Modelling an Assembly Attacker by Reflection

Adriaan Larmuseau\textsuperscript{1} and Dave Clarke\textsuperscript{1,2}
\textsuperscript{1} Uppsala University, Sweden, \texttt{first.last@it.uu.se}
\textsuperscript{2} iMinds-Distrinet, K.U. Leuven, Belgium \texttt{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 technical report we formally prove that a light-weight ML-like including references, equipped with a reflection operator can serve as an accurate model for malicious assembly language programs, when reasoning about the security threats posed to the abstractions of high-level functional programs that reside within a protected memory space. The proof proceeds by relating a bisimulation over the inputs and observations of the assembly language attacker to a bisimulation over the inputs and observations of 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 [2,9] or extended with Foreign Function Interfaces (FFIs) that enable interoperation with these low-level target languages [4]. The security features of these low-level languages, however, rarely coincide with those of functional languages. In practice the high-level programs are thus 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 such malicious low-level code has on high-level programs is, however, rather challenging as the syntax and semantics of low-level code differs greatly from that of high-level functional programming languages. To that end different high-level models that capture the capabilities of certain types of low-level attackers have been introduced. Jagadeesan \textit{et al}. [6], for example, make use of a $\lambda$-calculus extended with low-level memory access operators to model the power of a low-level attacker within memory with randomized address space layouts. The validity of such high-level models for low-level attackers is, however, often called into question.

In this paper we present $\mathcal{L}^\circ$ 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 [8] has been that this attacker model accurately captures the threats posed by an assembly-level attacker to the abstractions of a source language $L$, 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. Because PMA is set to be supported in a future generation of commercial processors [13], this accurate high-level model of the threats that an assembly language attacker, residing outside of the protected memory, poses to the security features of programs residing within the protected memory, is bound to be useful for many different practical applications.

In what follows, we formally prove that $L^a$, 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 $L^a$ attacker model MiniML$^a$. The proof technique proceeds as follows: first we develop a notion of bisimulation over the interactions between MiniML$^a$ and MiniML. Next we develop a notion of bisimulation over the interactions between the low-level attacker and programs in MiniML, by adopting the labels of a previously developed fully abstract trace semantics for the attacker model [15]. 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 as follows. Firstly the paper introduces the assembly language attacker and its high-level replacement (Section 2). Secondly the paper 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 $L_a$, 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 $L$ (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 [15].

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 [17]. 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, neither the code nor the data of the protected memory can be executed, written or read by the unprotected memory. The code section can only be executed from the outside through the entry points and the data section can only accessed by the code section. An overview of the access control mechanism is given below.

<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 r x</td>
<td>r w r w x</td>
</tr>
<tr>
<td>Unprotected</td>
<td>x</td>
<td>r w x</td>
</tr>
</tbody>
</table>

A variety of PMA implementations exist. While current implementations of PMA are still research prototypes [17], Intel is developing a new instruction set, referred to as SGX, that will enable the usage of PMA in future commercially available processors [13].

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 [17].

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 [16]. 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 is 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:

$$P_1 \simeq P_2 \overset{\text{def}}{=} \forall C : C[P_1] \uparrow \iff C[P_2] \uparrow$$

where $\uparrow$ denotes divergence [16].

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) $(\lambda x.(x \ 2) + (x \ 2))$  (b) $(\lambda x.(x \ 2) \ast 2)$  (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.

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

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 $L$ as it can compare and manipulate any sequence of bits it has access to. When interop-erating 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 $L^a$

Our high-level attacker model $L^a$ aims to accurately model the threats posed by the assembly-language attacker to the contextual equivalences of a source language $L$, 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 $L$ to derive the high-level, but accurate, attacker model $L^a$. 

