual equivalence is closed under contexts we can now conclude that:

\[
\mu'_1 \mid \langle v_i \in 1..n \rangle.i \simeq^M \mu'_2 \mid \langle v'_i \in 1..n \rangle.i.
\]

- **\(!l_i\)**: It follows from l-D that \(e'_1 =!l\) and \(e'_2 =!l\) and that \(l_i \in K_1\). By \(\mu_1 \mid e_1 \simeq^M \mu_2 \mid e_2\) we have that \(K_1 = K_2\). The label \(!l_i\) can thus be encoded as the context

\[
C = ([\cdot]; !l_i),
\]

because contextual equivalence is closed under contexts we can now conclude that
\[ \mu' \mid (e_1; !l_i) \simeq^M \mu'_2(e_2; !l_i) \]

- \( l_i := v \): Similar to the \(!l\) case.
- \( \text{ref } v \): Similar to the \(!l\) case.

2. As in case 1, mutatis mutandis.

\[ \square \]

B.2 Soundness of \( \simeq^M \)

Lemma 3 states that:

\[ \Theta; \mu_1 \mid e_1 : \tau \simeq^M \Theta; \mu_2 \mid e_2 : \tau \Rightarrow \mu_1 \mid e_1 \simeq^M \mu_2 \mid e_2 \]

where \( \Gamma \vdash e_1 : \tau \) and \( \Gamma \vdash e_2 : \tau \).

**Proof:**

The thesis \( \mu_1 \mid e_1 \simeq^M \mu_2 \mid e_2 \) becomes:

\[ \forall C. \mu_1 \mid C[e_1] \uparrow \iff \mu_2 \mid C[e_2] \uparrow \]

The proof is divided into two cases, one case for each side of the co-implication.

1. \( \Rightarrow \): In this case the thesis is \( \forall C. \mu_1 \mid C[e_1] \uparrow \Rightarrow C[e_2] \uparrow \). The thesis is redefined as:

\[ \forall C. \forall k \in \mathbb{N}. \mu_1 \mid C[e_1] \rightarrow^k \mu_1' \mid C[e_1]' \Rightarrow \forall m \in \mathbb{N}. \mu_2 \mid C[e_2] \rightarrow^m \mu_2' \mid C[e_2]' \]

The proof proceeds by induction on \( m \).

**Base case:** \( m = 0 \). Straightforward: \( \mu_2 \mid C[e_2] \rightarrow^0 \mu_2 \mid C[e_2] \).

**Inductive case:** \( m = h + 1 \). The thesis is: \( \mu_2 \mid C[e_2] \rightarrow^{h+1} \mu_2' \mid C[e_2]' \).

The inductive hypotheses (IH) is:

\[ \forall C. \forall k \in \mathbb{N}. \mu_1 \mid C[e_1] \rightarrow^k \mu_1' \mid C[e_1]' \Rightarrow \mu_2 \mid C[e_2] \rightarrow^h C[e_2]^h \]

We know from this IH that:

\[ \exists C. \mu_1 \mid C[e_1] \rightarrow^h \mu_1^h \mid C[e_1]^h \rightarrow^{k-h} \mu_1' \mid C[e_1]' \]

We prove the thesis by reasoning about what the presence or absence of the last observable label \( \gamma \) tells us about the existence of a next reduction step \( h + l \). There are two cases: either the context \( C \) reduces or the term \( e \) reduces.

a) The MiniML term \( e \) is executing and the context is passive. In this case there are two sub-cases:

i. \( \exists \gamma. K_1^h; \mu_1^h \mid e_1^h : \tau^h \Rightarrow K_1'; \mu_1' \mid e_1' : \tau' \).

By the assumption \( \Theta; \mu_1 \mid e_1 : \tau \simeq^M \Theta; \mu_2 \mid e_2 : \tau \) we conclude that
As in case 1, \textit{mutatis mutandis}

b) The context \( C \)

In this case there are two sub-cases as well:

i. \( \not\exists \gamma. K^h_1; \mu^h_1 \mid v^h_1 \not\Rightarrow K^h_1; \mu^h_1 \mid e^h_1. \)

As per the definition of \( \not\Rightarrow \) we have that this is only possible if the term is diverging. By the assumption\( \theta; \mu_1 \mid e_1 : \tau \overset{M}{=} \theta; \mu_2 \mid t_2 : \tau \)
we have that both \( e_1 \) and \( e_2 \) diverge after producing the same set of labels, which implies the thesis.

ii. \( \not\exists \gamma. K^h_1; \mu^h_1 \mid v^h_1 \Rightarrow K^h_1; \mu^h_1 \mid e^h_1. \)

This, in conjunction with the IH, implies the thesis:

\[
C[e_2] \Rightarrow^{h+1} C[e_2']
\]

ii. \( \not\exists \gamma. K^h_1; \mu^h_1 \mid v^h_1 \Rightarrow K^h_1; \mu^h_1 \mid e^h_1. \)

As per the definition of \( \not\Rightarrow \) we have that this is only possible if the term is diverging. By the assumption

\[
\theta; \mu_1 \mid e_1 : \tau \overset{M}{=} \theta; \mu_2 \mid t_2 : \tau
\]
we have that both \( e_1 \) and \( e_2 \) diverge after producing the same set of labels, which implies the thesis.

b) The context \( C \) is executing and the term \( e \) is a passive value \( v \).

In this case there are two sub-cases as well:

i. \( \exists \gamma. K^h_1; \mu^h_1 \mid v^h_1 \not\Rightarrow K^h_1; \mu^h_1 \mid e^h_1. \)

Because the observable label \( \gamma \) is produced by the context, we must thus show that: \( C^h_1 = C^h_2 \), where the existence of \( C^h_2 \) derives from the induction hypothesis. We know by the assumption: \( \theta; \mu_1 \mid e_1 \overset{M}{=} \theta; \mu_2 \mid e_2 \) that both terms where modified by the same stream of observable labels if there are any such labels:

\[
\exists k \in \mathbb{N}. k \leq h \land \mu_1 \mid C[e_1] \Rightarrow^k \mu^k_1 \mid C[e_1]^k \land \mu^k_1 \mid C[e_1]^h
\]

and where

\[
K_1; \mu_1 \mid C[e_1] : \tau \Rightarrow^\gamma K^k_1; \mu^k_1 \mid C[e_1]^k : \tau^k
\]

and that

\[
\exists k \in \mathbb{N}. k \leq h \land K_2 \mu_2 \mid C[e_2] : \tau \Rightarrow^k K^k_2; \mu^k_2 \mid C[e_2]^k : \tau^k \land K^k_2; \mu^k_2 \mid C[e_2]^h : \tau^h
\]

where \( K_2; \mu_2 \mid C[e_2] : \tau \Rightarrow^\gamma K^k_2; \mu^k_2 \mid C[e_2]^k : \tau^k \).

Combining the fact that the reduction rules of MiniML are deterministic and with the fact that the MiniML-contexts are updated in the same way by identical labels \( \gamma \), we conclude that \( C^h_1 = C^h_2 \) and that

\[
K^h_2; \mu^h_2 \mid C[e_2]^h : \tau^h \Rightarrow^\gamma K^h_2; \mu^h_2 \mid C[e_2]^h : \tau^h
\]

This implies the thesis.

ii. \( \not\exists \gamma. K^h_1; \mu^h_1 \mid v^h_1 : \tau^h \Rightarrow^\gamma K^h_1; \mu^h_1 \mid e^h_1 : \tau^h. \)

If there exists no label \( \gamma \) the MiniML-context is diverging. In the previous case we established that \( C^h_1 = C^h_2 \). As such both \( C[e_1] \) and \( C[e_2] \) divergence, which implies the thesis.

2. \( \Leftarrow \) As in case 1, \textit{mutatis mutandis}. 

\( \square \)
B.3 Completeness of \( \approx^+ \)

Lemma 4 states that:

\[ M_1 \approx^+ M_2 \Rightarrow M_1 \approx^+ M_2. \]

**Proof:**

To prove that contextual equivalence implies bisimilarity we show that the contextual equivalence relation \( (\approx^+) \) is itself a bisimulation \( (S^+) \). Assume that: \( M_1 \approx^+ M_2 \). Because bisimilarity is symmetrical, we divide the proof into two parts.

1. Assume that: \( M_1 \xrightarrow{\gamma} M'_1 \). We must show that there exists a \( M_2 \) such that

   (1) \( M_2 \xrightarrow{\gamma} M'_2 \) and (2) \( M'_1 \approx^+ M'_2 \). The proof proceeds by case analysis on the labels \( \gamma^+ \). For the labels produced by the secure state \( M \) we rely on the fact that it follows from the assumption \( M_1 \approx^+ M_2 \) that the MiniML terms \( t_1 \) and \( t_2 \) reduced by both states are contextually equivalent as well and thus reduce to the same value and produce the same label. For the labels produced by the attacker state \( A \) we simply show that every label produced by the attacker can be encoded as a context \( C \), because contextual equivalence is closed under contexts that suffices to imply the thesis.

   • \( \gamma^+ = \sqrt{\_} : \) It follows from the LTS rule Done that \( M_1 = *; \mu \triangleright \varepsilon \) and that \( M'_1 = *; \theta \triangleright \varepsilon \). From the assumption \( M_1 \approx^+ M_2 \) we have that \( M_2 = *; \mu \triangleright \varepsilon \) as otherwise there exists a context \( A \) that can distinguish \( M_1 \) and \( M_2 \) as follows.

     – \( N \neq * \): In this case there is some name \( n_i \) that the attacker \( A \) can invoke as per the rules C-N and S-N.

     – \( \Sigma \neq \varepsilon \): In this case there is some input \( v \) that the secure context \( M \) accepts from the attacker \( A \) as per the rule A-V.

     – \( *; \mu \triangleright \varepsilon \circ t \mid *; \mu \triangleright \varepsilon \triangleright m \mid *; \mu \triangleright \varepsilon \triangleright m \): In this case \( M_2 \) will either produce a value as per M-V which the attacker can observe, otherwise \( M_2 \) diverges in which case the attacker distinguishes the states by default.

   It follows from the LTS that \( M_2 = *; \mu \triangleright \varepsilon \xrightarrow{\sqrt{\_}} M'_2 = *; \theta \triangleright \varepsilon \), we conclude that thesis (1) and (2) hold.

   • \( \gamma^+ = v! : \) It follows from the LTS rule M-V that \( M_1 = N; \mu \triangleright \Sigma \triangleright v \) and that \( M'_1 = N; \mu \triangleright \Sigma \). From the assumption \( M_1 \approx^+ M_2 \) we have that \( M_2 = N'; \mu' \triangleright \Sigma' \triangleright v' \) with \( v' = v \), where

   \[ \forall n_i \in \text{Dom}(N). N(n_i) \simeq N'(n_i) \quad \text{and} \quad \forall E \in \Sigma. \forall E' \in \Sigma'. E \simeq E' \]

   as otherwise there exists a context \( A \) that can distinguish \( M_1 \) and \( M_2 \) as follows.
\[M_2 = N'; \mu' \models \Sigma' \triangleright v' \land v' \neq v:\] The attacker \(A = ([·] \equiv \alpha v)\) can distinguish \(M_1\) and \(M_2\).

\(- \ M_2 \uparrow:\ M_1\) does not diverge, the attacker does distinguishes them by default.

\(- \ N \not\approx N':\) In this case there is some name \(n_i\) that the attacker \(A\) can invoke as per the rules C-N and S-N to distinguish \(M_1\) and \(M_2\).

\(- \ \Sigma \not\approx \Sigma':\) In this case there is some input \(v\) that the secure context \(M\) accepts from the attacker \(A\) as per the rule A-V that can be used to distinguish \(M_1\) and \(M_2\).

It follows from the LTS that
\[M_2 = N'; \mu' \models \Sigma' \triangleright v' \Rightarrow M'_2 = N'; \mu' \models \Sigma'\]
we conclude that thesis (1) and (2) hold.

\(- \ \gamma^+ = v?:\) It follows from the LTS rule A-V that \(M_1 = N; \mu \models \Sigma\) and that \(M'_1 = N; \mu \models \Sigma\). From the assumption \(M_1 \simeq^+ M_2\) we have that \(M_2 = N'; \mu' \models \Sigma'\) as otherwise there exists a context \(A\) that can distinguish \(M_1\) and \(M_2\). The label \(v?\) can be encoded as a context \(A = (_\circ v)\), as contextual equivalence is closed under contexts we have that for the resulting \(M'_2 = N'; \mu' \models \Sigma' \triangleleft v\) we can conclude that \(M'_1 \simeq M'_2\).

\(- \ \gamma^+ = \text{wr}:\) There are three sub cases depending on which LTS rule produces the label:

\(- \ \text{Wr-C}:\) This LTS transition captures the MiniML\(^+\) reduction rules A-WrD, A-WrS and A-WrC, each of these rules are actions by the attacker that result in the failure state. If \(M_2\) doesn’t produce the failure state for these actions the states \(M_1\) and \(M_2\) can be distinguished contradicting the assumption \(M_1 \simeq M_2\).

\(- \ \text{Wr-O}:\) If \(M_2\) does not reduce the program to the failure state the states \(M_1\) and \(M_2\) can be distinguished contradicting the assumption \(M_1 \simeq M_2\).

\(- \ \text{Wr-I}:\) similar to Wr-O.

