# A High-Level Model for an Assembly Language Attacker by Means of Reflection

Adriaan Larmuseau[1], Marco Patrignani[2], and Dave Clarke[1,2]

[1] Uppsala University, Sweden,
first.last@it.uu.se
[2] iMinds-Distrinet, K.U. Leuven, Belgium
first.last@cs.kuleuven.be

**Abstract.** Many high-level functional programming languages are compiled to or interoperate with, low-level languages such as C and assembly. Research into the security of these compilation and interoperation mechanisms often makes use of high-level attacker models to simplify formalisations. In practice, however, the validity of such high-level attacker models is frequently called into question. In this paper we formally prove that a light-weight ML equipped with a reflection operator can serve as an accurate model for malicious assembly language programs, when reasoning about the security threats such an attacker model poses to the abstractions of ML programs that reside within a protected memory space. The proof proceeds by relating bisimulations over the assembly language attacker and the high-level attacker.

## 1 Introduction

High-level functional programming languages such as ML and Haskell offer programmers numerous security features through abstractions such as type systems, module systems and encapsulation primitives. Motivated by speed and memory efficiency, these high-level functional programming languages are often compiled to low-level target languages such as C and assembly [7] or extended with Foreign Function Interfaces (FFIs) that enable interoperation with these low-level target languages [2]. The security features of these low-level languages, however, rarely coincide with those of functional languages. In practice, the high-level programs are often compromised by low-level components and/or libraries that may be written with malicious intent or susceptible to code injection attacks.

Accurately modeling the impact that such malicious low-level code has on high-level programs is rather challenging, as the semantics of low-level code differs greatly from that of high-level functional programming languages. As an alternative, high-level models that capture the capabilities of certain low-level attackers have been introduced. Jagadeesan *et al.* [3], for example, make use of a $\lambda$-calculus extended with low-level memory access operators to model a low-level attacker within a memory with randomized address spaces. The validity of such high-level models for low-level attackers is, however, often called into question.

In this paper we present $\mathcal{L}^a$, a high-level attacker model derived directly from a source language $\mathcal{L}$ by removing type safety and adding a reflection operator.

Our claim in previous works [5] has been that this attacker model accurately captures the threats posed by an assembly language attacker to the abstractions of a source language $\mathcal{L}$, when the programs of that language reside within a protected memory space. This protected memory space is provided by the Protected Module Architecture (PMA) [19]. PMA is a low-level memory isolation mechanism, that protects a certain memory area by restricting access to that area based on the location of the program counter. PMA will be supported in a future generation of commercial processors [10]. Our high-level model of the threats that the assembly language attacker, residing outside of the protected memory, poses to the abstractions of programs residing within the protected memory, is thus bound to be useful for many different practical applications.

In what follows, we prove that $\mathcal{L}^a$, despite being simple to derive and formalise, is an accurate model of this assembly language attacker. We do so for an example source language MiniML: a light-weight ML featuring references and recursion, from which we derive a $\mathcal{L}^a$ attacker model MiniML$^a$. The proof technique proceeds as follows: first we develop a notion of bisimulation over the interactions between the high-level attacker MiniML$^a$ and programs in the source language MiniML. Next we develop a notion of bisimulation over the interactions between the assembly language attacker and programs in MiniML by adopting the labels of a previously developed fully abstract trace semantics for the attacker model [11]. Finally, we establish our result by proving that the latter bisimulation is a full abstraction of the former and vice versa.

The remainder of this paper is organised as follows. Firstly the paper introduces the assembly language attacker and its high-level replacement (Section 2). Secondly it details the example source language MiniML, the derived attacker model MiniML$^a$ and the bisimulation over MiniML$^a$ (Section 3). Next, the paper introduces a bisimulation over the assembly language attacker (Section 4) and then presents a proof of full abstraction between both bisimulations (Section 5). Finally the paper presents related work (Section 6) and concludes (Section 7).

## 2 Security Overview

This section presents the security-relevant notions of this paper. Firstly it details the PMA enhanced low-level machine model and the associated assembly language attacker (Section 2.1). Then it details contextual equivalence: the formalism used to reason about the abstractions of high-level programming languages as well as the threats that attackers pose to them (Section 2.2). Lastly we introduce our high-level attacker model $\mathcal{L}^a$, for which we prove further on in this paper, that it captures all threats that the low-level attacker poses to the contextual equivalence of a source language $\mathcal{L}$ (Section 2.3).

### 2.1 PMA and the Assembly Language Attacker

Our low-level attacker is a standard 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. For a full formalisation of these instructions and their operational semantics we refer the interested reader to Patrignani and Clarke's formalisation [11].

To enable the development of secure applications, for this paper the development of secure programs in MiniML, this machine model has been enhanced with the Protected Module Architecture (PMA). PMA is a fine-grained, program counter-based, memory access control mechanism that divides memory into a protected memory module and unprotected memory [13]. 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 below.

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

A variety of PMA implementations exist. While current implementations of PMA are still research prototypes [13], Intel is developing a new instruction set, referred to as SGX, that will enable the usage of PMA in future commercially available processors [10].

*The attacker* 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 above, PMA is a program counter-based mechanism, which the kernel-level code injection capabilities of this attacker model cannot bypass [13].

### 2.2 Contextual Equivalence

As detailed in Section 1 our interest in the assembly language attacker of Section 2.1, revolves around the security threat this attacker poses to the abstractions of programs that reside within a protected memory space. We formally reason about this threat by means of *contextual equivalence*, as is often the case in this research field [12]. Contextual equivalence (also known as observational equivalence) provides a notion of observation of the behaviour of a program and states when two programs exhibit the same observable behaviour. Only what

can be observed by the context is of any relevance, and this changes from language to language, since different languages have different levels of abstractions. Languages that feature many strong abstractions will thus produce a larger set of contextually equivalent programs then those languages that do not.

Informally, a context $C$ is a program with a single hole $[\cdot]$ that can be filled with a program $P$, generating a new program $C[P]$. For example, if $P$ is a $\lambda$-calculus expression $\lambda x.x$, a context is another $\lambda$-calculus expression with a hole, such as $((\lambda y.y)[\cdot])$. Two programs $P_1$ and $P_2$ are said to be contextually equivalent if and only if there exists no context $C$, that can distinguish between the two programs. Contextual equivalence is formalised as follows.

**Definition 1. Contextual equivalence** $(\simeq)$ *is defined as:*

$$P_1 \simeq P_2 \stackrel{def}{=} \forall C.\ C[P_1]\Uparrow \iff C[P_2]\Uparrow$$

where $\Uparrow$ denotes divergence [12].