4
Transformation 1: removal of type safety Type safety forces programs to preserve types and avoids stuck states. Removing the typing rules of \( L \) ensures that \( L^a \) 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 [19], 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 \( L \). The high-level attacker model \( L^a \), however, is derived from \( L \) and is thus susceptible to the same execution mechanisms as \( L \). For \( L^a \) 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 \( L^a \) attacker model to different source languages \( L \), we have encountered but one constraint. It is only possible to derive an accurate attacker model \( L^a \) from a source languages \( L \) 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 \( L^a \) 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 \( L^a \) 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 \( L^a \) attacker model denoted as \( \text{MiniML}^a \) (Section 3.2). Next, we model the interactions between MiniML and the high-level attacker \( \text{MiniML}^a \) by applying our previously developed interoperation semantics [8], resulting in a combined calculus \( \text{MiniML}^+ \) (section 3.3). Lastly a bisimulation \( B^a \) that captures the observations and inputs of the high-level \( \text{MiniML}^a \) attacker is derived over the semantics of this \( \text{MiniML}^+ \) (Section 3.4). Later on, in Section 5, this bisimulation is related to a bisimulation \( B^d \) over the observations and inputs of the assembly-language attacker (Section 4), to prove the accuracy of the high-level \( \text{MiniML}^a \) attacker.

In what follows, the source language MiniML is typeset in a black font. The attacker model \( \text{MiniML}^a \) 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.
The reduction rules of MiniML are as expected.

\[
\begin{align*}
\mu | E[[\text{if } \text{true } t_2 t_3]] & \rightarrow \mu | E[t_2] \\
\mu | E[[\lambda x : \tau . t] v] & \rightarrow \mu | E[t[v/x]] \\
\mu | E[\text{let } x = v \text{ in } t] & \rightarrow \mu | E[e[v/x]] \\
\mu | E[\text{letrec } x : \tau_1 \text{ in } t_2] & \rightarrow \mu | E[t'] \\
\mu | E[\text{op } n_1 n_2] & \rightarrow \mu | E[\text{max}(\text{rnd}(n_1 \text{ op } n_2), 0)] \\
\mu | E[\text{ref } v] & \rightarrow \mu' | E[l_i] \\
\mu | E[\text{fix}(\lambda x : \tau . t)] & \rightarrow \mu | E[t[\text{fix}(\lambda x : \tau . t)/x]]
\end{align*}
\]
The typing rules of MiniML are entirely standard as well.

\[
\begin{align*}
\Gamma &\vdash x : \tau & \Gamma &\vdash b : \text{Bool} & \Gamma &\vdash \pi : \text{Int} \\
\Gamma &\vdash t_1 : \text{Ref } \tau & \Gamma &\vdash t : \text{ref } \tau & \Gamma &\vdash \text{unit} : \text{Unit} \\
\Gamma &\vdash t_1 : \text{Int} & \Gamma &\vdash t_2 : \text{Int} & \Gamma &\vdash e_1 : \tau \rightarrow \tau \\
\Gamma &\vdash \text{hash} t : \text{Int} & \Gamma &\vdash \text{fix } e_1 : \tau \\
\Gamma &\vdash (\text{op } t_1 t_2) : \text{Int} & \Gamma &\vdash (\text{cp } t_1 t_2) : \text{Bool} \\
\Gamma &\vdash t_1 : \text{Ref } \tau & \Gamma &\vdash t_2 : \tau \\
\Gamma &\vdash (\lambda x : \tau_1 t_1 : \tau_2) : \tau_2 & \Gamma &\vdash \text{let } x = t_1 \text{ in } t_2 : \tau \\
\Gamma &\vdash \text{letrec } x : \tau_1 \rightarrow \tau_2 & \Gamma &\vdash t_1 t_2 : \tau \\
\Gamma &\vdash \text{if } t_1 t_2 t_3 : \tau & \Gamma &\vdash \text{if } t_1 t_2 t_3 : \tau
\end{align*}
\]

Where the typing environment \( \Gamma \) is defined formally as follows.

\[
\Gamma ::= \emptyset \mid \Gamma, l_i : \tau \mid \Gamma, x : \tau
\]

**Contextual Equivalence** A MiniML context \( C \) is a MiniML term with a single hole \([\cdot]\), two MiniML terms are contextually equivalent if and only if there is no context \( C \) that can distinguish them. Contextual equivalence is formalised as follows.