In each of these cases as per the LTS \(M'_1 = \ast; \emptyset \models \varepsilon\) and \(M'_2 = \ast; \emptyset \models \varepsilon\), we conclude that \(M'_1 \simeq M'_2\).

\(- \ \gamma^+ = !n^i_1:\) It follows from the LTS rule D-N that \(M_1 = N; \mu \models \Sigma\) and that \(M'_1 = N; \mu \models \Sigma \circ !i\). From the assumption \(M_1 \simeq^+ M_2\) we have that \(M_2 = N'; \mu' \models \Sigma'\) as otherwise there exists a context \(A\) that can distinguish \(M_1\) and \(M_2\). The label \(v?\) can be encoded as a context \(A = (_\circ n_i)\), as contextual equivalence is closed under contexts we have that for the resulting \(M'_2 = N'; \mu' \models \Sigma' \circ !i\) we can conclude that \(M'_1 \simeq M'_2\).
• $\gamma^+ = \gg n_i^f$: It follows from the LTS rule C-N that $M_1 = N; \mu \vdash \Sigma$ and that $M'_1 = N; \mu \vdash \Sigma, (\lambda(x : \tau.e) [:])$. From the assumption $M_1 \simeq^+ M_2$ we have that $M_2 = N'; \mu' \vdash \Sigma'$ as otherwise there exists context that can distinguish $A$ that can distinguish them. It follows from the MiniML$^+$ rule A-Call that the label is always followed by an LTS transition $A-V$. As such we can encode the label as a context $A = \text{call } n_i v$. As contextual equivalence is closed under contexts we have that for the resulting $M'_2 = N'; \mu' \vdash \Sigma', (\lambda(x : \tau.e) [:]) \ll v$ we can conclude that $M'_1 \simeq M'_2$.

• $\gamma^+ = \gg n_i^t$: It follows from the LTS rule S-N that $M_1 = N; \mu \vdash \Sigma$ and that $M'_1 = N; \mu \vdash \Sigma, (l_i := [\cdot])$. From the assumption $M_1 \simeq^+ M_2$ we have that $M_2 = N'; \mu' \vdash \Sigma'$ as otherwise there exists context that can distinguish $A$ that can distinguish them. It follows from the MiniML$^+$ rule A-Set that the label is always followed by an LTS transition $A-V$. As such we can encode the label a context $A = n_i := v$. As contextual equivalence is closed under contexts we have that for the resulting $M'_2 = N'; \mu' \vdash \Sigma', (l_i := [\cdot]) \ll v$ we can conclude that $M'_1 \simeq M'_2$.

• $\gamma^+ = \gg \text{ref}^t$: similar to $\gg n_i^f$ and $\gg n_i^t$.

• $\gamma^+ = \gg (\lambda x.t)$: It follows from the LTS rule C-L that $M_1 = N; \mu \vdash \Sigma \circ E[(\tau_1 \rightarrow \tau_2 F(\lambda x.t) v)] : \tau$ and that $M'_1 = N; \mu \vdash \Sigma, E : \tau_2 \rightarrow \tau \triangleright v : \tau_1$. From the assumption $M_1 \simeq^+ M_2$ we have that $M_2 = N'; \mu' \vdash \Sigma' \circ E[(\tau_1 \rightarrow \tau_2 F(\lambda x.t) v')] : \tau$ where $v \simeq v'$ as otherwise the following context:

\[ \Lambda = (\mu \triangleright \overline{C}, (((\lambda x.t) v) \equiv a[\cdot])) \]

can distinguish between $M_1$ and $M_2$. We conclude that: $M'_2 = N'; \mu' \vdash \Sigma', E' : \tau_2 \rightarrow \tau \triangleright v : \tau_1$. and thus that the thesis $M'_1 \simeq M'_2$ holds.

2. As in case 1, mutatis mutandis.

\[ \square \]

### B.4 Soundness of $\approx^+$

Lemma 5 states that:

$M_1 \approx^+ M_2 \Rightarrow M_1 \simeq^+ M_2$

**Proof:**

As per Definition 2 we have that the thesis $M_1 \simeq^+ M_2$ becomes:

\[ 182 \]
\[ \forall A. A \parallel M_1 \uparrow \iff A \parallel M_2 \uparrow \]

The proof is divided into two cases, one case for each side of the co-implication.

1. \( \Rightarrow \) In this case the thesis is:
\[ \forall A. A \parallel M_1 \uparrow \Rightarrow A \parallel M_2 \uparrow . \]

The thesis can be redefined as:
\[ \forall A. \forall k \in \mathbb{N}. A \parallel M_1 \rightarrow^k A'_1 \parallel M'_1 \Rightarrow \forall m \in \mathbb{N}. A \parallel M_2 \rightarrow^m A'_2 \parallel M'_2 \]

The proof proceeds by induction on \( m \).

**Base case:** \( m = 0 \). Straightforward: \( A \parallel M_2 \rightarrow^0 A \parallel M_2 \).

**Inductive case:** \( m = h + 1 \). The thesis is:
\[ A \parallel M_2 \rightarrow^{h+1} A'_2 \parallel M'_2. \]

The inductive hypotheses (IH) is:
\[ \forall A. \forall k \in \mathbb{N}. A \parallel M_1 \rightarrow^k A'_1 \parallel M'_1 \Rightarrow A \parallel M_2 \rightarrow^h A'_2 \parallel M'_2 \]

We know from this IH that:
\[ \exists A. M_1. A \parallel M_1 \rightarrow^h A'_1 \parallel M'_1 \Rightarrow \forall m \in \mathbb{N}. A \parallel M_2 \rightarrow^m A'_2 \parallel M'_2 \]

We prove the thesis by reasoning about what the presence or absence of the last observable label \( \gamma^+ \) tells us about the existence of a next reduction step \( h + 1 \). There are two cases: either the attacker in MiniML\(^a\) is passive or executing.

a) The attacker is passive and the program in MiniML is executing. In this case there are two sub-cases:

i. \( \exists \gamma^+. M_1^h \stackrel{\gamma^+}{\longrightarrow} M'_1 \).

By the assumption \( M_1 \approx^+ M_2 \) we conclude that \( M_2^h \stackrel{\gamma^+}{\longrightarrow} M'_2 \) and \( M'_1 \approx^+ M'_2 \). This, in conjunction with the IH, implies the thesis:
\[ A \parallel M_2 \rightarrow^{h+1} A' \parallel M'_2. \]

ii. \( \not\exists \gamma^+. M_e_1 \parallel M'_1. \)

Per the definition of bisimulation we have that this is only possible if \( M_1^h \) diverges, more particularly the MiniML term \( t^h_1 \) that it executes diverges. It follows from the assumption that \( M_1 \approx^+ M_2 \) that \( M_1^h \approx^+ M_2^h \) and thus that: \( \not\exists \gamma^+. M_e_1 \parallel M'_1 \).

b) The attacker is executing and the MiniML program is waiting for input from the attacker. In this case there are two sub-cases as well:

i. \( \exists \gamma^+. N \models \Sigma^h \gamma^+ \Rightarrow M'_1. \)

Because the observable label \( \gamma^+ \) is produced by the respective attackers, we must thus show that: \( A^h_1 = A^h_2 \), where the existence of \( A^h_2 \) derives from the induction hypothesis.

We know by the assumption: \( M_1 \approx^+ M_2 \) that both attacker states were modified by the same stream of observable labels if there are any such labels: \( \exists k \in \mathbb{N}. k \leq h \land A \parallel M_1 \rightarrow^k \)
\[ A^k \mid M^k_1 \land A^k \mid M^k_1 \rightarrow^{h-k} A^h_{1} \mid M^h_1 \] where \( M_1 \xrightarrow{R} M_1^k \) and that \( \exists k \in \mathbb{N} : k \leq h \land A \mid M_2 \rightarrow^{k} A^k_{2} \mid M^k_2 \land A^k \mid M^k_2 \rightarrow^{h-k} \]

\[ A^k_{2} \mid M^k_{2} \] where \( M_2 \xrightarrow{R} M^k_{2} \).

Combining the fact that the reduction rules of the MiniML\(^+\) calculus are deterministic and with the fact that the MiniML\(^a\) contexts are updated in the same way by identical labels \( \gamma^+ \) we conclude that \( A^h_{1} = A^k_{2} \) and that \( N \models \Sigma^h \xrightarrow{\gamma^+} M^h_{2} \). This implies the thesis.

ii. \( \not\exists \gamma^+. N \models \Sigma^h \xrightarrow{\gamma^+} M^h_{1} \).

If there exists no label \( \gamma^+ \) the attacker is diverging. In the previous case we established that \( A^h_{1} = A^h_{2} \). As such both \( A \mid M_1 \) and \( A \mid M_2 \) diverge, which implies the thesis.

2. \( \Leftarrow \) As in case 1, \textit{mutatis mutandis}.

\[ \square \]

**B.5 Preservation of the Fully Abstract Translation**

Lemma 8 states that:

\[ \emptyset; \mu_1 \mid e_1 : \tau \approx^M \emptyset; \mu_2 \mid e_2 : \tau \Rightarrow \{ \mu_1 \mid e_1 \}^+ \approx^+ \{ e_2 \}^+ . \]

**Proof:**

We must develop a relation \( \mathcal{R} \) such that:

\[ \{ \mu_1 \mid e_1 \}^+ \mathcal{R} \{ \mu_2 \mid e_2 \}^+ \] (1)

and that for all \( M_1 \mathcal{R} M_2 \) we have that:

\[ M_1 \xrightarrow{R} M_1' \land \exists M_2'M_2 \xrightarrow{R} M_2' \Rightarrow M_1' \mathcal{R} M_2' \] (2)

\[ M_2 \xrightarrow{R} M_2' \land \exists M_1'M_1 \xrightarrow{R} M_1' \Rightarrow M_1' \mathcal{R} M_2' \] (3)

We define \( \mathcal{R} \) as \( \mathcal{R} = \mathcal{R}_0 \cup \mathcal{R}_1 \cup \mathcal{R}_2 \cup \mathcal{R}_3 \):

\[ \mathcal{R}_0 = \{(N_1; \mu_1 \models \Sigma_1, N_2; \mu_2 \models \Sigma_2) \mid \forall N_1, N_2, \Sigma_1, \Sigma_2 \text{ such that} \ N_1 \approx_N N_2 \text{ and } \text{Dom}(\mu_1) = \text{Dom}(\mu_2) \text{ and } \Sigma_1 \approx_{\Sigma} \Sigma_2 \} \]

\[ \mathcal{R}_1 = \{(N_1; \mu_1 \models \Sigma_1 \circ e_1 : \tau_1, N_2; \mu_2 \models \Sigma_2 \circ e_2 : \tau_2) \mid \forall N_1, N_2, \Sigma_1, \Sigma_2, e_1, e_2 \]

such that \( (N_1; \mu_1 \models \Sigma_1, N_2; \mu_2 \models \Sigma_2) \in \mathcal{R}_0 \) and \( \mu_1 \mid e_1 \approx^M \mu_2 \mid e_2 \)

and \( \tau_1 = \tau_2 \}

\[ \mathcal{R}_2 = \{(N_1; \mu_1 \models \Sigma_1 \triangleright m_1 : \tau_1, N_2; \mu_2 \models \Sigma_2 \triangleright m_2 : \tau_2) \mid \forall N_1, N_2, \Sigma_1, \Sigma_2, m_1, m_2, \text{ such that } (N_1; \mu_1 \models \Sigma_1, N_2; \mu_2 \models \Sigma_2) \in \mathcal{R}_0 \text{ and } m_1 \approx_m m_2 \text{ and } \tau_1 = \tau_2 \}

\[ \mathcal{R}_3 = \{(N_1; \mu_1 \models \Sigma_1 \triangleleft m_1 : \tau_1, N_2; \mu_2 \models \Sigma_2 \triangleleft m_2 : \tau_2) \mid \forall N_1, N_2, \Sigma_1, \Sigma_2, m_1, m_2, \text{ such that } (N_1; \mu_1 \models \Sigma_1, N_2; \mu_2 \models \Sigma_2) \in \mathcal{R}_0 \text{ and } m_1 \approx_m m_2 \text{ and } \tau_1 = \tau_2 \}

184
where \( \approx_N \) is defined as:
\[
N_1 \approx_N N_2 \iff \text{dom}(N_1) = \text{dom}(N_2) \land \forall n_i \in \text{dom}(N_1). N_1(n_i) \approx N_2(n_i)
\]
where \( \approx_\Sigma \) is defined as:
\[
\Sigma_1 \approx_\Sigma \Sigma_2 \iff \\
\text{For } (E_1^1, ..., E_n^1) : (\tau_1^1, ..., \tau_n^1) \text{ in } \Sigma_1 \text{ and }
(E_1^2, ..., E_n^2) : (\tau_1^2, ..., \tau_n^2) \text{ in } \Sigma_2
\text{ we have } n = n' \text{ and for every } 1 \leq i \leq n : \tau_i^1 = \tau_i^2 \text{ and }
\forall e', e''. \Gamma \vdash e' : \tau_i^1 \land \Gamma \vdash e'' : \tau_i^1 \land e' \approx e'' \Rightarrow E_i^1[e'] \approx E_i^2[e'']
\]
and where \( \approx_m \) is defined as:
\[
m_1 \approx_m m_2 \iff \\
(m_1 = t_1 \land m_2 = t_2 \land t_1 \approx t_2)
\lor (m_1 = t_1 \land m_2 = t_2 \land t_1 \equiv_\alpha t_2)
\lor (m_1 = \langle m_i^{1..n} \rangle \land m_2 = \langle m_i^{1..n} \rangle \land \forall i \in 1..n. m_i \approx m_i')
\]
We now proof the three cases.