From our security based perspective, contexts model malicious attackers that interoperate with a program $P$ and attack it. Consider, for example, the following two higher-order $\lambda$-terms:

$$(a)\ (\lambda x.(x\ 2) + (x\ 2)) \qquad (b)\ (\lambda x.(x\ 2) * 2) \tag{Ex-1}$$

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 that includes references these two terms are, however, not equivalent as the following context/attack can distinguish between them.

$$\texttt{let}\ r = (\texttt{ref}\ 0)\ \texttt{in}\ ([\cdot]\ (\lambda y.r := !r + 1; y));\ \texttt{if}\ !r = 2\ \texttt{then}\ \Omega\ \texttt{else}\ 1$$

Applying $\lambda$-term (a) will result in divergence as the reference $r$ will be increased twice, whereas applying $\lambda$-term (b) will not. The above is thus considered a successful attack against the implementation details of the two $\lambda$-terms.

Our low-level assembly-language attacker of Section 2.1 poses an incredibly strong threat to the contextual equivalences of any source language $\mathcal{L}$ as it can compare and manipulate any sequence of bits it has access to. When interoperating with the $\lambda$-terms of Ex-1 our low-level attacker could thus distinguish them by doing a bit-wise comparison on their memory encodings.

### 2.3 The High-Level Attacker Model $\mathcal{L}^a$

Our high-level attacker model $\mathcal{L}^a$ aims to accurately model the threats posed by the assembly-language attacker to the contextual equivalences of a source language $\mathcal{L}$, whose programs reside in the protected memory space of PMA. 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 the high-level, but accurate, attacker model $\mathcal{L}^a$.

*Transformation 1: removal of type safety* Type safety forces programs to preserve types and avoids stuck states. Removing the typing rules of $\mathcal{L}$ ensures that $\mathcal{L}^a$ has no such restrictions.

*Transformation 2: introduction of reflection* The assembly language attacker is not constrained by the source level restrictions of any programming language as it can inspect and manipulate any sequence of bits it has access to. To replicate this observational power we apply an insight from Wand [15], who discovered that the inclusion of reflection into a programming language renders all abstractions and associated source level restrictions meaningless.

*Transformation 3: limit 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 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.

In all of our experimentations 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 an $\mathcal{L}^a$ style attacker from a purely functional $\lambda$-calculus, for example, because, as illustrated in Ex-1 of Section 2.2, function calls are not observable there.

## 3 A Bisimulation over the High-Level Attacker

To prove the accuracy of the $\mathcal{L}^a$ attacker models in a general manner would require a proof technique capable of reasoning over all source languages. This not being possible, we instead introduce an example source language MiniML (Section 3.1), for which we derive an instance of our $\mathcal{L}^a$ attacker model denoted as MiniML$^a$ (Section 3.2). Next, we model the interactions between MiniML and the high-level attacker MiniML$^a$ by applying our previously developed interoperation semantics [5], resulting in a combined calculus MiniML$^+$ (section 3.3). Lastly a bisimulation $\mathcal{B}^a$ that captures the observations and inputs of the high-level MiniML$^a$ attacker is derived over the semantics of this MiniML$^+$ (Section 3.4). Later on, in Section 5, this bisimulation is related to a bisimulation $\mathcal{B}^l$ over the observations and inputs of the assembly-language attacker (Section 4), to prove the accuracy of the high-level MiniML$^a$ attacker.

In what follows, the source language MiniML is typeset in a `black font`, The attacker model MiniML$^a$ is typeset in a **bold red font**.

### 3.1 The Source Language MiniML

The source language is MiniML: an extension of the typed $\lambda$-calculus featuring constants, references and recursion. The syntax is as follows.

$$\begin{array}{rl} t & ::= v \mid x \mid (t_1\ t_2) \mid t_1\ \texttt{op}\ t_2 \mid \texttt{if}\ t_1\ t_2\ t_3 \mid \texttt{ref}\ t \mid t_1 := t_2 \mid t_1; t_2 \\ & \mid \texttt{let}\ x = t_1\ \texttt{in}\ t_2 \mid !t \mid \texttt{fix}\ t \mid \texttt{hash}\ t \mid \texttt{letrec}\ x : \tau = t_1\ \texttt{in}\ t_2 \end{array}$$

$$
\begin{array}{lll}
\texttt{op} & ::= & + \mid - \mid * \mid < \mid > \mid == \\
v & ::= & \texttt{unit} \mid l_i \mid \overline{n} \mid (\lambda x : \tau . t) \mid \texttt{true} \mid \texttt{false} \\
\tau & ::= & \texttt{Bool} \mid \texttt{Int} \mid \texttt{Unit} \mid \tau_1 \rightarrow \tau_2 \mid \texttt{Ref}\ \tau \\
\texttt{E} & ::= & [\cdot] \mid \texttt{E}\ t \mid v\ \texttt{E} \mid \texttt{op}\ \texttt{E}\ t \mid \texttt{op}\ v\ \texttt{E} \mid \texttt{if}\ \texttt{E}\ t_2\ t_3 \mid \ldots
\end{array}
$$

Here $\overline{n}$ indicates the syntactic term representing the number n, $\tau$ denotes the types and E is a Felleisen-Hieb-style evaluation context with a hole $[\cdot]$ that lifts the basic reduction steps to a standard left-to-right call-by-value semantics [1]. The `letrec` operator is syntactic sugar for a combination of `let` and `fix`. The operators `op` apply only to booleans and integers. Locations $l_i$ are an artefact of the dynamic semantics that do not appear in the syntax used by programmers and are tracked at run-time in a store $\mu ::= \emptyset \mid \mu, l_i = v$. Allocating new locations is done deterministically $l_1, .., l_n$. The term `hash` $t$ maps a location to its index: $l_i \mapsto \bar{i}$, similar to how Java's *.hashCode* method converts references to integers.

The reduction and type rules are standard and are thus omitted. The interested reader can find a full formalisation of the semantics of MiniML in a companion technical report [6].

### 3.2 The High-Level Attacker Model MiniML$^a$

We now apply the three transformations specified for $\mathcal{L}^a$ to MiniML, resulting in a new calculus MiniML$^a$: the high-level attacker.

*Transformation 1: removal of type safety* Removing type safety is a straightforward transformation. The types and type checking rules of MiniML are removed from the formalism and a new term **wr** is introduced that captures non reducible expressions such as the following one:

$$\mu \mid \mathbf{E[if\ v\ t_2\ t_3]} \longrightarrow \mu \mid \mathbf{E[wr]} \qquad \textit{where } \mathbf{v} \neq \mathbf{true} \textit{ or } \mathbf{v} \neq \mathbf{false}$$

where $\mu$ is the run-time store of MiniML$^a$. While capturing the stuck states of the attacker is not required, removing them from the semantics does significantly simplify proofs over the attacker model without reducing its effectiveness.