**Definition 2.** Contextual equivalence \( (\simeq) \) is defined as:

\[
t_1 \simeq t_2 \overset{\text{def}}{=} \forall C. \ C[t_1] \uparrow \iff C[t_2] \uparrow
\]

where \( \uparrow \) denotes divergence \([16]\), \( t_1 \) and \( t_2 \) are closed terms and neither the terms and the contexts feature explicit locations \( l_i \) as they are not part of the static semantics. Note that two contextually equivalent MiniML terms \( t_1 \) and \( t_2 \) have the same type \( \tau \) as a context \( C \) observes the same typing rules as the terms.

### 3.2 The High-Level Attacker Model MiniML\(^a\)

We now apply the three transformations specified for \( L^a \) to MiniML, resulting in an attacker model denoted as MiniML\(^a\).
**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 a new term \( \text{wr} \) is introduced that captures non reducible expressions such as the following one:

\[
\mu \mid E[\text{if } v \ t_2 t_3] \rightarrow \mu \mid E[\text{wr}]
\]

where \( v \neq \text{true} \) or \( v \neq \text{false} \)

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 [19]. Reflection is added to MiniML\(^a\) by means of a syntactical equality testing operator modulo \( \alpha \)-equivalence \( =_\alpha \). It enables a program in MiniML\(^a\) to compare the syntax of any two terms as follows.

\[
\begin{align*}
\text{if } t_1 \ t_2 \text{ and } t_2 \text{ are } \alpha\text{-equiv} & \rightarrow \mu \mid E[\text{true}] \\
\text{if } t_1 \ t_2 \text{ and } t_2 \text{ are not } \alpha\text{-equiv} & \rightarrow \mu \mid E[\text{false}] \\
\end{align*}
\]

**Transformation 3: limit control flow** MiniML enforces that the evaluation order be from left to right using the evaluation contexts \( E \) as listed in Section 3.1. The \( \alpha \)-equivalence testing operator \( \equiv_\alpha \) works around this enforced control flow, by not reducing either of its sub-terms to values.

**Semantics of MiniML\(^a\)** The full syntax of MiniML\(^a\) is as follows.

\[
t ::= v \mid x \mid (t_1 \ t_2) \mid \text{op } t_1 \ t_2 \mid t. \mid \text{if } t_1 \ t_2 \ t_3 \mid \text{wr} \mid t_1 := t_2 \\
\mid \text{let } x = t_1 \text{ in } t_2 \mid \! t \mid \text{ref } t \mid \text{letrec } x = t_1 \text{ in } t_2 \mid t_1 ; t_2 \mid \text{fix } t \\
\mid \text{hash } t_1 \\
\text{op} ::= + \mid - \mid * \mid < \mid > \mid == \\
v ::= \text{unit} \mid l_i \mid \pi \mid (\lambda x. t) \mid \text{true} \mid \text{false} \\
E ::= [.] \mid E t \mid v E \mid \text{op } E t \\
\mid \text{op } v E \mid \text{if } E t_1 t_2 \mid \text{let } x = E \text{ in } t \mid \! E \mid \text{ref } E \mid E := t \mid E; t \\
\mid v := E \mid \text{fix } E \mid \text{hash } E
\]