- In case (1) we have that \( \star : 0 \vdash \epsilon : 0 \circ e_1 \mathrel{R} \star : 0 \vdash \epsilon \circ e_2 \) as we have that \( e_1 \approx^M e_2 \) from the assumption.

- In case (2) we assume \( M_1 \mathrel{R} M_2 \) and that \( M_1 \gamma^+ \Rightarrow M_1 \). We proceed by case analysis on \( \gamma^+ \).
  
  - \( \gamma^+ = v! \): By the LTS we have that:
    \[
    (M_1 \Rightarrow)N_1; \mu_1 \vdash \Sigma_1 \triangleright v : \tau \Rightarrow (M_1' \Rightarrow)N_1; \mu_1 \vdash \Sigma_1
    \]
    It follows from \( M_1 \mathrel{R} M_2 \) more specifically \( M_1 \mathrel{R} M_2 \) that:
    \[
    M_2 = N_2; \mu_2 \vdash \Sigma_2 \triangleright v : \tau
    \]
    By the LTS we have that:
    \[
    M_2 \Rightarrow v! \Rightarrow (M_2' \Rightarrow)N_2; \mu_2 \vdash \Sigma_2
    \]
    It follows from \( M_1 \mathrel{R} M_2 \) that \( N_1 \approx_N N_2 \), \( \text{Dom}(\mu_1) = \text{Dom}(\mu_2) \) and \( \Sigma_1 \approx_\Sigma \Sigma_2 \) and thus that \( M_1' \mathrel{R} M_2' \).

- \( \gamma^+ = v? \): By the LTS we have that:
    \[
    (M_1 \Rightarrow)N_1; \mu_1 \vdash \Sigma_1, E_1 : \tau_1 \rightarrow \tau_1' \vdash v \Rightarrow \\
    (M_1' \Rightarrow)N_1; \mu_1 \vdash \Sigma_1, E_1 : \tau_1 \rightarrow \tau_1' \triangleright v : \tau_1
    \]
It follows from $M_1 \Rrightarrow M_2$ more specifically $M_1 \Rrightarrow_0 M_2$ that:

$$M_2 = N_2; \mu_2 \models \Sigma_2, E_2 : \tau_2 \rightarrow \tau'_2$$

Where $N_1 \approx_N N_2$, $Dom(\mu_1) = Dom(\mu_2)$ and

$$\Sigma_1, E_1 : \tau_1 \rightarrow \tau'_1 \approx_\Sigma \Sigma_2, E_2; \tau_2 \rightarrow \tau'_2$$

from which we conclude that $\tau_1 = \tau_2$ and that $\tau'_1 = \tau'_2$. By the LTS we have that:

$$M_2 \xrightarrow{\tau_2} (M_2' =) N_2; \mu_2 \models \Sigma_2, E : \tau_2 \rightarrow \tau'_2$$

We conclude by $M_1' \Rrightarrow_3 M_2'$ that $M_1' \Rrightarrow M_2'$.

- $\gamma^+ = \checkmark$: By the LTS we have three cases:

$$\begin{align*}
(M_1 = )\ast; \mu_1 \models \varepsilon & \xrightarrow{\checkmark} (M_1' = )\ast; \emptyset \models \varepsilon
\end{align*}$$

It follows from $M_1 \Rrightarrow M_2$ more specifically $M_1 \Rrightarrow_0 M_2$ that:

$$M_2 = \ast; \mu_2 \models \Sigma_2$$

By the LTS we have that:

$$M_2 \xrightarrow{\ast; \emptyset} \emptyset \models \varepsilon$$

Given that $\ast \approx_N \ast$, $Dom(\emptyset) = Dom(\emptyset)$ and $\varepsilon \approx_\Sigma \varepsilon$ we conclude that $M_1' \Rrightarrow M_2'$.

- $\gamma^+ = \text{wr}$: By the LTS we have three cases:

1. $(M_1 = )N_1; \mu_1 \models \Sigma_1 \xrightarrow{\text{wr}} (M_1' = )\ast; \emptyset \models \varepsilon$ It follows from $M_1 \Rrightarrow M_2$ more specifically $M_1 \Rrightarrow_0 M_2$ that:

$$M_2 = N_2; \mu_2 \models \Sigma_2$$

Where $N_1 \approx_N N_2$, $Dom(\mu_1) = Dom(\mu_2)$ and $\Sigma_1 \approx_\Sigma \Sigma_2$. The LTS rule ref1l:Wr-C corresponds to the MiniML$^+$ reduction rules A-WrD, A-WrC and A-WrS. In each of these reduction rules the error state is produced by type check fails. It follows from $N_1 \approx_N N_2$ that the names of $M_1$ and $M_2$ share the same types and thus fails in the same situations.

$$M_2 \xrightarrow{\text{wr}} \ast; \emptyset \models \varepsilon$$

Given that $\ast \approx_N \ast$, $Dom(\emptyset) = Dom(\emptyset)$ and $\varepsilon \approx_\Sigma \varepsilon$ we conclude that $M_1' \Rrightarrow M_2'$.

2. $(M_1 = )N_1; \mu_1 \models \Sigma_1 \xleftarrow{\text{wr}} \tau : \varepsilon \xrightarrow{\text{wr}} (M_1' = )\ast; \emptyset \models \varepsilon$ It follows from $M_1 \Rrightarrow M_2$ more specifically $M_1 \Rrightarrow_3 M_2$ that:

$$M_2 = N_2; \mu_2 \models \Sigma_2$$
It follows from the LTS that:

\[ M_2 \xrightarrow{wr} \star; \emptyset \vdash \epsilon \]

Given that \( \star \equiv_N \star \), \( \text{Dom}(\emptyset) = \text{Dom}(\emptyset) \) and \( \epsilon \equiv_{\Sigma} \epsilon \) we conclude that \( M_1 \xrightarrow{\mathcal{R}} M_2 \).

3. \( (M_1 = \epsilon)N_1; \mu_1 \vdash \Sigma_1 \vdash \rightarrow \xrightarrow{wr} (M'_1 = \epsilon)\emptyset \vdash \epsilon \): Analogous to the previous case.

- \( \gamma^+ = \lambda n_1 \): By the LTS we have that:

\[ (M_1 = \epsilon)N_1; \mu_1 \vdash \Sigma_1 \xrightarrow{\mathcal{R}} (M'_1 = \epsilon)N_1; \mu_1 \vdash \Sigma_1 \circ !l_i : \tau_1 \]

Where \( N(n_1) = (l_i, \text{Ref } \tau) \). It follows from \( M_1 \xrightarrow{\mathcal{R}} M_2 \) more specifically \( M_1 \xrightarrow{\mathcal{R}_0} M_2 \) that:

\[ M_2 = N_2; \mu_2 \vdash \Sigma_2 \]

Where \( N_1 \equiv_N N_2 \), \( \text{Dom}(\mu_1) = \text{Dom}(\mu_2) \) and \( \Sigma_1 \equiv_{\Sigma} \Sigma_2 \). By the LTS we thus have that:

\[ M_2 \xrightarrow{\mathcal{R}} (M'_2 = \epsilon)N_2; \mu_1 \vdash \Sigma_2 \circ !l_i : \tau_2 \]

It follows from \( N_1 \equiv_N N_2 \) that \( N_1(n_1) \approx N_2(n_1) \) from which we conclude that \( !l_i \approx !l_i \) and that \( \tau_1 \approx \tau_2 \). We conclude that \( M'_1 \xrightarrow{\mathcal{R}} M'_2 \).

- \( \gamma^+ = \rightarrow \text{ref}^\tau \): By the LTS we have that:

\[ (M_1 = \epsilon)N_1; \mu_1 \vdash \Sigma_1 \xrightarrow{\rightarrow \text{ref}^\tau} (M'_1 = \epsilon)N_1; \mu_1 \vdash \Sigma_1, (\text{ref} [\cdot]): \tau \rightarrow \text{Ref } \tau \]

It follows from \( M_1 \xrightarrow{\mathcal{R}} M_2 \) more specifically \( M_1 \xrightarrow{\mathcal{R}_0} M_2 \) that:

\[ M_2 = N_2; \mu_2 \vdash \Sigma_2 \]

Where \( N_1 \equiv_N N_2 \), \( \text{Dom}(\mu_1) = \text{Dom}(\mu_2) \) and \( \Sigma_1 \equiv_{\Sigma} \Sigma_2 \). By the LTS we thus have that:

\[ M_2 \xrightarrow{\rightarrow \text{ref}^\tau} (M'_2 = \epsilon)N_2; \mu_2 \vdash \Sigma_2 (\text{ref} [\cdot]): \tau \rightarrow \text{Ref } \tau \]

By \( (\text{ref} [\cdot]): \tau \rightarrow \text{Ref } \tau \approx_{\Sigma} (\text{ref} [\cdot]): \tau \rightarrow \text{Ref } \tau \) we conclude that \( M'_1 \xrightarrow{\mathcal{R}_0} M'_2 \) and thus that the thesis: \( M'_1 \xrightarrow{\mathcal{R}} M'_2 \) holds.

- \( \gamma^+ = \rightarrow n_1^1 \): Similar to the \( \rightarrow \text{ref}^\tau \) case.

- \( \gamma^+ = \rightarrow n_1^1 \): Similar to the \( \rightarrow \text{ref}^\tau \) case.

- \( \gamma^+ = \rightarrow (\lambda x.e) \): By the LTS we have that:

\[ (M_1 = \epsilon)N_1; \mu_1 \vdash \Sigma_1 \circ E_1[\subseteq \pi \rightarrow \tau \text{F}(\lambda x.e) \triangleright] : \tau \xrightarrow{\rightarrow (\lambda x.e)} (M'_1 = \epsilon)N_1; \mu_1 \vdash \Sigma_1, E_1 : \tau_2 \rightarrow \tau \triangleright \triangleright \vdash \tau : \tau_1 \]

187
It follows from $M_1 \mathcal{R} M_2$ more specifically $M_1 \mathcal{R}_1 M_2$ that:

$$M_2 = N_2; \mu_2 \models \Sigma_2 \circ t_2$$

Where $N_1 \approx_N N_2$, $\text{Dom}(\mu_1) = \text{Dom}(\mu_2)$, $\Sigma_1 \approx \Sigma_2$ and $\mu_1 \models e_1 \approx^M \mu_2 \models e_2$.

It follows from the definition of $\{\mu \mid re\}$ that $(\tau_1 \rightarrow \tau_2 \mathcal{F}(\lambda x. e) v)$ derives from an input by the attacker. From the assumption $M_1 \mathcal{R} M_2$ it follows that $M_1$ and $M_2$ received the same inputs from the attacker and we thus have that like $e_1 = E_1[(\tau_1 \rightarrow \tau_2 \mathcal{F}(\lambda x. e) v)]$, $e_2$ was produced by an input $(\lambda x. e)$ from the attacker. We can thus rewrite $e_1$ as

$$((\lambda x : \tau_1 \rightarrow \tau_2. e'_1) \tau_1 \rightarrow \tau_2 \mathcal{F}(\lambda x. e))$$

and likewise $e_2$ as

$$((\lambda x : \tau_1 \rightarrow \tau_2. e'_2) \tau_1 \rightarrow \tau_2 \mathcal{F}(\lambda x. e)),$$

where from $\mathcal{R}_1$ we have that

$$\mu_1 \models (\lambda x : \tau_1 \rightarrow \tau_2. e'_1) \sim^M \mu_2 \models (\lambda x : \tau_1 \rightarrow \tau_2. e'_2)$$

We must now show that both $\lambda$-terms call their arguments an equal number of times with contextually equivalent arguments. It follows from the inclusion of references in MiniML that the contextually equivalent terms $(\lambda x : \tau_1 \rightarrow \tau_2. e'_1)$ and $(\lambda x : \tau_1 \rightarrow \tau_2. e'_2)$ call their higher-order argument $x$ the same number of times as otherwise an argument of the form $(\lambda x : \tau. l_1 := (+!l_1); x)$ can be used to distinguish both terms (Section 2.3.2). We thus conclude that:

$$M_2 \mathcal{R} (\lambda x. t) \Rightarrow (M'_2 =) N_2; \mu_2 \models \Sigma_2, E_2 : \tau_2 \rightarrow \tau \triangleright v : \tau_1$$

and that $M'_1 \mathcal{R} M'_2$ and thus that the thesis: $M'_1 \mathcal{R} M'_2$ holds.

* For case (3) we have that: mutatis mutandis.

\[\square\]

### B.6 Reflection of the Fully Abstract Translation

Lemma 9 states that:

$$\{\mu_1 \mid e_1\} \approx^+ \{\mu_2 \mid e_2\} \Rightarrow \emptyset; \mu_1 \models e_1 \approx^M \emptyset; \mu_2 \models e_2.$$  

**Proof:**

We prove the lemma by the contrapositive, the lemma is restated as:

$$\emptyset; \mu_1 \models e_1 \not\approx^M \emptyset; \mu_2 \models e_2 \Rightarrow \{\mu_1 \mid e_1\} \not\approx^+ \{\mu_2 \mid e_2\}$$

The proof has two cases. In the first case the bisimulation fails immediately as $\emptyset; \mu_1 \models e_1$ and $\emptyset; \mu_2 \models e_2$ produce differently labelled transitions after silent reduction. There are two different directions.