*Transformation 2: introduce reflection* The most important feature of the $\mathcal{L}^a$ attacker model is the inclusion of a reflection operator, as it renders the abstractions and the associated source level restrictions of a language meaningless [15]. Reflection is added to MiniML$^a$ by means of a syntactic equality testing operator modulo $\alpha$-equivalence $\equiv_\alpha$. It enables a program in MiniML$^a$ to compare the syntax of any two terms as follows.

$$\frac{\mathbf{t_1}\ \textit{and}\ \mathbf{t_2}\ \textit{are}\ \alpha\textit{-equiv}}{\mu \mid \mathbf{E[t_1 \equiv_\alpha t_2]} \longrightarrow \mu \mid \mathbf{E[true]}} \qquad \frac{\mathbf{t_1}\ \textit{and}\ \mathbf{t_2}\ \textit{are not}\ \alpha\textit{-equiv}}{\mu \mid \mathbf{E[t_1 \equiv_\alpha t_2]} \longrightarrow \mu \mid \mathbf{E[false]}}$$

*Transformation 3: limit control flow* MiniML enforces an evaluation order through the evaluation contexts E (Section 3.1). The $\alpha$-equivalence testing operator $\equiv_\alpha$ works around this enforced control flow, by not reducing its sub-terms to values.

*Attacks in MiniML$^a$* While MiniML$^a$ is clearly not a low-level code formalism,

it does capture 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 [15]. Consider, for example, the following two contextually equivalent MiniML terms.

$$(\lambda x : \texttt{Int}.(+\ x\ x)) \qquad (\lambda x : \texttt{Int}.(*\ 2\ x)) \tag{Ex-2}$$

There exists no context/attack in MiniML that can distinguish these two terms. The following MiniML$^a$ context, however, is an attack against this equivalence.

$$\mathbf{C} = (\mathbf{if}\ ((\lambda \mathbf{y}.(*\ \mathbf{2}\ \mathbf{y}))\ \equiv_\alpha\ [\cdot])\ \mathbf{\Omega}\ \mathbf{true})$$

The context distinguishes the two equivalent terms due to the $\equiv_\alpha$ operator's ability to inspect the syntax of MiniML terms, where $\mathbf{\Omega}$ is a diverging MiniML$^a$ term. A similar context $\mathbf{C}$ can thus be built for every pair of contextually equivalent terms in MiniML apart from $\alpha$-equivalent terms.

### 3.3 MiniML$^+$: Interoperation between MiniML$^a$ and MiniML

To accurately capture the inputs and observations of the high-level attacker we must first introduce a formalism for its interactions with programs in MiniML. To do so we apply our previously developed language interoperation semantics [5]. While there exists many different multi-language semantics (Section 6), our interoperation semantics is the only one that supports separated program states and explicit marshalling rules. The former is required to accurately capture the behaviour of the attacker, the latter is used to simplify and streamline the transition to the low-level attacker model in Section 4.

Concretely the MiniML$^+$-calculus combines the attacker model MiniML$^a$ and the source language MiniML by defining separated program states, specifying marshalling rules, encoding cross boundary function calls through call stacks and sharing data structures through reference objects.

*Separated program states* The program state $P = \mathbf{A} \mid\mid \mathtt{S}$ of MiniML$^+$ is split into two sub-states: an attacker state $\mathbf{A}$ and a MiniML program state $\mathtt{S}$. The reduction rules for MiniML$^+$ programs are denoted as follows: $\mathbf{A} \mid\mid \mathtt{S} \twoheadrightarrow \mathbf{A}' \mid\mid \mathtt{S}'$.

The MiniML state $\mathtt{S}$ is either (1) executing a term $t$ 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.

$$(1)\ \mathtt{N}; \mu \Vdash \Sigma \circ t : \tau \quad (2)\ \mathtt{N}; \mu \Vdash \Sigma \rhd m : \tau \quad (3)\ \mathtt{N}; \mu \Vdash \Sigma \lhd m : \tau \quad (4)\ \mathtt{N}; \mu \Vdash \Sigma$$

where $m = \mathbf{v} \mid v$ as the marshalling rules convert MiniML values to MiniML$^a$ values, and vice versa. The attacker state takes two forms: (1) it executes a MiniML$^a$ term $\mathbf{t}$ or (2) is suspended waiting on input from the MiniML program.

$$(1)\ \mathbf{A} = \mu \Vdash \overline{\mathbf{C}} \bullet \mathbf{t} \qquad (2)\ \mathbf{A} = \mu \Vdash \overline{\mathbf{C}}$$

The states never compute concurrently. Whenever the MiniML state $\mathtt{S}$ computes, the attacker state $\mathbf{A}$ is suspended and vice-versa.

*Marshalling* Marshalling converts the result of MiniML programs to MiniML$^a$

values and inputs from the MiniML$^a$ attacker to MiniML values. Marshalling, booleans for example, is done as follows.

$$\mathbf{A} \;||\; \mathtt{N}; \mu \Vdash \Sigma \lhd \mathbf{b} : \mathtt{Bool} \twoheadrightarrow \mathbf{A} \;||\; \mathtt{N}; \mu \Vdash \Sigma \lhd b : \mathtt{Bool} \qquad \text{(In-B)}$$

$$\mathbf{A} \;||\; \mathtt{N}; \mu \Vdash \Sigma \rhd b : \mathtt{Bool} \twoheadrightarrow \mathbf{A} \;||\; \mathtt{N}; \mu \Vdash \Sigma \rhd \mathbf{b} : \mathtt{Bool} \qquad \text{(Out-B)}$$

Note that when marshalling, the typing information encoded in the MiniML state is used to ensure that the input does not violate MiniML typing rules.

*Call stacks* To ensure that the program state is separable, the combined language must explicitly encode the depth of the interactions between MiniML and the attacker MiniML$^a$. To do so each state is extended with a call stack. The MiniML state $\mathtt{S}$ encodes this call stack as a type annotated stack of evaluation contexts $\Sigma ::= \overline{\mathtt{E} : \tau \to \tau'} \mid \varepsilon$, where $\overline{\mathtt{E}}$ denotes a sequence of evaluation contexts $\mathtt{E}$ 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, these types are incorporated into the dynamic type checks of the marshalling rules to ensure that the input from the attacker does not break type safety.

In contrast the attacker encodes the call stack through a sequence of contexts/attacks $\overline{\mathbf{C}}$, enabling it to attack each interaction with the MiniML program. The attacker stack $\overline{\mathbf{C}}$ is updated directly (Share), the MiniML stack $\Sigma$ is plugged by the result of the marshalling rules (Plug), as follows.

$$\mu \Vdash \overline{\mathbf{C}}, \mathbf{C} \;||\; \mathtt{N}; \mu \Vdash \Sigma \rhd \mathbf{v} : \tau \twoheadrightarrow \mu \Vdash \overline{\mathbf{C}} \bullet \mathbf{C}[\mathbf{v}] \;||\; \mathtt{N}; \mu \Vdash \Sigma \qquad \text{(Share)}$$

$$\mu \Vdash \overline{\mathbf{C}} \;||\; \mathtt{N}; \mu \Vdash \Sigma, \mathtt{E} : \tau \to \tau' \lhd v : \tau \twoheadrightarrow \mu \Vdash \overline{\mathbf{C}} \;||\; \mathtt{N}', \mu \Vdash \Sigma \circ \mathtt{E}[\mathrm{v}] : \tau' \qquad \text{(Plug)}$$

*Reference objects* Security relevant MiniML terms, namely $\lambda$-terms and locations, 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 the MiniML program residing within the protected memory, to keep track of which locations or $\lambda$-terms and locations have been shared with the attacker. MiniML$^+$ models reference objects for $\lambda$-terms and locations through names $\mathbf{n}^{\mathbf{f}}_{\mathbf{i}}$ and $\mathbf{n}^{\mathbf{l}}_{\mathbf{i}}$ respectively. Both names are tracked in the MiniML state $\mathtt{S}$ through a map $\mathtt{N}$ that records the associated term and type, as follows.

$$\mathtt{N} ::= \star \mid \mathtt{N}, \mathbf{n}^{\mathbf{f}}_{\mathbf{i}} \mapsto (t, \tau) \mid \mathtt{N}, \mathbf{n}^{\mathbf{l}}_{\mathbf{i}} \mapsto (t, \tau)$$

A *fresh* name $\mathbf{n}^{\mathbf{f}}_{\mathbf{i}}$ is created deterministically every time a $\lambda$-term is shared between the MiniML program and the attacker, in contrast the index $i$ of the name $\mathbf{n}^{\mathbf{l}}_{\mathbf{i}}$ will correspond to the index of the location it refers to $(\mathbf{n}^{\mathbf{l}}_{\mathbf{i}} \mapsto l_i)$.

The MiniML$^a$ attacker shares only its functions with the MiniML programs. These attacker functions are embedded through a term ${}^{\tau}\mathtt{F}(\lambda\mathbf{x}.\mathbf{t})$. A MiniML program calls this embedded attacker function, as follows.

$$\mu \Vdash \overline{\mathbf{C}}, \mathbf{C} \;||\; \mathtt{N}; \mu \Vdash \Sigma \circ \mathtt{E}[({}^{\tau_1 \to \tau_2}\mathtt{F}(\lambda\mathbf{x}.\mathbf{t})\ \mathrm{v})] : \tau \twoheadrightarrow \qquad \text{(M-Call)}$$

$$\mu \Vdash \overline{\mathbf{C}}, \mathbf{C}[((\lambda\mathbf{x}.\mathbf{t})\ [\cdot])] \;||\; \mathtt{N}; \mu \Vdash \Sigma, \mathtt{E} : \tau_2 \to \tau \rhd v : \tau_1$$

### 3.4 $\mathcal{B}^a$: a Bisimulation over the MiniML$^a$ Attacker

To capture the inputs and observations of the high-level MiniML$^a$ attacker in a formalism that can be easily related to the inputs and observations of the assembly language attacker, we define a notion of bisimulation $\mathcal{B}^a$. To do so we define an applicative bisimulation in the style of Jeffrey and Rathke's applicative bisimulation for the $\nu$ref-calculus [4]. The applicative bisimulation is defined through a labelled transition system (LTS), that models the inputs and observations of the high-level MiniML$^a$ attacker in its interactions with the MiniML program. The LTS is a triple $(\mathtt{S}, \alpha, \xrightarrow{\alpha})$ where the MiniML states $\mathtt{S}$ of MiniML$^+$ are the states of the LTS, $\alpha$ the set of labels and $\xrightarrow{\alpha}$ the labelled transitions between states. The attacker state $\mathbf{A}$ is thus not represented in these labelled reductions, instead the labels $\alpha$ denote the observations of the high-level MiniML$^a$ attacker as follows.

$$\alpha ::= \gamma \mid \tau \mid \surd$$
$$\gamma ::= \mathbf{v?} \mid \mathbf{v!} \mid \mathbf{wr} \mid \gg (\lambda \mathbf{x.t}) \mid \gg \mathbf{n_i^l} \mid \gg \mathbf{n_i^f} \mid \gg \mathbf{ref}^{\tau} \mid \mathbf{!n_i^l}$$

The labelled reductions of the LTS are of the form: $\mathtt{S} \xrightarrow{\gamma} \mathtt{S}'$. The most relevant transitions are as follows.

$$\frac{\mathbf{A} \mid\mid \mathtt{N}; \mu \Vdash \Sigma \circ t : \tau \twoheadrightarrow \mathbf{A} \mid\mid \mathtt{N}; \mu' \Vdash \Sigma \circ t' : \tau}{\mathtt{N}; \mu \Vdash \Sigma \circ t : \tau \xrightarrow{\tau} \mathtt{N}; \mu' \Vdash \Sigma \circ t' : \tau} \quad \text{(S-Inner)}$$

$$\mathtt{N}; \mu \Vdash \Sigma, \mathtt{E} : \tau \to \tau' \xrightarrow{\mathbf{v?}} \mathtt{N} \Vdash \Sigma, \mathtt{E} : \tau \to \tau' \lhd \mathbf{v} : \tau \quad \text{(A-V)}$$

$$\mathtt{N}; \mu \Vdash \Sigma \xrightarrow{\mathbf{!n_i^l}} \mathtt{N}; \mu \Vdash \Sigma \circ !l_i : \tau \quad \textit{where } \mathtt{N}(\mathbf{n_i^l}) = (l_i, \mathtt{Ref}\ \tau) \quad \text{(D-N)}$$

$$\mathtt{N}; \mu \Vdash \Sigma \xrightarrow{\gg \mathbf{ref}^{\tau}} \mathtt{N}; \mu \Vdash \Sigma, (\mathtt{ref}\ [\cdot]) : \tau \to \mathtt{Ref}\ \tau \quad \text{(A-R)}$$

$$\mathtt{N}; \mu \Vdash \Sigma \xrightarrow{\gg \mathbf{n_i^f}} \mathtt{N}; \mu \Vdash \Sigma, (t\ [\cdot]) : \tau \to \tau' \quad \textit{where } \mathtt{N}(\mathbf{n_i^f}) = (t, \tau \to \tau') \quad \text{(C-N)}$$

$$\mathtt{N}; \mu \Vdash \Sigma \circ \mathtt{E}[(^{\tau_1 \to \tau_2}\mathtt{F}(\lambda \mathbf{x.t})\ \mathtt{v})] : \tau \xrightarrow{\gg (\lambda \mathbf{x.t})} \mathtt{N}; \mu \Vdash \Sigma, \mathtt{E} : \tau_2 \to \tau \rhd v : \tau_1 \quad \text{(C-L)}$$

$$\mathtt{N}; \mu \Vdash \Sigma \lhd \mathbf{wr} : \tau \xrightarrow{\mathbf{wr}} \star; \emptyset \Vdash \varepsilon \quad \text{(Wr-I)}$$

$$\mathtt{N}; \mu \Vdash \Sigma \rhd \mathbf{v} : \tau \xrightarrow{\mathbf{v!}} \mathtt{N}; \mu \Vdash \Sigma \quad \text{(M-V)} \qquad \mathtt{N}; \mu \Vdash \Sigma \xrightarrow{\mathbf{wr}} \star; \emptyset \Vdash \varepsilon \quad \text{(Wr-C)}$$