188
1. $(\emptyset; \mu_1 \mid e_1 \Rightarrow \xi'_1 \land \bar{\nu}' \cdot \emptyset; \mu_2 \mid e_2 \Rightarrow \xi'_2) \Rightarrow \\
(\{\mu_1 \mid e_1\} + \Rightarrow M'_1 \land \bar{\nu} \cdot \{\mu_2 \mid e_2\} + \Rightarrow M'_2)$.

We proceed by case analysis over the label $\gamma$:

- **true**: We have that:

  $\emptyset; \mu_1 \mid e_1 : \text{True} \rightarrow^* \emptyset; \mu'_1 \mid \text{true} : \text{True}$

  By the assumption, $\emptyset; \mu_1 \mid e_1 : \tau \not\Rightarrow^M \emptyset; \mu_2 \mid e_2 : \tau_2$ and the LTS we have that $\emptyset; \mu_2 \mid e_2 : \tau_2$ will either reduce to a value or diverge.

  - $\emptyset; \mu_2 \mid e_2 : \tau_2 \rightarrow^* \emptyset; \mu'_2 \mid v_2 : \tau_2$ where $v_2 \neq \text{true}$.

    By the reduction rules (Internal MiniML) and (Setup) and the fact that MiniML preserves the semantics of MiniML we have that:

    $\forall A.A \mid \{\mu_1 \mid e_1\} + \rightarrow^* A \mid \ast; \mu'_1 \vdash \varepsilon \triangleright \text{true} : \text{True}$

    and that

    $\forall A.A \mid \{\mu_2 \mid e_2\} + \rightarrow^* A \mid \ast; \mu_2 \vdash \varepsilon \triangleright v_2 : \tau_2$

    Where $v_2 \neq \text{true}$. We now conclude from the marshalling rules and the LTS rule (M-V) that $\{\mu_1 \mid t_1\} + \rightarrow^M M'_1$ and that $\{\mu_2 \mid e_2\} + \not\Rightarrow M'_2$

  - $\emptyset; \mu_2 \mid e_2 \uparrow$

    By the reduction rules (Internal MiniML) and (Setup) and the fact that MiniML preserves the semantics of MiniML we have that:

    $\forall A.A \mid \{\mu_1 \mid e_1\} + \rightarrow^* A \mid \ast; \mu'_1 \vdash \varepsilon \triangleright \text{true} : \text{True}$

    and that

    $\forall A.A \mid \{\mu_2 \mid e_2\} + \uparrow$

We conclude that the thesis holds.

- **false**: Analogous to the true case.

- **unit**: Analogous to the true case.

- **$\bar{n}$**: Analogous to the true case.

- **l_i**: Analogous to the true case.

- **i**: We have that:

  $\emptyset; \mu_1 \mid e_1 : \langle i \in 1..n \rangle \rightarrow^* \emptyset; \mu'_1 \mid \langle i \in 1..n \rangle : \langle i \in 1..n \rangle \Rightarrow i \mid v_i : \tau_i$

  By the assumption, $\emptyset; \mu_1 \mid e_1 : \langle i \in 1..n \rangle \not\Rightarrow^M \emptyset; \mu_2 \mid e_2 : \tau_2$ and the LTS we have that $\emptyset; \mu_2 \mid e_2 : \tau_2$ will either reduce to a value or diverge.

  - $\emptyset; \mu_2 \mid e_2 : \tau_2 \rightarrow^* \emptyset; \mu'_2 \mid v_2 : \tau_2$ where $v_2 \neq \langle i \in 1..n \rangle$.

    By the reduction rules (Internal MiniML) and (Setup) we have
that:

\[ \forall A.A \parallel \{ \mu_1 \mid e_1 \}_+ \rightarrow^* A \parallel \ast; \mu'_1 \vdash \varepsilon \triangleright (\nu_{i \in 1..n}) : \langle \tau_{i \in 1..n} \rangle \]

and that

\[ \forall A.A \parallel \{ \mu_2 \mid e_2 \}_+ \rightarrow^* A \parallel \ast; \mu_2 \vdash \varepsilon \triangleright v_2 : \tau_2 \]

Where \( v_2 \neq (\nu_{i \in 1..n}) \). We now conclude from the marshalling rules and the LTS rule (M-V) that the thesis holds.

- The divergence case is as in the \texttt{true}.

- \( @v \): We have that:

\[ \theta; \mu_1 \mid e_1 : \tau \rightarrow \tau' \rightarrow^* \theta; \mu'_1 \mid (\lambda x : \tau.\tau'_1) : \tau \rightarrow \tau' \overset{\text{\( @v \)}}{=} \theta; \mu'_1 \mid ((\lambda x : \tau.\tau'_1) v) : \tau' \]

By the assumption, \( \theta; \mu_1 \mid e_1 : \tau \rightarrow \tau' \neq \theta; \mu_2 \mid e_2 : \tau_2 \) and the LTS we have that: \( \theta; \mu_2 \mid e_2 : \tau_2 \) will either reduce to a value or diverge.

- \( \theta; \mu_2 \mid e_2 : \tau_2 \rightarrow^* \theta; \mu'_2 \mid v_2 : \tau_2 \) where \( v_2 \neq (\lambda x : \tau'.\tau'_2) \) or \( v_2 = (\lambda x : \tau'_1, \tau'_2) \) where \( \Gamma \vdash (\lambda x : \tau'.\tau'_1) : \tau' \land \Gamma \vdash (\lambda x : \tau'_1, \tau'_2) : \tau'_2 \land \tau'_1 \neq \tau'_2 \).

By the reduction rules (Internal MiniML) and (Setup) and the fact that MiniML \( ^+ \) preserves the semantics of MiniML we have that:

\[ \forall A.A \parallel \{ \mu_1 \mid e_1 \}_+ \rightarrow^* A \parallel \ast; \mu'_1 \vdash \varepsilon \triangleright (\lambda x : \tau.\tau'_1) : \tau_1 \]

and that

\[ \forall A.A \parallel \{ \mu_2 \mid e_2 \}_+ \rightarrow^* A \parallel \ast; \mu'_2 \vdash \varepsilon \triangleright v_2 : \tau_2 \]

Where \( \tau_1 \neq \tau_2 \). We now conclude from the marshalling rules and the LTS rule (M-V) that \( \{ \mu_1 \mid e_1 \}_+ \overset{\text{n}}{\Rightarrow} M'_1 \) and that \( \{ \mu_2 \mid e_2 \}_+ \overset{\text{n}}{\not\Rightarrow} M'_2 \)

- The divergence case is as in the \texttt{true}.

- \texttt{ref} \( v \): Given that the that the store \( \mu \) never runs out of space, we have that the label applies to any two halted states, contradicting the assumption.

- \( ! l_i \): By the rules of the LTS we have that \( \theta; \mu_1 \mid e_1 : \tau \overset{l_i}{\not\Rightarrow} \zeta'_i \) as \( ! l_i \) is only applicable if \( K_i \neq \emptyset \). This case thus contradicts the assumption.

- \( l_i := v \): Similar as the \( ! l_i \) case.

2. \( \theta; \emptyset \mid t_2 \overset{\beta}{\Rightarrow} \zeta'_2 \land \beta \zeta'_1 \). \( \theta; \emptyset \mid t_1 \overset{\beta}{\Rightarrow} \zeta'_1 \): Similar to case (1).
In the second case there is a sequence of MiniML context actions that result in two states where different LTS transitions apply. In this case we establish the thesis by showing that each MiniML context action can be replicated by an MiniML$^+$ attacker action. We proceed by case analysis over these actions of the MiniML context:

- $\gamma = \forall v$: By the LTS over $\approx^M$ we have that:

$$K; \mu \mid (\lambda x : \tau.e) : \tau \rightarrow \tau' \xrightarrow{\forall v} K; \mu \mid ((\lambda x : \tau.e) v) : \tau \rightarrow \tau'$$

where $\vdash v : \tau$

The LTS over $\approx^+$ can replicate this action as follows:

$$\begin{align*}
N; \mu \models \Sigma \circ : (\lambda x : \tau.e) \tau : \tau_1 \rightarrow \tau_2 \xrightarrow{\tau^*} \\
N, n^f_i \mapsto ((\lambda x : \tau.e), \tau_1 \rightarrow \tau_2); \mu \models \Sigma \triangleright n^f_i : \tau_1 \rightarrow \tau_2 \\
N, n^f_i \mapsto ((\lambda x : \tau.e), \tau_1 \rightarrow \tau_2); \mu \models \Sigma \triangleright n^f_i : \tau_1 \rightarrow \tau_2 \\
N, n^f_i \mapsto ((\lambda x : \tau.e), \tau_1 \rightarrow \tau_2); \mu \models \Sigma \triangleright n^f_i : \tau_1 \rightarrow \tau_2 \\
N, n^f_i \mapsto ((\lambda x : \tau.e), \tau_1 \rightarrow \tau_2); \mu \models \Sigma \triangleright n^f_i : \tau_1 \rightarrow \tau_2
\end{align*}$$

• $\gamma = \forall l_i$: By the LTS over $\approx^M$ we have that:

$$K; \mu \mid v : \tau \xrightarrow{l_i} K; \mu \mid !l_i : \tau' \quad \text{where } l_i \in K \text{ and } K \vdash l_i : \text{Ref } \tau'$$

Every time a location is shared in MiniML the LTS over $\approx^M$ adds it to the context knowledge base $K$. Similarly every shared location in MiniML$^+$ is added to $N$. It follows from the fact that MiniML$^+$ preserves the semantics of MiniML that every location shared by the MiniML term will be shared and stored to $N$ for $\{\mu \mid e\}_+$. The LTS over $\approx^+$ can thus replicate this action as follows:

$$N, n^f_i \mapsto (l_i, \text{Ref } \tau); \mu \models \Sigma \xrightarrow{[n^f_i]} N; \mu \models \Sigma \circ !l_i : \tau$$

• $\gamma = \text{ref } v$: By the LTS over $\approx^M$ we have that:

$$K; \mu \mid v : \tau \xrightarrow{\text{ref } v'} K; \mu \mid \text{ref } v' : \text{Ref } \tau' \quad \text{where } K \vdash v' : \tau'$$
The LTS over $\approx^+$ can replicate this action as follows:

\[
\begin{align*}
N; \mu \models \Sigma \xrightarrow{\text{ref} \tau} N; \mu \models \Sigma, (\text{ref } \cdot) : \tau \rightarrow \text{Ref } \tau \\
N; \mu \models \Sigma, (\text{ref } \cdot) : \tau \rightarrow \text{Ref } \tau \xrightarrow{\text{ref } v} N; \mu \models \Sigma, (\text{ref } v) : \tau \rightarrow \text{Ref } \tau
\end{align*}
\]

Where $\Gamma \vdash v : \tau$.

- $\gamma = l_i := v$: Similar to the $!l_i$ case.

\[\square\]
Appendix C.
Proofs Between the High-Level and Low-Level Attacker Models

C.1 Preservation of the High-Level Attacker

Proof:
Lemma 8 states the following.
\[ \{ \mu_1 \mid e_1 \} \approx \{ \mu_1 \mid e_1 \} \Rightarrow (\{ \mu_1 \mid e_1 \} , s, 0) \approx (\{ \mu_2 \mid e_2 \} , s, 0) \]
We must develop a relation \( R \) such that:
\[ (\{ \mu_1 \mid e_1 \} , s, 0) R (\{ \mu_2 \mid e_2 \} , s, 0) \]
and that for all \( (M_1, s_1, a_1) R (M_2, s_2, a_2) \) we have that:
\[ \langle M_1, s_1, a_1 \rangle \xrightarrow{L} \langle M_1', s_1, a_1' \rangle \]
and \( \exists \langle M_2', s_2, a_2' \rangle, \langle M_2, s_2, a_2 \rangle \xrightarrow{L} \langle M_2', s_2, a_2' \rangle \Rightarrow \langle M_1, s_1, a_1 \rangle R \langle M_2', s_2, a_2' \rangle \]
\[ \langle M_2, s_2, a_2 \rangle \xrightarrow{L} \langle M_2', s_2, a_2' \rangle \]
and \( \exists \langle M_1', s_1', a_1' \rangle, \langle M_1, s_1, a_1 \rangle \xrightarrow{L} \langle M_1', s_1', a_1' \rangle \Rightarrow \langle M_1', s_1', a_1' \rangle R \langle M_1', s_1', a_1' \rangle \]
We define \( R \) as \( R = R_0 \cup R_1 \cup R_2 \cup R_3 \):
\[ R_0 = \{ (N_1; \mu_1 \vdash \Sigma_1, s_1, a_1, \bar{a}_1), (N_2; \mu_2 \vdash \Sigma_2, s_2, a_2) \mid \]
\[ \forall N_1, N_2, \Sigma_1, \Sigma_2, \mu_1, \mu_2, a_1, a_2 \text{ such that } N_1 \approx_N N_2 \text{ and } \text{Dom}(\mu_1) = \text{Dom}(\mu_2) \text{ and } \Sigma_1 \approx \Sigma_2 \]
\[ \text{and } a_1 = a_2 \text{ and } s_1 = s_2 \} \]
\[ R_1 = \{ (N_1; \mu_1 \vdash \Sigma_1 \circ e_1 : \tau_1, s_1, a_1), (N_2; \mu_2 \vdash \Sigma_2 \circ e_2 : \tau_2, s_2, a_2) \mid \]
\[ \forall N_1, N_2, \Sigma_1, \Sigma_2, \mu_1, \mu_2, e_1, e_2, \tau_1, \tau_2, a_1, a_2, s_1, s_2, \text{ such that } \]
\[ (N_1; \mu_1 \vdash \Sigma_1, s_1, a_1), (N_2; \mu_2 \vdash \Sigma_2, s_2, a_2) \in R_0 \]
\[ \text{and } e_1 \approx e_2 \text{ and } \tau_1 = \tau_2 \} \]
\[ R_2 = \{ (N_1; \mu_1 \vdash \Sigma_1 \triangleright m_1 : \tau_1, s_1, a_1), (N_2; \mu_2 \vdash \Sigma_2 \triangleright m_2 : \tau_2, s_2, a_2) \mid \]
\[ \forall N_1, N_2, \Sigma_1, \Sigma_2, \mu_1, \mu_2, m_1, m_2, \tau_1, \tau_2, a_1, a_2, s_1, s_2, \text{ such that } \]
\[ (N_1; \mu_1 \vdash \Sigma_1, s_1, a_1), (N_2; \mu_2 \vdash \Sigma_2, s_2, a_2) \in R_0 \]
\[ \text{and } m_1 \approx_m m_2 \text{ and } \tau_1 = \tau_2 \} \]
\[ R_3 = \{ \langle (N_1; \mu_1 \models \Sigma_1 < m_1 : \tau_1), (s_1, \overline{a_1}) \rangle, \langle (N_2; \mu_2 \models \Sigma_2 < m_2 : \tau_2), (s_2, \overline{a_2}) \rangle \mid \forall N_1, N_2 \Sigma_1, \Sigma_2, \mu_1, \mu_2, m_1, m_2, \tau_1, \tau_2, \overline{a_1}, \overline{a_2}, s_1, s_2 \text{such that} \]