The internal reduction steps (S-Inner) and the marshalling transitions are *not observable* to the attacker and are thus labelled as silent through the label $\tau$. The values $\mathbf{v}$ that the attacker returns or inputs are decorated with ? (A-V). The values returned by the MiniML program to the attacker, returned as marshalled values $\mathbf{v}$, are decorated with ! (M-V). The attacker can dereference a shared location in a one step transition that is labeled as $\mathbf{!n_i}$ (D-N). The attacker can also set locations, create shared locations (A-R) and apply shared MiniML $\lambda$-terms through two transitions. In the first step, whose label is decorated with $\gg$, the MiniML program is updated with the requested operation and the targeted term. In the second step the attacker injects an argument as captured by the value sharing rule (A-V). Whenever, an MiniML program calls a function of the attacker (C-L) the attacker observes the call as well as the, immediately following,

argument to the function (M-V). If the marshalling fails (Wr-I) or the attacker makes an inappropriate call (Wr-C), the transition is labelled as wrong (**wr**).

We define a weak bisimulation over this LTS. In contrast to a strong bisimulation, such a bisimulation does not use the silent transitions between two states, thus capturing the fact that the attacker cannot directly observe the number of internal reduction steps within a MiniML program. Define the transition relation $\mathtt{S} \overset{\gamma}{\Rightarrow} \mathtt{S}'$ as $\mathtt{S} \xrightarrow{\tau}{}^{*} \xrightarrow{\gamma} \mathtt{S}'$ where $\xrightarrow{\tau}{}^{*}$ is the reflexive transitive closure of the silent transitions $\xrightarrow{\tau}$. Our bismulation $\mathcal{B}^a$ over the observations and inputs of the MiniML$^a$ attacker is now defined as follows.

**Definition 2.** *The relation $\mathcal{B}^a$ is a **bisimulation** iff $\mathtt{S}_1\ \mathcal{B}^a\ \mathtt{S}_2$ implies:*

*(1) Given $\mathtt{S}_1 \overset{\gamma}{\Rightarrow} \mathtt{S}_1'$ there is $\mathtt{S}_2'$ such that: $\mathtt{S}_2 \overset{\gamma}{\Rightarrow} \mathtt{S}_2'$ and $\mathtt{S}_1'\ \mathcal{B}^a\ \mathtt{S}_2'$*

*(2) Given $\mathtt{S}_2 \overset{\gamma}{\Rightarrow} \mathtt{S}_2'$ there is $\mathtt{S}_1'$ such that: $\mathtt{S}_1 \overset{\gamma}{\Rightarrow} \mathtt{S}_1'$ and $\mathtt{S}_1'\ \mathcal{B}^a\ \mathtt{S}_2'$*

We denote bisimilarity, the largest bisimulation, as $\approx^a$.

*Congruence* Just defining a bisimulation over the observations and inputs of the MiniML$^a$ attacker is not enough. We must also prove that the bisimulation accurately captures those observations and inputs. We do this by proving that the bisimulation $\mathcal{B}^a$ is a *congruence*: it coincides with contextual equivalence in MiniML$^+$ where the contexts of MiniML$^+$ are all possible attacks definable in MiniML$^a$. Formally contextual equivalence over MiniML$^+$ is defined as follows.

**Definition 3. Contextual equivalence for MiniML$^+$** ($\simeq^a$) *is defined as:*

$$\mathtt{S}_1 \simeq^a \mathtt{S}_2 \overset{def}{=} \forall \mathbf{A}.(\mathbf{A} \mid\mid \mathtt{S}_1)\Uparrow \iff (\mathbf{A} \mid\mid \mathtt{S}_2)\Uparrow$$

**Theorem 1 (Congruence of the Bisimilarity).** $\mathtt{S}_1 \simeq^a \mathtt{S}_2 \iff \mathtt{S}_1 \approx^a \mathtt{S}_2$.

A proof of this property is an adaptation of existing results [5], as such we leave it to the companion technical report [6].

## 4 A Bisimulation over the Assembly Language Attacker

In this section we introduce a bisimulation over the assembly language that captures its interactions with an 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 a fully abstract trace semantics over the interactions between the attacker and the protected memory space (Section 4.1). Next, we define the applicative bisimulation $\approx^l$ over an LTS whose state is a low level extension of the MiniML state of Section 3.3 (Section 4.2). Later on in Section 5, we relate this bisimulation to the bisimulation over the high-level attacker to prove the accuracy of the high-level attacker.

### 4.1 A Trace Semantics for the Assembly Language Attacker

To accurately reason about the capabilities and behaviour of the assembly attacker we make use of the *labels* used by the fully abstract trace semantics of Patrignani and Clarke [11] for assembly programs enhanced with PMA. These trace semantics transitions over a state $\Lambda = (p, r, f, m, s)$ where $p$ is the program

counter, $m$ is 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. Additionally $\Lambda$ can be $(\textbf{unkown}, m, s)$ when modeling the attacker. The attacker thus does not feature an explicit state, instead the labels $L$ capture its observations and inputs as follows.

$$L ::= \ \alpha \mid \tau \qquad \alpha ::= \ \surd \mid \gamma! \mid \gamma? \qquad \gamma ::= \ \mathit{call}\ p(r) \mid \mathit{ret}\ p(r)$$

A label $L$ can be either an observable action $\alpha$ or a non-observable action $\tau$. Decorations ? and ! indicate the direction of the observable action: from the attacker to the protected memory (?) or vice-versa (!). Observable actions include a tick $\surd$ indicating termination, and actions $\gamma$: function calls or returns to a certain address $p$, combined with the registers $r$. These registers convey the arguments of the calls and returns.

The traces provide an accurate model of the attacker as they coincide with contextual equivalence for assembly programs enhanced with PMA.