\[ (\langle (N_1; \mu_1 \models \Sigma_1), (s_1, \overline{a_1}) \rangle, \langle (N_2; \mu_2 \models \Sigma_2), (s_2, \overline{a_2}) \rangle) \in R_0 \]

and \( m_1 \approx_m m_2 \) and \( \tau_1 = \tau_2 \}

where \( \approx_N \) is defined as:

\[ N_1 \approx_N N_2 \overset{\text{def}}{=} \]

\[ \text{dom}(N_1) = \text{dom}(N_2) \quad \text{and} \quad \forall n_i \in \text{dom}(N_1). N_1(n_i) \approx N_2(n_i) \]

where \( \approx_\Sigma \) is defined as:

\[ \Sigma_1 \approx_\Sigma \Sigma_2 \overset{\text{def}}{=} \]

For \( (E_1, \ldots, E_n) : (\tau_1^1, \ldots, \tau_n^1) \) in \( \Sigma_1 \) and \( (E_1, \ldots, E_n') : (\tau_1^2, \ldots, \tau_n^2) \) in \( \Sigma_2 \)

\[ \text{we have } n = n' \text{ and for every } 1 \leq i \leq n : \tau_i^1 = \tau_i^2 \text{ and} \]

\[ \forall e', e''. \Gamma \vdash e' : \tau_i^1 \land e'' : \tau_i^1 \land e' \approx e'' \Rightarrow E_i^1[e'] \approx E_i^2[e''] \]

and where \( \approx_m \) is defined as:

\[ m_1 \approx_m m_2 \overset{\text{def}}{=} \]

\[ (m_1 = e_1 \land m_2 = e_2 \land e_1 \approx e_2) \]

\[ \lor (m_1 = w_1 \land m_2 = w_2 \land w_1 = w_2) \]

\[ \lor (m_1 = \overline{w_1} \land m_2 = \overline{w_2} \land \overline{w_1} = \overline{w_2}) \]

\[ \lor (m_1 = \text{wr} \land m_2 = \text{wr}) \]

\[ \lor (m_1 = \langle m_i^1 \rangle \land m_2 = \langle m_i^2 \rangle \land \forall i \in 1..n. m_i \approx m_i') \]

We now prove the three cases.

- In case (1) we have that \( \langle (\ast; \theta \models \varepsilon; \theta \circ e_1), s_1, \theta \rangle \not\in R \langle (\ast; \theta \models \varepsilon; \theta \circ e_2), s_2, \theta \rangle \) as \( e_1 \approx e_2 \) follows from the assumption (Theorem 5).
- In case (2) we assume that

\[ \langle M_1, s_1, \overline{a_1} \rangle \not\in \langle M_2, s_2, \overline{a_2} \rangle \]

and that

\[ \langle M_1, s_1, \overline{a_1} \rangle \not\models L \langle M_1', s_1, \overline{a_1} \rangle \]

we now proceed by case analysis on \( L \).

- \( L = \sqrt{v} \): Termination of interoperation.
  By the LTS we have five sub-cases:
  1. Transition Done:

\[ \langle (\ast; \theta \models \varepsilon), s_1, \theta \rangle \xrightarrow{\sqrt{v}} \langle (\ast; \theta \models \varepsilon), s_1, \theta \rangle \]

194
It follows from \( \langle M_1, s_1, a_1 \rangle R \langle M_2, s_2, a_2 \rangle \) more specifically \( \langle M_1, s_1, a_1 \rangle \not\sim_0 \langle M_2, s_2, a_2 \rangle \) that:

\[
\langle M_2, s_2, a_2 \rangle = \langle (\ast; \emptyset \models \Sigma_2), s_2, \emptyset \rangle
\]

where \( s_1 = s_2 \). By the LTS we have that only rule Done applies to \( \langle M_2, s_2, a_2 \rangle \) as follows:

\[
\langle (\ast; \emptyset \models \Sigma_2), s_2, \emptyset \rangle \xrightarrow{\checkmark} \langle (\ast; \emptyset \models \epsilon), s_2, \emptyset \rangle
\]

Given that \( * \equiv_N *, \ Dom(\emptyset) = Dom(\emptyset) \) and \( \epsilon \equiv_{\Sigma} \epsilon \) and \( \emptyset = \emptyset \), we conclude that

\[
\langle (\ast; \emptyset \models \epsilon), s_1, \emptyset \rangle \not\sim_0 \langle (\ast; \emptyset \models \epsilon), s_2, \emptyset \rangle
\]

2. Transition Wr-l:

\[
\langle (N_1; \mu_1 \models \Sigma_1 < \wr : \tau_1), s_1, a_1 \rangle \xrightarrow{\checkmark} \langle (\ast; \emptyset \models \epsilon), s_1, \emptyset \rangle
\]

It follows from \( \langle M_1, s_1, a_1 \rangle R \langle M_2, s_2, a_2 \rangle \) more specifically \( \langle M_1, s_1, a_1 \rangle \not\sim_1 \langle M_2, s_2, a_2 \rangle \) where \( N_1 \approx_N N_2 \), \ Dom(\mu_1) = Dom(\mu_2) \), \( \Sigma_1 \equiv_\Sigma \Sigma_2 \) and \( a_1 = a_2 \) and \( s_1 = s_2 \) and that \( \tau_1 = \tau_2 \). By the LTS we have that only rule Wr-l applies as follows:

\[
\langle (N_2; \mu_2 \models \Sigma_2 < \wr : \tau_2), s_2, a_2 \rangle \xrightarrow{\checkmark} \langle (\ast; \emptyset \models \epsilon), s_2, \emptyset \rangle
\]

Given that \( * \equiv_N *, \ Dom(\emptyset) = Dom(\emptyset) \) and \( \epsilon \equiv_{\Sigma} \epsilon \) and \( \emptyset = \emptyset \) we conclude that

\[
\langle (\ast; \emptyset \models \epsilon), s_1, \emptyset \rangle \not\sim_0 \langle (\ast; \emptyset \models \epsilon), s_2, \emptyset \rangle
\]

3. Transition Wr-MR: similar to the Wr-l case and Done case, main difference being that we use \( \not\sim_2 \) to relate the initial states and rely on the fact that the stacks \( a_1 \) and \( a_2 \) are equal and that the descriptors \( s_1 \) and \( s_2 \) are equal to prove that the same address \( a_r \) causes immediate termination in both cases.

4. Transition (Wr-MRT):

\[
\langle (N_1; \mu_1 \models \Sigma_1 \uparrow w_1 : (\tau_1^{j_{1}^{c_{1}.n_{1}}}), s_1, a_r : a_{d_1} : a_1 \rangle \xrightarrow{\checkmark} \langle (\ast; \emptyset \models \epsilon), s_1, \emptyset \rangle \text{ where } s \not\triangleright \text{ unprotectedCode}(a_{r_1}) \land s \not\triangleright \text{ unprotectedData}(a_{d_1}) \land s \not\triangleright \text{ unprotectedData}(a_{d_1} + |w_1|) \rangle
\]
It follows from $\langle M_1, s_1, \overline{a}_1 \rangle \mathcal{R} \langle M_2, s_2, \overline{a}_2 \rangle$ more specifically $\langle M_1, s_1, \overline{a}_1 \rangle \mathcal{R}_2 \langle M_2, s_2, \overline{a}_2 \rangle$ that:

$$\langle M_2, s_2, \overline{a}_2 \rangle = \langle (N_2; \mu_2 \vdash \Sigma_2 \triangleright \overline{w}_2 : \tau_2), s_2, \overline{a}_2 \rangle$$

Where $N_1 \approx_N N_2$, $Dom(\mu_1) = Dom(\mu_2)$, $\Sigma_1 \approx \Sigma_2$ and $s_1 = s_2$ and that $\tau_1^{\text{e}_1} = \tau_2$. and that the stacks are equal $a_{r_1} : a_{d_1} : \overline{a}_1 = \overline{a}_2$ and that the word sequences are equal $\overline{w}_1 = \overline{w}_2$. Given that the return and addresses addresses are the same and that the size of the word sequence is also the same we conclude that:

$$\langle (N_2; \mu_2 \vdash \Sigma_2 \triangleright \overline{w}_2 : \langle \tau_2^{\text{e}_1} \rangle), s_1, a_{r_2} : a_{d_2} : \overline{a}_2 \rangle \xrightarrow{\vdash}$$

$$(\star; \emptyset \vdash \epsilon), s_2, \emptyset) \text{ where } s /\not\vdash \text{ unprotectedCode}(a_{r_2}) \lor$$

$$s /\not\vdash \text{ unprotectedData}(a_{d_2}) \lor$$

$$s /\not\vdash \text{ unprotectedData}(a_{d_2} + |\overline{w}_2|)$$

We conclude that

$$\langle (\star; \emptyset \vdash \epsilon), s_1, \emptyset \rangle \mathcal{R}_0 \langle (\star; \emptyset \vdash \epsilon), s_2, \emptyset \rangle$$

5. Transition (Read-Tuple-Incorrect): Similar to the (Wr-MRT) case, main difference being that we use $\mathcal{R}_3$ to relate the initial states and that we rely on $w_1$ being equal $w_2$ to ensure that we read from the same address. Given that the type check fail depends on the contents of the memory of the attacker, we conclude that the same error transition applies to state $\langle M_2, s_2, \overline{a}_2 \rangle$.

6. Transition (Call-Complex-Error):

$$\langle (N_1; \mu_1 \vdash \Sigma_1 \circ E[\big\langle \tau'' \rightarrow \tau' \mid F(a_f, a_d) \big\rangle | v_1] : \tau_1), s, \overline{a} \rangle \xrightarrow{\vdash}$$

$$\langle (\star; \emptyset \vdash \epsilon), s, \emptyset \rangle$$

It follows from $\langle M_1, s_1, \overline{a}_1 \rangle \mathcal{R} \langle M_2, s_2, \overline{a}_2 \rangle$ more specifically $\langle M_1, s_1, \overline{a}_1 \rangle \mathcal{R}_1 \langle M_2, s_2, \overline{a}_2 \rangle$ that:

$$\langle M_2, s_2, \overline{a}_2 \rangle = \langle (N_2; \mu_2 \vdash \Sigma_2 \circ e_2 : \tau_2), \overline{a} \rangle$$

As explained further on for transition (M-Call) we can deduce that $e_2$ will call the same outside function $\tau'' \rightarrow \tau' \mid F(a_f, a_d)$ and that the argument $v_2$ is contextually equivalent to $v_1$ and thus also a tuple type of the same length $n$. Given that it follows from $\mathcal{R}_1$ that $s_1 = s_2$, we can conclude that the same error transition will occur.

$$\langle (N_2; \mu_2 \vdash \Sigma_2 \circ e_2 : \tau_2), \overline{a} \rangle \xrightarrow{\vdash} \langle (\star; \emptyset \vdash \epsilon), s, \emptyset \rangle$$

We now conclude the thesis:
\[(\{\star; \emptyset \vdash \varepsilon\}, s_1, \emptyset) \xrightarrow{\text{ret } a(w)} (\{\star; \emptyset \vdash \varepsilon\}, s_2, \emptyset)\]

- \(L = \text{ret } a(w)\): Returns by the attacker.