**Proposition 1 (Full Abstraction [11]).** $P_1 \simeq_l P_2 \iff \mathsf{Tr}(P_1) = \mathsf{Tr}(P_2)$

Where $\simeq_l$ denotes contextual equivalence between low-level programs and where $\mathsf{Tr}(P)$ computes the traces of a program, with an initial state $\Lambda(P)$ as follows.

$$\mathsf{Tr}(P) = \{\overline{\gamma} \mid \exists \Lambda'.\Lambda(P) \stackrel{\overline{\gamma}}{\Longrightarrow} \Lambda'\}$$

Note that this trace semantics does not include explicit reads or writes from the protected memory to the unprotected memory or reads and writes from the attacker to the protected memory. The latter is not possible as it violates PMA (Section 2.1), the former is not required in our work as the data shared by MiniML programs fits in to the registers $r$. Incorporating larger data structures that require low-level reads and writes, has been left for future work.

### 4.2 $\mathcal{B}^l$: a Bisimulation over the Assembly Language Attacker

While the trace semantics of Section 4.1 provides an accurate method for reasoning about the attacker, the states $\Lambda$ of that semantics include many low-level details of the protected memory that are not relevant to the result of this paper. We thus define a bisimulation $\mathcal{B}^l$ that keeps the labels of the trace semantics, to denote the inputs and observations of the assembly language attacker, but features a more high-level state that denotes only the relevant information.

This state is a triple $\langle \mathtt{S}, e, \overline{p} \rangle$: the MiniML state of MiniML$^+$ extended with static set of entry points $e$ and a stack of return pointers $\overline{p}$. The MiniML state $\mathtt{S}$ captures the current state of the MiniML program interacting with the attacker from within protected memory. The set of entry points $e$ contains the addresses $p^e$ of the entry points into the protected memory that the attacker can call. The stack of return pointers $\overline{p}$ enables the MiniML program to return to the address of the attacker were a call to an entry point originated from.

Note that assembly language attacker inputs and outputs words of bytes $w$ instead of the high-level values $\mathbf{v}$. The marshalling rules of MiniML$^+$ over the MiniML state $\mathtt{S}$ are thus adapted to convert to and from words $w$. Marshalling

*in* a `true` value and marshalling *out* a `false` value, for example, is as follows.

$$\mathtt{N};\mu \Vdash \Sigma \rhd \mathtt{false} : \mathtt{Bool} \twoheadrightarrow \mathtt{N};\mu \Vdash \Sigma \rhd \mathtt{0x0} : \mathtt{Bool} \quad \text{(Out-False)}$$

$$\mathtt{N};\mu \Vdash \Sigma \lhd \mathtt{0x01} : \mathtt{Bool} \twoheadrightarrow \mathtt{N};\mu \Vdash \Sigma \lhd \mathtt{true} : \mathtt{Bool} \quad \text{(In-True)}$$

The numbers $\overline{n}$, and names $\mathbf{n}^{\mathbf{l}}_{\mathbf{j}}$ and $\mathbf{n}^{\mathbf{f}}_{\mathbf{i}}$ are converted into a word of bytes $w$, in a similar manner. Functions $p_f$ from the attacker are embedded as ${}^{\tau_1\to\tau_2}\mathtt{F}p_f$.

The bisimulation $\mathcal{B}^l$ is now over defined an LTS $(\langle \mathtt{S}, e, \overline{p}\rangle, L, \xrightarrow{L})$, where $L$ are the labels of the fully abstract trace semantics and $\xrightarrow{L}$ denotes the labelled transitions between the states. The most relevant transitions are as follows.

$$\frac{\mathtt{N};\mu \Vdash \Sigma \circ t : \tau \twoheadrightarrow \mathtt{N};\mu' \Vdash \Sigma \circ t' : \tau}{\langle (\mathtt{N};\mu \Vdash \Sigma \circ t : \tau), e, p_r : \overline{p}\rangle \xrightarrow{\tau} \langle (\mathtt{N};\mu' \Vdash \Sigma \circ t' : \tau), e, p_r : \overline{p}\rangle} \quad \text{(S-Inner)}$$

$$\langle (\mathtt{N};\mu \Vdash \Sigma \circ t : \tau), e, \emptyset\rangle \xrightarrow{call\ p^e_{\mathsf{start}}(p_r)?} \langle (\mathtt{N};\mu \Vdash \Sigma \circ t : \tau), e, p_r : \emptyset\rangle \quad \text{(A-Start)}$$

$$\langle (\mathtt{N};\mu \Vdash \Sigma \rhd w : \tau), e, p_r : \overline{p}\rangle \xrightarrow{ret\ p_r\ (w)!} \langle (\mathtt{N};\mu \Vdash \Sigma), e, \overline{p}\rangle \quad \text{(M-Ret)}$$

$$\langle (\mathtt{N};\mu \Vdash \Sigma, \mathtt{E} : \tau \to \tau'), e, \overline{p}\rangle \xrightarrow{ret\ p^e_{\mathsf{retb}}(w)?} \langle (\mathtt{N};\mu \Vdash \Sigma, \mathtt{E} : \tau_f \lhd w : \tau), e, \overline{p}\rangle \quad \text{(A-R)}$$

$$\langle (\mathtt{N};\mu \Vdash \Sigma), e, \overline{p}\rangle \xrightarrow{call\ p^e_{\mathsf{deref}}(w_n, p_r)?} \langle (\mathtt{N};\mu \Vdash \Sigma \circ !l_i : \tau), e, p_r : \overline{p}\rangle \quad \text{(A-Deref)}$$
$$\textit{where } \mathtt{N}(w_n) = (l_i, \mathtt{Ref}\ \tau)$$

$$\langle (\mathtt{N};\mu \Vdash \Sigma), e, \overline{p}\rangle \xrightarrow{call\ p^e_{\mathsf{appl}}(w_n, w, p_r)?} \langle (\mathtt{N};\mu \Vdash \Sigma, (t\ [\cdot]) : \tau \to \tau' \lhd w : \tau), e, \overline{p}'\rangle$$
$$\textit{where } \mathtt{N}(w_n) = (t, \tau \to \tau') \textit{ and } \overline{p}' = p_r : \overline{p} \quad \text{(A-Apply)}$$

$$\langle (\mathtt{N};\mu \Vdash \Sigma \lhd \mathbf{wr} : \tau), e, p_r : \overline{p}\rangle \xrightarrow{\checkmark} \langle (\star; \emptyset \Vdash \boldsymbol{\varepsilon}), e, \emptyset\rangle \quad \text{(Wr-I)}$$

$$\langle (\mathtt{N};\mu \Vdash \Sigma, \mathtt{E} : \tau \to \tau'), e, \overline{p}\rangle \xrightarrow{ret\ p(w)?} \langle (\star; \emptyset \Vdash \boldsymbol{\varepsilon}), e, \emptyset\rangle \textit{ where } p \neq p^e_{\mathsf{retb}} \quad \text{(Wr-R)}$$