By the LTS we have two sub-cases:

1. Transition A-Ret:

\[
\begin{align*}
\langle (N_1; \mu_1 \vdash \Sigma_1, E_1 : \tau_1 \rightarrow \tau'_1), s_1, \overline{a_1} \rangle & \xrightarrow{\text{ret } a(w)\forall} \\
\langle (N_1; \mu_1 \vdash \Sigma_1, E_1 : \tau_1 \rightarrow \tau'_1 \triangleleft \overline{\tau_1}), s_1, \overline{a_1} \rangle &
\end{align*}
\]

It follows from \(\langle M_1, s_1, \overline{a_1} \rangle \xrightarrow{\text{ret } a(w)\forall} \langle M_2, s_2, \overline{a_2} \rangle\) more specifically \(\langle M_1, s_1, \overline{a_1} \rangle \xrightarrow{\text{ret } a(w)\forall} \langle M_2, s_2, \overline{a_2} \rangle\) that:

\[
\langle M_2, s_2, \overline{a_2} \rangle = \langle (N_2; \mu_2 \vdash \Sigma_2, E_2 : \tau_2 \rightarrow \tau'_2), s_2, \overline{a_2} \rangle
\]

Where \(N_1 \approx_N N_2, \text{Dom}(\mu_1) = \text{Dom}(\mu_2), s_1 = s_2, \Sigma_1, E_1 : \tau_1 \rightarrow \tau'_1 \approx_{\Sigma} \Sigma_2, E_2 \rightarrow \tau'_2 \) and thus that \(\tau_1 = \tau_2\) and \(\tau'_1 = \tau'_2\) and \(\overline{a_1} = \overline{a_2}\). By the LTS we have that many rules including A-Ret apply and can thus conclude that \(\langle M_2, s_2, \overline{a_2} \rangle\) can produce the same label:

\[
\begin{align*}
\langle (N_2; \mu_2 \vdash \Sigma_2, E_2 : \tau_2 \rightarrow \tau'_2), s_2, \overline{a_2} \rangle & \xrightarrow{\text{ret } a(w)\forall} \\
\langle (N_2; \mu_2 \vdash \Sigma_2, E_2 : \tau_2 \rightarrow \tau'_2 \triangleleft \overline{\tau_2}), s_2, \overline{a_2} \rangle &
\end{align*}
\]

Given that \(N_1 \approx_N N_2, \text{Dom}(\mu_1) = \text{Dom}(\mu_2), \Sigma_1 \approx_{\Sigma} \Sigma_2\) and \(\overline{a_1} = \overline{a_2}\) and \(s_1 = s_2\) and \(\overline{w} = \overline{w}\), we conclude that:

\[
\langle (N_1; \mu_1 \vdash \Sigma_1, E_1 : \tau_1 \rightarrow \tau'_1 \triangleleft \overline{\tau_1}), s_1, \overline{a_1} \rangle \xrightarrow{\text{ret } a(w)\forall} \langle (N_2; \mu_2 \vdash \Sigma_2, E_2 : \tau_2 \rightarrow \tau'_2 \triangleleft \overline{\tau_2}), s_2, \overline{a_2} \rangle
\]

2. Transition Wr-AR:

\[
\begin{align*}
\langle (N_1; \mu_1 \vdash \Sigma_1, E_1 : \tau_1 \rightarrow \tau'_1), s_1, \overline{a_1} \rangle & \xrightarrow{\text{ret } a(w)\forall} \\
\langle \{\star; \emptyset \vdash \varepsilon\}, s_1, \emptyset \rangle & \text{ where } s_1 \not\vdash \text{returnBack}(a)
\end{align*}
\]

It follows from \(\langle M_1, s_1, \overline{a_1} \rangle \xrightarrow{\text{ret } a(w)\forall} \langle M_2, s_2, \overline{a_2} \rangle\) more specifically \(\langle M_1, s_1, \overline{a_1} \rangle \xrightarrow{\text{ret } a(w)\forall} \langle M_2, s_2, \overline{a_2} \rangle\) that:

\[
\langle M_2, s_2, \overline{a_2} \rangle = \langle (N_2; \mu_2 \vdash \Sigma_2, E_2 : \tau_2 \rightarrow \tau'_2), s_2, \overline{a_2} \rangle
\]

Where most importantly \(s_1 = s_2\). The state \(\langle (N_2; \mu_2 \vdash \Sigma_2, E_2 : \tau_2 \rightarrow \tau'_2), s_2, \overline{a_2} \rangle\) does not enforce any restrictions on the abilities of the attacker to call the wrong return address to return values to, as such we conclude that Wr-AR applies to \(\langle (N_2; \mu_2 \vdash \Sigma_2, E_2 : \tau_2 \rightarrow \tau'_2), s_2, \overline{a_2} \rangle\) as well.

\[
\begin{align*}
\langle (N_2; \mu_2 \vdash \Sigma_2, E_2 : \tau_2 \rightarrow \tau'_2), s_2, \overline{a_2} \rangle & \xrightarrow{\text{ret } a(w)\forall} \\
\langle \{\star; \emptyset \vdash \varepsilon\}, s_2, \emptyset \rangle & \text{ where } s_2 \not\vdash \text{returnBack}(a)
\end{align*}
\]

197
Given that $\star \approx N \star$, $\text{Dom}(\emptyset) = \text{Dom}(\emptyset)$ and $\varepsilon \approx \Sigma \varepsilon$ and $\emptyset = \emptyset$, we conclude that

$$\langle (\star; \emptyset \vdash \varepsilon), s_1, \emptyset \rangle \R_0 \langle (\star; \emptyset \vdash \varepsilon), s_2, \emptyset \rangle$$

$L = \text{ret } a(w)!$: By the LTS we have that this label applies only to the transition M-Ret:

$$\langle (N_1; \mu_1 \vdash \Sigma_1 \triangleright w_1 : \tau_1), s_1, a_{r_1} : a_{d_1} : \overline{a_1} \rangle \xrightarrow{\text{ret } a_{r_1}(w_1)!} \langle (N_1; \mu_1 \vdash \Sigma_1), s_1, \overline{a_1} \rangle$$

It follows from $\langle M_1, s_1, \overline{a_1} \rangle \mathcal{R} \langle M_2, s_2, \overline{a_2} \rangle$ more specifically $\langle M_1, s_1, \overline{a_1} \rangle \mathcal{R}_2 \langle M_2, s_2, \overline{a_2} \rangle$ that:

$$\langle M_2, s_2, \overline{a_2} \rangle = \langle (N_2; \mu_2 \vdash \Sigma_1 \triangleright w_2 : \tau_2), s_2, a_{r_2} : a_{d_2} : \overline{a_2} \rangle$$

Where $N_1 \approx_N N_2$, $\text{Dom}(\mu_1) = \text{Dom}(\mu_2)$, $s_1 = s_2$, $\Sigma_1 \approx \Sigma \Sigma_2$, $\overline{a_1} = \overline{a_2}$, $a_{r_1} = a_{r_2}$, $a_{d_1} = a_{d_2}$ and that $w_1 = w_2$ and $\tau_1 = \tau_2$. By the LTS we have that only M-Ret applies and that it will produce the same label:

$$\langle (N_2; \mu_2 \vdash \Sigma_1 \triangleright w_2 : \tau_2), s_2, a_{r_2} : a_{d_2} : \overline{a_2} \rangle \xrightarrow{\text{ret } a_{r_2}(w_1)!} \langle (N_2; \mu_2 \vdash \Sigma_2), s_2, \overline{a_2} \rangle$$

Given that $N_1 \approx_N N_2$, $\text{Dom}(\mu_1) = \text{Dom}(\mu_2)$, $s_1 = s_2$, and $\overline{a_1} = \overline{a_2}$, we conclude that

$$\langle (N_1; \mu_1 \vdash \Sigma_1), s_1, \overline{a_1} \rangle \R_0 \langle (N_2; \mu_2 \vdash \Sigma_2), s_2, \overline{a_2} \rangle$$

$L = \text{write}(a,w).\text{ret } a'(w)!$: By the LTS we have that this label applies only to the transition (M-RetT):

$$\langle (N_1; \mu_1 \vdash \Sigma_1 \triangleright \overline{w_1} : \langle \tau_{i \in 1..n} \rangle), s_1, a_{r_1} : a_{d_1} : \overline{a} \rangle \xrightarrow{\text{write}(a_{r_1}(\overline{w_1}),\text{ret } a_{r_1}!) \text{write}(a_{d_1}(\overline{w_1}))} \langle (N_1; \mu_1 \vdash \Sigma_1), s_1, \overline{a_1} \rangle$$

It follows from $\langle M_1, s_1, \overline{a_1} \rangle \mathcal{R} \langle M_2, s_2, \overline{a_2} \rangle$ more specifically $\langle M_1, s_1, \overline{a_1} \rangle \mathcal{R}_2 \langle M_2, s_2, \overline{a_2} \rangle$ that:

$$\langle M_2, s_2, \overline{a_2} \rangle = \langle (N_2; \mu_2 \vdash \Sigma_1 \triangleright \overline{w_2} : \tau_2), s_2, a_{r_2} : a_{d_2} : \overline{a_2} \rangle$$

Where $N_1 \approx_N N_2$, $\text{Dom}(\mu_1) = \text{Dom}(\mu_2)$, $s_1 = s_2$, $\Sigma_1 \approx \Sigma \Sigma_2$, $\overline{a_1} = \overline{a_2}$, $a_{r_1} = a_{r_2}$, $a_{d_1} = a_{d_2}$ and that $\overline{w_1} = \overline{w_2}$ and $\langle \tau_{i \in 1..n} \rangle = \overline{\tau_2}$. By the LTS we have that only the rule (M-RetT) applies and that it will produce the same label:

$$\langle (N_2; \mu_2 \vdash \Sigma_1 \triangleright \overline{w_2} : \tau_2), s_2, a_{r_2} : a_{d_2} : \overline{a_2} \rangle \xrightarrow{\text{write}(a_{d_1}(\overline{w_1}),\text{ret } a_{r_1}!) \text{write}(a_{d_1}(\overline{w_1}))} \langle (N_2; \mu_2 \vdash \Sigma_2), s_2, \overline{a_2} \rangle$$

Given that $N_1 \approx_N N_2$, $\text{Dom}(\mu_1) = \text{Dom}(\mu_2)$, $\Sigma_1 \approx \Sigma \Sigma_2$, $s_1 = s_2$, and $\overline{a_1} = \overline{a_2}$, we conclude that
\[ \langle (N_1; \mu_1 \models \Sigma_1), s_1, \overline{a_1} \rangle \Rightarrow_0 \langle (N_2; \mu_2 \models \Sigma_2), s_2, \overline{a_2} \rangle \]

\(- L = \text{read}(a, w) \cdot \delta: \) By the LTS we have that this label applies only to the transition (Read-Tuple):

\[ \langle (N_1; \mu_1 \models \Sigma_1), s_1 \triangleleft a_{d_1} : \langle \tau_i \in 1..n \rangle, s_1, \overline{a_1} \rangle \xrightarrow{\text{read}(a_{d_1}, w) \cdot \delta!} \langle M''_1, s_1, \overline{a''_1} \rangle \]

It follows from \( \langle M_1, s_1, \overline{a_1} \rangle \Rightarrow \langle M_2, s_2, \overline{a_2} \rangle \) and \( \langle M_1, s_1, \overline{a_1} \rangle \Rightarrow_3 \langle M_2, s_2, \overline{a_2} \rangle \) that:

\[ \langle M_2, s_2, \overline{a_2} \rangle = \langle (N_2; \mu_2 \models \Sigma_2 \triangleleft a_{d_2} : \tau_2), s_2, \overline{a_2} \rangle \]

Where \( N_1 \approx_N N_2, \text{Dom}(\mu_1) = \text{Dom}(\mu_2), s_1 = s_2, \Sigma_1 \approx \Sigma_2, \overline{a_1} = \overline{a_2}, a_{d_1} = a_{d_2} \) and that \( \langle \tau_i \in 1..n \rangle = \tau_2 \). The read words \( w \) depend on the unprotected memory, whose contents are dependent on the attacker. Given the same input words \( w \), the question is now whether or not the same label \( \delta! \) will be generated subsequently by the intermediate state

\[ \langle (N_2; \mu_2 \models \Sigma'_2 \circ E_2[v_2] : \tau'_2), s'_2, \overline{a_2} \rangle, \]

the state that results from successfully marshalling in the tuple \( v_2 \). This intermediate state is bisimilar to the intermediate state of the first state

\[ \langle (N_2; \mu_2 \models \Sigma'_1 \circ E_1[v_1] : \tau'_1), s'_2, \overline{a_2} \rangle, \]

It follows from the fact that \( \tau_1 = \text{tau}_2 = \langle \tau_i \in 1..n \rangle, s_1 = s_2 \) and \( N_1 \approx_N N_2 \) that the marshalling in rules will produce two contextually equivalent tuples \( v_1 \) and \( v_2 \). By the definition of \( \approx_\Sigma \) and \( \Sigma_1 \approx_\Sigma \Sigma_2 \) we now have that:

\[ E_2[v_2] \approx E_1[v_1] \]

We can now conclude that:

\[ \langle (N_2; \mu_2 \models \Sigma'_1 \circ E_1[v_1] : \tau'_1), s'_2, \overline{a_2} \rangle \Rightarrow_1 \]

\[ \langle (N_2; \mu_2 \models \Sigma'_1 \circ E_1[v_1] : \tau'_1), s'_2, \overline{a_2} \rangle \]

By induction we now conclude that:

\[ \langle (N_2; \mu_2 \models \Sigma_2 \triangleleft a_{d_1} : \langle \tau_i \in 1..n \rangle, s_2, \overline{a_2} \rangle \xrightarrow{\text{read}(a_{d_1}, w) \cdot \delta!} \langle M''_2, s_2, \overline{a''_2} \rangle \]

Where \( \langle M''_1, s_1, \overline{a''_1} \rangle \Rightarrow_0 \langle M''_2, s_2, \overline{a''_2} \rangle \).