$$\langle (\mathtt{N};\mu \Vdash \Sigma), e, p_r : \overline{p}\rangle \xrightarrow{call\ p(\overline{w})?} \langle (\star; \emptyset \Vdash \boldsymbol{\varepsilon}), e, \emptyset\rangle \quad \textit{where } p \notin e \quad \text{(Wr-C)}$$

$$\text{(M-Call)}\quad \frac{\begin{array}{c}\mathtt{N};\mu \Vdash \Sigma \circ \mathtt{E}[({}^{\tau_1\to\tau_2}\mathtt{F}p_f\ \mathtt{v})] : \tau \twoheadrightarrow \mathtt{N};\mu \Vdash \Sigma, \mathtt{E} : \tau_2 \to \tau \rhd v : \tau_1 \\ \mathtt{N};\mu \Vdash \Sigma, \mathtt{E} : \tau_2 \to \tau \rhd v : \tau_1 \twoheadrightarrow^* \mathtt{N};\mu \Vdash \Sigma, \mathtt{E} : \tau_2 \to \tau \rhd w : \tau_1\end{array}}{\langle (\mathtt{N};\mu \Vdash \Sigma \circ \mathtt{E}[({}^{\tau_1\to\tau_2}\mathtt{F}p_f\ \mathtt{v})] : \tau), e, \overline{p}\rangle \xrightarrow{call\ p_f(w)!} \langle (\mathtt{N};\mu \Vdash \Sigma, \mathtt{E} : \tau_2 \to \tau), e, \overline{p}\rangle}$$

Transitions within the MiniML program, such as for example S-Inner, are not observable the attacker and are thus again labelled as silent. To start the computation of the MiniML program, the low-level attacker calls the entry point $p^e_{\mathsf{start}}$ passing as its only argument $p_r$ the address at which it expects the result returned (A-Start). When the MiniML program returns to that address (M-Ret), it makes use of modified marshalling rules to return a word $w$ to the address at the head of the stack $\overline{p}$ instead of MiniML$^a$ values, as detailed earlier. The assembly language attacker, in contrast, has less freedom for its returns. Because it cannot jump to an address of the protected memory outside of the entry points, it must return its values through a return entry point $p^e_{\mathsf{retb}}$ (A-R). Whereas each

operation by the high-level attacker on the MiniML terms shared to it through names $\mathbf{n_i}$ was denoted with its own label, the assembly language attacker calls a separate entry point for each operation (A-Deref,A-Apply) passing a byte word representation of the names ($w_n$) as an argument to the call. Whenever the assembly-language attacker makes a mistake by either providing words that cannot be marshalled (Wr-I) or by calling or returning to an inaccessible address (Wr-C,Wr-R) the protected memory is terminated to the empty state $\langle(\star; \emptyset \Vdash \boldsymbol{\varepsilon}), e, \emptyset\rangle$. While the attacker makes many different types of calls to the protected memory, the MiniML program, only calls attacker functions $p_f$ whenever it applies them to an MiniML value (M-Call).

We now define a notion of *weak* bisimulation, that does not take into account the silent transitions $\tau$ only the actions $\alpha$, over the LTS. Define the transition relation $\langle \mathtt{S}, e, \overline{p}\rangle \stackrel{\alpha}{\Longrightarrow} \langle \mathtt{S}', e, \overline{p}'\rangle$ as $\langle \mathtt{S}, e, \overline{p}\rangle \stackrel{\tau}{\longrightarrow}^* \stackrel{\alpha}{\longrightarrow} \langle \mathtt{S}', e, \overline{p}'\rangle$ where $\stackrel{\tau}{\longrightarrow}^*$ is the reflexive transitive closure of the silent transitions $\stackrel{\tau}{\longrightarrow}$.

**Definition 4.** *$\mathcal{B}^l$ is a bisimulation iff $\langle \mathtt{S}_1, e_1, \overline{p_1}\rangle\, \mathcal{B}\, \langle \mathtt{S}_2, e_2 \overline{p_2}\rangle$ implies:*

1. *Given $\langle \mathtt{S}_1, e_1, \overline{p_1}\rangle \stackrel{\alpha}{\Longrightarrow} \langle \mathtt{S}_1', e_1, \overline{p_1}'\rangle$, There is $\langle \mathtt{S}_2', e_2, \overline{p_2}'\rangle$ such that $\langle \mathtt{S}_2, e_2, \overline{p_2}\rangle \stackrel{\alpha}{\Longrightarrow} \langle \mathtt{S}_2', e_2, \overline{p_2}'\rangle$ and $\langle \mathtt{S}_1', e_1, \overline{p_1}'\rangle \mathcal{B} \langle \mathtt{S}_2', e_2, \overline{p_2}'\rangle$*
2. *Given $\langle \mathtt{S}_2, e_2, \overline{p_2}\rangle \stackrel{\alpha}{\Longrightarrow} \langle \mathtt{S}_2', e_2, \overline{p_2}'\rangle$, There is $\langle \mathtt{S}_1', e_1, \overline{p_1}'\rangle$ such that $\langle \mathtt{S}_1, e_1, \overline{p_1}\rangle \stackrel{\alpha}{\Longrightarrow} \langle \mathtt{S}_1', e_1, \overline{p_1}'\rangle$ and $\langle \mathtt{S}_1', e_1, \overline{p_1}'\rangle \mathcal{B} \langle \mathtt{S}_2', e_2, \overline{p_2}'\rangle$*

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

## 5 Full Abstraction

We now establish the accuracy of the high-level attacker by proving that the bisimulation over the assembly-language attacker is a full abstraction of the bisimulation over the high-level MiniML$^a$ attacker. We thus prove that there is no assembly language attacker action that affects the abstractions of MiniML programs residing in the protected memory, that cannot be replicated by the high-level attacker MiniML$^a$.

**Theorem 2 (Full Abstraction).** $\{t_1\}^\uparrow \approx^a \{t_2\}^\uparrow \iff \{t_1\}^\downarrow \approx^l \{t_2\}^\downarrow$

where $\{t\}^\uparrow$ denotes the start state: $(\star; \emptyset \Vdash \boldsymbol{\varepsilon} \circ t : \tau)$ of an MiniML term $t$ when faced with the MiniML$^a$ attacker and where $\{t\}^\downarrow$ denotes the start state: $\langle(\star; \emptyset \Vdash \boldsymbol{\varepsilon} \circ t : \tau), e, \emptyset\rangle$ of an MiniML term when faced with the assembly language attacker.

The proof splits the thesis into two sublemma: preservation and reflection.

**Lemma 1. (Preservation)** $\{t_1\}^\uparrow \approx^a \{t_2\}^\uparrow \Rightarrow \{t_1\}^\downarrow \approx^l \{t_2\}^\downarrow$.