\(- L = \text{call} a(\overline{w})?: \) By the LTS we have 5 sub-cases:

1. Transition A-Start:

\[ \langle (N_1; \mu_1 \models \Sigma_1 \circ e_1 : \tau_1), s_1, \emptyset \rangle \xrightarrow{\text{call} a(a_r, a_d)\?} \]

\[ \langle (N_1; \mu_1 \models \Sigma_1 \circ e_1 : \tau_1), s_1, a_r : a_d : \emptyset \rangle \]
It follows from \( \langle M_1, s_1, \alpha_1 \rangle \# \langle M_2, s_2, \alpha_2 \rangle \) more specifically \( \langle M_1, s_1, \alpha_1 \rangle \#_1 \langle M_2, s_2, \alpha_2 \rangle \) that:

\[
\langle M_2, s_2, \alpha_2 \rangle = \langle (N_2; \mu_2 \vdash \Sigma_2, E_2 \circ e_2 : \tau_2), s_2, \emptyset \rangle
\]

Where \( N_1 \approx_N N_2, Dom(\mu_1) = Dom(\mu_2), \Sigma_1 \approx_\Sigma \Sigma_2, s_1 = s_2, e_1 \simeq e_2 \) and \( \tau_1 = \tau_2 \). By the LTS we have that transition \( A\text{-Start} \) applies to \( \langle M_2, s_2, \alpha_2 \rangle \) given that \( \alpha_2 = \emptyset \).

\[
\langle M_2, s_2, \alpha_2 \rangle = \langle (N_2; \mu_2 \vdash \Sigma_2, E_2 \circ e_2 : \tau_2), s_2, \emptyset \rangle \xrightarrow{\text{call } a_r(a_d)} \langle (N_2; \mu_2 \vdash \Sigma_2 \circ e_2 : \tau_2), s_2, a_r : a_d : \emptyset \rangle
\]

Given that \( \langle a_r : a_d : \emptyset \rangle = \langle a_r : a_d : \emptyset \rangle \), we conclude that:

\[
\langle (N_1; \mu_1 \vdash \Sigma_1 \circ e_1 : \tau_1), s_1, a_r : a_d : \emptyset \rangle \#_1 \langle (N_2; \mu_2 \vdash \Sigma_2 \circ e_2 : \tau_2), s_2, a_r : a_d : \emptyset \rangle
\]

2. Transition (\( A\text{-Set} \)):

\[
\langle (N_1; \mu_1 \vdash \Sigma_1), s_1, \alpha_1 \rangle \xrightarrow{\text{call } a_r(a_d)} \langle (N; \mu \vdash \Sigma, \langle l_i := [:] \rangle : \tau_1 \rightarrow \text{Unit} \downarrow w : \tau_2), s_1, a_r : a_d : \alpha_1 \rangle
\]

where \( N(w_n) = \langle l_i, \text{Ref } \tau_1 \rangle \)

It follows from \( \langle M_1, s_1, \alpha_1 \rangle \# \langle M_2, s_2, \alpha_2 \rangle \) more specifically \( \langle M_1, s_1, \alpha_1 \rangle \#_0 \langle M_2, s_2, \alpha_2 \rangle \) that:

\[
\langle M_2, s_2, \alpha_2 \rangle = \langle (N_2; \mu_2 \vdash \Sigma_2), s_2, \alpha_2 \rangle
\]

Where \( N_1 \approx_N N_2, Dom(\mu_1) = Dom(\mu_2), s_1 = s_2, \Sigma_1, E_1 : \tau_1 \rightarrow \tau'_1 \approx_\Sigma \Sigma_2, E_2 : \tau_2 \rightarrow \tau'_2 \) and thus that \( \tau_1 = \tau_2 \) and \( \tau'_1 = \tau'_2 \) and \( \alpha_1 = \alpha_2 \). From the LTS it follows that the same transition \( A\text{-Set} \) applies to \( \langle M_2, s_2, \alpha_2 \rangle \).

\[
\langle (N_2; \mu_2 \vdash \Sigma_2), s_2, \alpha_2 \rangle \xrightarrow{\text{call } a_r(a_d)} \langle (N_2; \mu_2 \vdash \Sigma_2, \langle l_j := [:] \rangle : \tau_2 \rightarrow \text{Unit} \downarrow w : \tau_2), s_2, a_r : a_d : \alpha_2 \rangle
\]

where \( N(w_n) = \langle l_j, \text{Ref } \tau_2 \rangle \)

By the fact that \( N_1 \approx_N N_2 \), it follows that \( l_j \simeq l_i \) and thus that \( \tau_2 = \tau_1 \). Because MiniML incorporates the \textbf{index} operator we have that \( j = i \) and thus that \( \langle l_j := [:] \rangle \simeq \langle l_i := [:] \rangle \). We conclude that:

\[
\langle (N; \mu \vdash \Sigma, \langle l_i := [:] \rangle : \tau_1 \rightarrow \text{Unit} \downarrow w : \tau_1), s_1, a_r : a_d : \alpha_1 \rangle \#_3
\]

\[
\langle (N_2; \mu_2 \vdash \Sigma_2, \langle l_j := [:] \rangle : \tau_2 \rightarrow \text{Unit} \downarrow w : \tau_2), s_2, a_r : a_d : \alpha_2 \rangle
\]
5. Transition (A-Ref):

\[
\langle (N_1; \mu_1 \vdash \Sigma_1), \overline{a}_1 \rangle \xrightarrow{\text{call } a(w, w, a_r, a_d) \,!} \\
\langle (N_1; \mu_1 \vdash \Sigma_1, (\text{ref } [\cdot]) : \tau \rightarrow \text{Ref } \tau < w : \tau), a_r : a_d : \overline{a}_1 \rangle
\]

where \( \text{convt}(w) = \tau_1 \)

It follows from \( \langle M_1, s_1, \overline{a}_1 \rangle \mathcal{R} \langle M_2, s_2, \overline{a}_2 \rangle \) more specifically \( \langle M_1, s_1, \overline{a}_1 \rangle \mathcal{R}_0 \langle M_2, s_2, \overline{a}_2 \rangle \) that:

\[
\langle M_2, s_2, \overline{a}_2 \rangle = \langle (N_2; \mu_2 \vdash \Sigma_2), s_2, \overline{a}_2 \rangle
\]

Where \( N_1 \approx_N N_2, \text{Dom}(\mu_1) = \text{Dom}(\mu_2), \Sigma_1 \approx_\Sigma \Sigma_2 \) and \( \overline{a}_1 = \overline{a}_2 \). By the LTS we have that several transitions apply including (rule A-Ref):

\[
\langle (N_2; \mu_2 \vdash \Sigma_2), s_2, \overline{a}_2 \rangle \xrightarrow{\text{call } a(w, w, a_r, a_d) \,!} \\
\langle (N_2; \mu_2 \vdash \Sigma_2, (\text{ref } [\cdot]) : \tau \rightarrow \text{Ref } \tau < w : \tau), a_r : a_d : \overline{a}_2 \rangle
\]

where \( \text{convt}(w) = \tau \)

We can now conclude from the fact that

\[
\langle N_1; \mu_1 \vdash \Sigma_1, (\text{ref } [\cdot]) : \tau \rightarrow \text{Ref } \tau \approx_\Sigma \Sigma_2, (\text{ref } [\cdot]) : \tau \rightarrow \text{Ref } \tau \rangle
\]

that as requested by the thesis:

\[
\langle (N_1; \mu_1 \vdash \Sigma_1, (\text{ref } [\cdot]) : \tau \rightarrow \text{Ref } \tau < w : \tau), a_r : a_d : \overline{a}_1 \rangle \mathcal{R}_3
\]

\[
\langle (N_2; \mu_2 \vdash \Sigma_2, (\text{ref } [\cdot]) : \tau \rightarrow \text{Ref } \tau < w : \tau), a_r : a_d : \overline{a}_2 \rangle
\]

6. Transition Wr-AC: similar to the Wr-AR case, only difference being that the equal descriptors \( s_1 \) and \( s_2 \) reject the same address \( a \) if it’s not one of the functionality entry points.

- \( L = \text{call } a(w)! \): Direct calls by secure state. By the LTS we have that only Transition (M-Call) applies:

\[
\langle (N_1; \mu_1 \vdash \Sigma_1 \circ E_1 [[(\tau \rightarrow \! \tau F(a_f, a_d) \, v_1) : \tau], \overline{a}_1] \xrightarrow{\text{call } a_f(w) \,!} \\
\langle (N_1; \mu_1 \vdash \Sigma_1, E_1 : \tau' \rightarrow \tau), \overline{a}_1 \rangle
\]

where \( \downarrow \|v_1\|_N = w \)

It follows from \( \langle M_1, s_1, \overline{a}_1 \rangle \mathcal{R} \langle M_2, s_2, \overline{a}_2 \rangle \) more specifically \( \langle M_1, s_1, \overline{a}_1 \rangle \mathcal{R}_1 \langle M_2, s_2, \overline{a}_2 \rangle \) that:

\[
\langle M_2, s_2, \overline{a}_2 \rangle = \langle (N_2; \mu_2 \vdash \Sigma_2 \circ e_2 : \tau_2), \overline{a}_2 \rangle
\]
Where $N_1 \approx_N N_2$, $Dom(\mu_1) = Dom(\mu_2)$, $\Sigma_1 \approx_\Sigma \Sigma_2$, $\alpha_1 = \alpha_2$, $\tau_1 = \tau_2$ and $e_2 \approx E_1[\tau' \mapsto F(a_f, a_d) v_1)]$.

It follows from the fact that the bisimulation starts from a state $\langle \{ \mu \mid e \}^+, s, \emptyset \rangle$, where $e$ is a pure MiniML term, that the term $\tau' \mapsto F(a_f, a_d)$ derives from the attacker. It now follows from the fact that $E_1[\tau' \mapsto F(a_f, a_d) v_1] \approx e_2$ that $e_2$ will call the same outside function $\tau' \mapsto F(a_f, a_d)$ the same number of times with contextually equivalent arguments ($v_1 \approx v_2$), as otherwise a MiniML context such as the one discussed in the first example of Section 2.3.2 can be used to distinguish between the MiniML terms $e_1$ and $e_2$. As contextually equivalent MiniML terms will be marshalled into equal low-level values $w$, we conclude that $\langle M_2, s_2, \alpha_2 \rangle$ will produce the same label $L$ as follows.

$$\langle (N_2; \mu_2 \vdash \Sigma_2 \circ e_2 : \tau_2), \alpha_2 \rangle \xrightarrow{\text{call } \alpha_f(w)!} \langle (N_2; \mu_2 \vdash \Sigma_2, E_2 : \tau' \to \tau), \alpha_2 \rangle$$

We now conclude the thesis:

$$\langle (N_1; \mu_1 \vdash \Sigma_1, E_1 : \tau' \to \tau), \alpha_1 \rangle \mathcal{R}_0$$

$$\langle (N_2; \mu_2 \vdash \Sigma_2, E_2 : \tau' \to \tau), \alpha_2 \rangle$$

$L = \overline{\text{write } (a_d, w). \text{call } a_r!}$: By the LTS we have that this label applies only to the transition (Call-Complex):

$$\langle (N_1; \mu_1 \vdash \Sigma_1 \circ E[\tau^1 \mapsto F(a_f, a_d) v_1]) : \tau_1), s, \alpha \rangle \xrightarrow{\text{write } (a_d, w). \text{call } a_r!} \langle (N_1; \mu_1 \vdash \Sigma_1, E : \tau' \to \tau_1), s, \alpha_1 \rangle$$

It follows from $\langle M_1, s_1, \alpha_1 \rangle \mathcal{R} \langle M_2, s_2, \alpha_2 \rangle$ more specifically $\langle M_1, s_1, \alpha_1 \rangle \mathcal{R}_1 \langle M_2, s_2, \alpha_2 \rangle$ that:

$$\langle M_2, s_2, \alpha_2 \rangle = \langle (N_2; \mu_2 \vdash \Sigma_2 \circ e_2 : \tau_2), \alpha_2 \rangle$$

As was the case for (M-Call) we can deduce that $e_2$ will call the same outside function $(\tau^1 \mapsto F(a_f, a_d)$ and that the argument $v_2$ is contextually equivalent to $v_1$ and thus also a tuple type of the same length $n$. By the LTS and the fact that $s_1 = s_2$ we have that only the rule (Call-Complex) applies and that it will produce the same label:

$$\langle (N_2; \mu_2 \vdash \Sigma_2 \circ e_2 : \tau_2), \alpha_2 \rangle \xrightarrow{\text{write } (a_d, w). \text{call } a_r!} \langle (N_2; \mu_2 \vdash \Sigma_2, E_2 : \tau' \to \tau_2), s, \alpha_2 \rangle$$
The remaining challenge is now whether or not the resulting name maps are bisimilar. We have that $N_2 \approx_N N_1$ from $R_1$, we also have that the tuples are also contextually equivalent $v_1 \simeq v_2$. The members of the tuples that update $N_1$ and $N_2$ are thus also contextually equivalent, allowing us to conclude that:

$$N'_2 \approx_N N'_1$$

from which we can conclude the thesis:

$$\langle (N'_1; \mu_1 \models \Sigma_1, E_1 : \tau' \rightarrow \tau_1), s, \overline{a_1} \rangle \mathcal{R}_0$$

$$\langle (N'_2; \mu_2 \models \Sigma_2, E_2 : \tau' \rightarrow \tau_2), s, \overline{a_2} \rangle$$

- For case (3) the proof is similar to, mutatis mutandis, to the proof of case (2).

□

C.2 Reflection of the Low-Level Attacker

Proof:

Lemma 9 states the following.

$$\langle \{ \mu_1 | e_1 \} +, s, \emptyset \rangle \approx^l \langle \{ \mu_2 | e_2 \} +, s, \emptyset \rangle \Rightarrow \{ \mu_1 | e_1 \} + \approx^+ \{ \mu_1 | e_1 \} +$$

Where $s$ is a valid memory descriptor. We prove the lemma by the contrapositive:

$$\{ \mu_1 | e_1 \} + \not\approx^+ \{ \mu_2 | e_2 \} + \Rightarrow \langle \{ \mu_1 | e_1 \} +, s, \emptyset \rangle \not\approx^l \langle \{ \mu_2 | e_2 \} +, s, \emptyset \rangle$$

The proof has two cases. In the first case the bisimulation fails immediately as the embedded MiniML expressions $e_1$ and $e_2$ produce differently labelled transitions after silent reductions.

1. $(\exists \gamma. \{ \mu_1 | e_1 \} + \Rightarrow M'_1 \land \not\exists M'_2, \{ \mu_2 | e_2 \} + \Rightarrow M'_2)$ implies:

$$\langle \{ \mu_1 | e_1 \} +, s, \emptyset \rangle \xrightarrow{\text{call}(a_r, a_d)} \langle \{ \mu_1 | e_1 \} +, s, a_r : a_d \theta \rangle \xrightarrow{L} \langle M'_1, s, \overline{a'_1} \rangle$$

$$\land \not\exists (M'_2, s, \overline{a'_2})$$

$$\langle \{ \mu_2 | e_2 \} +, s, \emptyset \rangle \xrightarrow{\text{call}(a_r, a_d)} \langle \{ \mu_2 | e_2 \} +, s, a_r : a_d \theta \rangle \xrightarrow{L} \langle M'_2, s, \overline{a'_2} \rangle$$

We proceed by case analysis over the label $\gamma$. Only label applies $\gamma = v!$ as $e_1$ and $e_2$ are two well typed MiniML terms that either diverge or reduce to a value $v$ that is then marshalled out as MiniML $v$ value $v$.

We can expand the assumption $\{ \mu_1 e_1 \} + \not\approx^+ \{ \mu_2 | e_2 \} +$ as:

$$\ast; \emptyset \models \varepsilon \circ e_1 : \tau \quad \xrightarrow{\tau}^{*} N_1; \mu_1 \models \varepsilon \triangleright v_1 : \tau$$

$$\langle N_1; \mu_1 \models \varepsilon \triangleright v_1 : \tau \rangle \xrightarrow{v_1!} N_1; \mu_1 \models \varepsilon$$

203
were either (a) \( \{ \mu_2 \mid e_2 \}^+ \) diverges

\[
\star; \emptyset \vdash \epsilon \circ t_2 : \tau \uparrow
\]

or (b) \( \{ \mu_2 \mid e_2 \}^+ \) reduces and marshalls to a value \( v_2 \), where \( v_1 \neq v_2 \).

\[
\star; \emptyset \vdash \epsilon \circ e_2 : \tau \quad \tau \rightarrow^* N_2; \mu_2 \vdash \epsilon \triangleright v_2 : \tau
\]

\[
N_2; \mu_2 \vdash \epsilon \triangleright v_2 : \tau \quad \rightarrow^! v_2 N_2; \mu_2 \vdash \epsilon
\]

The low-level bisimulation over \( \langle \{ \mu_1 \mid e_1 \}^+, s, \emptyset \rangle \) will produce the labels \( \text{ret } a_r(w_1)! \) or \( \sqrt{\ldots} \) if \( v_1 \) is not a tuple, and the labels \( \text{write}(a_d,w).\text{ret } a_r! \) or \( \sqrt{\ldots} \) otherwise. At the low-level each of these labels are produced after the low-level attacker calls the start entry point.

\[
\langle (\star; \emptyset \vdash \epsilon \circ e_1 : \tau), s, \emptyset \rangle \xrightarrow{\text{call } (a_r,a_d)} \langle (\star; \emptyset \vdash \epsilon \circ e_1 : \tau), s, a_r : a_d : \emptyset \rangle
\]

The labels \( \sqrt{\ldots} \) are produced by the attacker inputting incorrect addresses \( a_r \) and \( a_d \) in its start call and can thus not be the label \( L \) that distinguishes between them.

If (a) \( \{ \mu_2 \mid e_2 \}^+ \) diverges, so does \( \langle \{ \mu_2 \mid e_2 \}^+, s, \emptyset \rangle \) after the attacker calls the entry point.

\[
\langle (\star; \emptyset \vdash \epsilon \circ e_2 : \tau), s, \emptyset \rangle \xrightarrow{\text{call } (a_r,a_d)} \langle (\star; \emptyset \vdash \epsilon \circ e_2 : \tau), s, a_r : a_d : \emptyset \rangle
\]

If (b) \( \{ \mu_2 \mid e_2 \}^+ \) does not diverge, we must show that the different value \( v_2 \) that it reduces to can be related to different traces being produced at the low-level. The different MiniML\( ^a \) values \( v_1 \) and \( v_2 \) will marshall to different low-level words \( w \) or different sequences \( \overline{w} \), we thus have that: \( \not\exists \langle M'_2, s, \overline{a}_2' \rangle \) such that:

\[
\langle \{ \mu_2 \mid e_2 \}^+, s, \emptyset \rangle \xrightarrow{\text{call } (a_r,a_d)} \langle \{ \mu_2 \mid e_2 \}^+, s, a_r : a_d : \emptyset \rangle
\]

\[
\xrightarrow{\text{ret } a_r(w_1)!} \langle \{ \mu_2 \mid e_2 \}^+, s, \overline{a}_2' \rangle
\]

when \( e_1 \) does not reduce to a tuple and

\[
\langle \{ \mu_2 \mid e_2 \}^+, s, \emptyset \rangle \xrightarrow{\text{call } (a_r,a_d)} \langle \{ \mu_2 \mid e_2 \}^+, s, a_r : a_d : \emptyset \rangle
\]

\[
\xrightarrow{\text{write}(a_d,w).\text{ret } a_r!} \langle \{ \mu_2 \mid e_2 \}^+, s, \overline{a}_2' \rangle
\]

when \( e_1 \) does reduce to a tuple.
2. \((\exists \gamma. \{\mu_2 \mid e_2\} \rightarrow \equiv \text{M}_2 \land \not\equiv \text{M}_1', \{\mu_1 \mid e_1\} \rightarrow \equiv \text{M}_1')\) implies:

\[(\exists L. (\{\mu_2 \mid e_2\} \rightarrow \equiv \text{M}_2 \land \not\equiv \text{M}_1', \{\mu_1 \mid e_1\} \rightarrow \equiv \text{M}_1') \rightarrow \langle \{\mu_2 \mid e_2\}, s, a_r : a_d \theta \rangle \stackrel{L}{\rightarrow} \langle \text{M}_2', s, a_r \rangle'.\]

The proof for this case is similar, mutatis mutandis, to the proof of the previous case.

In the second case there is a sequence of high-level attacker MiniML\(^a\) actions that result in two states where different LTS transitions apply. In this case we establish the thesis by showing that each MiniML\(^a\) action can be replicated by an assembly language attacker. We proceed by case analysis over the actions of the MiniML\(^a\) attacker:

- \(\gg n^f_i: \) by the LTS for \(\approx^+\) we have that:

  \(\text{N}; \mu \models_\Sigma \gg n^f_i \rightarrow \text{N}; \mu \models_\Sigma (e [\cdot]) : \tau \rightarrow \tau' \quad \text{where} \quad \text{N}(n^f_i) = (e, \tau \rightarrow \tau')\)

  It follows from the semantics of MiniML\(^a\) that this transition is always followed by an input by the attacker.

  \(\text{N}; \mu \models_\Sigma (e [\cdot]) : \tau \rightarrow \tau' \quad \text{where} \quad \text{N}(n^f_i) = (e, \tau \rightarrow \tau') \quad \text{and} \quad s \vdash \text{ApplyEntryPoint}(a)\)

  The assembly language attacker replicates this action as follows:

  \(\langle (\text{N}; \mu \models_\Sigma), s, \overline{a} \rangle \rightarrow \text{call a(w_n, w, a_r, a_d)} \rightarrow \langle (\text{N}; \mu \models_\Sigma), s, a_r : a_d \theta \rangle \)

  where \(\text{N}(w_n) = (e, \tau \rightarrow \tau')\).

- \(!n^f_i: \) by the LTS for \(\approx^+\) we have that:

  \(\mu \models_\Sigma \exists C \circ \text{E}[\text{deref n}^f_i] \quad \text{where} \quad \text{N}(n^f_i) = (l_i, \text{Ref } \tau)\)

  The assembly language attacker replicates this action as follows:

  \(\langle (\text{N}; \mu \models_\Sigma), e, \overline{a} \rangle \rightarrow \text{call a(w_n, a_r, a_d)} \rightarrow \langle (\text{N}; \mu \models_\Sigma), e, a_r : a_d \theta \rangle \)

  where \(\text{N}(w_i) = (l_i, \text{Ref } \tau)\) and \(s \vdash \text{DerefEntryPoint}(a)\)

- \(\gg n^f_i: \) similar to the \(!n^f_i\) case.

- \(\gg \text{ref}^f: \) by the LTS for \(\approx^+\) we have that:

  \(\text{N}; \mu \models_\Sigma \gg \text{ref}^f \rightarrow \text{N}; \mu \models_\Sigma (\text{ref } [\cdot]) : \tau \rightarrow \text{Ref } \tau\)
Again, it follows from the semantics of MiniML\textsuperscript{a} that this transition is always followed by an input by the attacker.

\[
N; \mu \models \Sigma_i (e \, [];) : \tau \rightarrow \tau' \xrightarrow{\mathcal{V}} \Sigma_i (\text{ref} \, [];) : \tau \rightarrow \text{Ref} \, \tau \triangleleft \mathcal{V} : \tau
\]

The assembly language attacker replicates this action as follows:

\[
\langle (N; \mu \models \Sigma), \mathcal{a} \rangle \xrightarrow{\text{call } a(w_t, w_r, a_d)?} \langle (N; \mu \models \Sigma, (\text{ref} \, ]); \tau \rightarrow \text{Ref} \, \tau \triangleleft w : \tau), a_r : a_d : \mathcal{a} \rangle
\]

where \( \text{convt}(w_t) = \tau \) and \( s \models \text{AllocEntryPoint}(a) \)

\[\square\]

12. **Lars Thofelt**: Studies on leaf temperature recorded by direct measurement and by thermography. 1975.


16. **Lennart Källström**: The significance of biotin and certain monosaccharides for the growth of Aspergillus niger on rhamnose medium at elevated temperature. 1977.


19. **Thomas C. Amu**: Diffusion in Dilute Solutions: An Experimental Study with Special Reference to the Effect of Size and Shape of Solute and Solvent Molecules. 1978.


30. **Christopher Juhlin**: Seismic Attenuation, Shear Wave Anisotropy and Some Aspects of Fracturing in the Crystalline Rock of the Siljan Ring Area, Central Sweden. 1990.

Prior to January 1994, the series was called Uppsala Dissertations from the Faculty of Science.

Acta Universitatis Upsaliensis
Uppsala Dissertations from the Faculty of Science and Technology
Editor: The Dean of the Faculty of Science

69. **Fredrik Robelius**: Giant Oil Fields – The Highway to Oil. Giant Oil Fields and their Importance for Future Oil Production. 2007.
70. **Anna Davour**: Search for low mass WIMPs with the AMANDA neutrino telescope. 2007.
71. **Magnus Ågren**: Set Constraints for Local Search. 2007.
77. **Samson Keleta**: Double Pion Production in $d d \rightarrow \alpha \pi$ Reaction. 2008.
82. **Arvid Pohl**: Search for Subrelativistic Particles with the AMANDA Neutrino Telescope. 2009.
84. **Karin Schönning**: Meson Production in $p d$ Collisions. 2009.
85. **Henrik Petrén**: $\eta$ Meson Production in Proton-Proton Collisions at Excess Energies of 40 and 72 MeV. 2009.
88. **Sophie Grape**: Studies of PWO Crystals and Simulations of the $\bar{p} p \rightarrow \Lambda \Lambda, \Lambda \Sigma^0$ Reactions for the PANDA Experiment. 2009.
100. Patrik Adlarson. Studies of the Decay $\eta \rightarrow \pi^+ \pi^- \pi^0$ with WASA-at-COSY. 2012.