*Proof Sketch* We must establish that there exists a relation $\mathcal{R}$, so that:
(1) $\{t_1\}^\downarrow \mathcal{R} \{t_2\}^\downarrow$ and (2) that $\mathcal{R}$ relates low-level states $\langle \mathtt{S}, e, \overline{p}\rangle$ and $\langle \mathtt{S}', e', \overline{p}'\rangle$ as would $\mathcal{B}^l$. We define $\mathcal{R}$ as a union of 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: $\langle(\mathtt{N}; \mu \Vdash \Sigma), e, \overline{p}\rangle$ and $\langle(\mathtt{N}'; \mu' \Vdash \Sigma'), e', \overline{p}'\rangle$ and enforces that the name maps are equivalent: $Dom(\mathtt{N}) = Dom(\mathtt{N}') \wedge \forall \mathbf{n_i}.\mathtt{N}(\mathbf{n_i}) \simeq \mathtt{N}'(\mathbf{n_i})$, the evaluation stacks

are equivalent: $|\Sigma| = |\Sigma'| \wedge \forall \mathtt{E}, \mathtt{E}', t.E[t] \simeq E'[t]$, the entry point sets are equal $e = e'$, and that the return address stacks are equal as well $\overline{p} = \overline{p}'$. The relation $\mathcal{R}_1$ relates two states reducing terms contextually equivalent terms $t$ and $t'$ 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 label $L$. The most challenging sub-case is the call from MiniML to the low-level attacker labelled as $L = \mathit{call}\ p_f(w)!$. To prove that both states will perform the same outward calls, we rely on the insight that by including references in MiniML we have that two equivalent MiniML terms will perform the same function calls, as illustrated for (Ex-1) in Section 2.2.

**Lemma 2. (Reflection)** $\{t_1\}^{\downarrow} \approx^{l} \{t_2\}^{\downarrow} \Rightarrow \{t_1\}^{\uparrow} \approx^{a} \{t_2\}^{\uparrow}$.

*Proof Sketch* We prove the contrapositive: $\{t_1\}^{\uparrow} \not\approx^{a} \{t_2\}^{\uparrow} \Rightarrow \{t_1\}^{\downarrow} \not\approx^{l} \{t_2\}^{\downarrow}$. The proof has two cases. In the first case the bisimulation fails immediately as the MiniML terms $t_1$ and $t_2$ embedded in $\mathtt{S}$ either reduce to difference values or diverge. These differing LTS transitions are replicated directly in the low-level bisimulation, the only difference being the inclusion of a start transition with label: $\mathit{call}\ p^{e}_{\mathtt{start}}(p_r)$ that starts the reduction of the embedded MiniML terms. In the second case there is a sequence of context actions ($\gg (\lambda \mathbf{x}.\mathbf{t})$ | $\gg \mathbf{n}^{\mathbf{l}}_{\mathbf{i}}$ | $\gg \mathbf{n}^{\mathbf{f}}_{\mathbf{i}}$ | $\gg \mathbf{ref}^{\tau}$ | $!\mathbf{n}^{\mathbf{l}}_{\mathbf{i}}$) 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 for both lemmas are provided in a companion report [6].

## 6 Related Work

Our attacker model is based on the insights of Wand [15] on the nature of programming language reflection. Alternative attacker models are Jagadeesan *et al.*'s attacker language with low-level memory access operators [3] or the erasure function approach of several non-interference works [8]. The former is only suitable for low-memory models with address space randomization, the latter does not lend itself to low-level attackers.

In Section 3.3 we use the interoperation semantics of Larmuseau *et al.* [5] to model the interoperation between the MiniML$^{a}$ attacker and the source language MiniML. There exist multiple alternatives for language interoperation: Matthews and Findler's multi-language semantics [9] enables two languages to interoperate through direct syntactic embedding and Zdancewic *et al.*'s multi-agent calculus that treats the different modules or calculi that make up a program as different principals, each with a different view of the environment [16]. These alternatives, however, do not provide separated program states or explicated marshalling rules both required to model the assembly language attacker.

Our notions of bisimulation over the interactions of the high-level and low-level attackers are based on the bisimulations for the $\nu$ref-calculus by Jeffrey and Rathke [4]. An alternative approach could be the environmental bisimulations of

Sumii and Pierce [14], which would not require a `hash` operation in MiniML to make the locations observable within the labels. Their bisimulations, however, do not provide a clear formalism to reason about the observations of an attacker.

## 7 Conclusions

This paper presented a high-level attacker model $\mathcal{L}^a$ that captures the threat that an assembly-language attacker poses to the abstractions of a program that resides within the memory space protected by PMA, a low-level memory isolation mechanism. The accuracy of this high-level attacker model was proven for an example language MiniML, by relating a bisimulation over the the high-level attacker model to a bisimulation over the assembly language attacker.

## References


1. M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.
2. M. Furr and J. S. Foster. Checking type safety of foreign function calls. In *PLDI '05*, pages 62–72. ACM, 2005.
3. R. Jagadeesan, C. Pitcher, J. Rathke, and J. Riely. Local memory via layout randomization. In *CSF '11*, pages 161–174. IEEE, 2011.
4. A. Jeffrey and J. Rathke. Towards a theory of bisimilarity for local names. In *Logic in Computer Science*, pages 56–66. IEEE, 2000.
5. A. Larmuseau and D. Clarke. Formalizing a secure foreign function interface. In *SEFM 2015*, LNCS. Springer. To appear, online at: https://db.tt/y87tcQ0V.
6. A. Larmuseau and D. Clarke. Modelling an Assembly Attacker by Reflection. Technical Report 2015-026, Uppsala University, 2015.
7. X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vôuillon. The Objective Caml system, release 4.02. Technical report, INRIA, August 2014.
8. P. Li and S. Zdancewic. Arrows for secure information flow. *Theoretical Computer Science*, 411(19):1974 – 1994, 2010.
9. J. Matthews and R. B. Findler. Operational semantics for multi-language programs. *TOPLAS*, 31(3):12:1–12:44, 2009.
10. F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP '13*, pages 10:1–10:1. ACM, 2013.
11. M. Patrignani and D. Clarke. Fully Abstract Trace Semantics of Low-level Isolation Mechanisms. In *SAC '14*, pages 1562–1569. ACM, 2014.
12. G. Plotkin. LCF considered as a programming language. *Theor. Comput. Science*, 5:223–255, 1977.
13. R. Strackx and F. Piessens. Fides: Selectively hardening software application components against kernel-level malware. In *CCS '12*, pages 2–13. ACM.
14. E. Sumii and B. C. Pierce. A bisimulation for dynamic sealing. In *POPL '04*, pages 161–172. ACM, 2004.
15. M. Wand. The theory of fexprs is trivial. *Lisp and Symbolic Computation*, 10(3):189–199, 1998.
16. S. Zdancewic, D. Grossman, and G. Morrisett. Principals in programming languages: a syntactic proof technique. In *ICFP '99*, pages 197–207. ACM.