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

\[
\begin{align*}
\mu \mid E[\text{if true } t_2 t_3] & \rightarrow \mu \mid E[t_2] \\
\mu \mid E[\text{if false } t_2 t_3] & \rightarrow \mu \mid E[t_3] \\
\mu \mid E[\text{wr}] & \rightarrow \mu \mid E[\text{wr}] \\
\mu \mid E[(\lambda x : \tau. t) v] & \rightarrow \mu \mid E[t[v/x]]
\end{align*}
\]
\[
\begin{align*}
&\mu \mid E[\text{let } x = v \text{ in } t] \rightarrow \mu \mid E[e/x] \\
&t' = \text{let } x = \text{fix}(\lambda x : \tau. t_1) \text{ in } t_2 \\
&\mu \mid E[\text{letrec } x : t \mid x \text{ in } t_2] \rightarrow \mu \mid E[t'] \\
&v_1 \neq v_2 \\
&\mu \mid E[cp v_1 v_2] \rightarrow \mu \mid E[wr] \\
&\mu' = \mu[1 \mapsto v] \\
&\mu \mid E[ref v] \rightarrow \mu' \mid E[l] \\
&t_1 \& t_2 \text{ are } \alpha-\text{equiv} \\
&\mu \mid E[t_1 \equiv \alpha t_2] \rightarrow \mu \mid E[true] \\
&l \notin \mu \\
&\mu \mid E[l] \rightarrow \mu \mid E[wr] \\
&\mu \mid E[max(rnd(n_1 \oplus n_2), 0)] \\
&\mu \mid E[\text{fix}(\lambda x : \tau. t)] \rightarrow E[t[\lambda x : \tau. t / x]] \\
&\mu' = \mu[l \mapsto v] \\
&\mu \mid E[l := v] \rightarrow \mu' \mid E[\text{unit}] \\
&\mu \mid E[\text{hash } l] \rightarrow E[l] \\
\end{align*}
\]

*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 \[19\]. Consider, for example, the following two contextually equivalent MiniML terms.

\((\lambda x : \text{Int.}(+ \cdot x)) \quad (\lambda x : \text{Int.}(\cdot \cdot x))\) (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.

\((\text{if } ((\lambda y : \cdot \cdot 2) x) \equiv_\alpha [] \Omega \text{ true})\)

The context distinguishes the two equivalent terms due to the \(\equiv_\alpha\) operator’s ability to inspect the syntax of MiniML terms, where \(\Omega\) is a diverging MiniML\(^a\) term. A similar context \(C\) 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 interaction with programs in MiniML. To do so we apply our previously developed language interoperation semantics [8]. While there exists many different multi-language semantics (Section 6), our interoperation semantics is the only one that supports separated program states and explicited 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 \(P\) of MiniML\(^+\) is split into two substates: the attacker state \(A\) and the MiniML program state \(S\).

\[ P = A \| S \]

The reduction rules for MiniML\(^+\) programs are denoted as follows.

\[
A \| S \rightarrow A' \| S'
\]

The MiniML state \(S\) 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.

\[
(1) N; \mu \vDash \Sigma \circ t : \tau \quad (2) N; \mu \vDash \Sigma \triangleright m : \tau \quad (3) N; \mu \vDash \Sigma \triangleleft m : \tau \quad (4) N; \mu \vDash \Sigma
\]

Where \(m = v | v\) 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 \(t\) or (2) is suspended waiting on input from the MiniML program.

\[
(1) A = \mu \vDash C \bullet t \quad (2) A = \mu \vDash C
\]

The states never compute concurrently. Whenever the MiniML state \(S\) computes, the attacker state \(A\) is suspended and vice-versa. Internal computations thus that only affect the terms of one of the two languages.

\[
\mu \vDash C \| N; \mu \vDash \Sigma \circ t : \tau \rightarrow \mu \vDash C' \| N; \mu' \vDash \Sigma \circ t' : \tau \quad \text{(Internal MiniML)}
\]

\[
\mu \vDash C \bullet t \| N; \mu \vDash \Sigma \rightarrow \mu' \vDash C' \bullet t' \| N; \mu \vDash \Sigma \quad \text{(Internal MiniML\(^a\))}
\]

*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 \(S\) encodes this call stack as a type annotated stack of evaluation contexts \(\Sigma ::= \overline{E} : \tau \rightarrow \tau' \mid \epsilon\), where \(\overline{E}\) 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, 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 $\overline{C}$, enabling it to attack each input or observation from the MiniML program.

**Marshalling** 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 $\overline{C}$. If the value is a location or a $\lambda$-term then it must be masked with a name $n^1_1$ or $n^2_1$, 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\textsuperscript{a} value. This conversion happens in a designated marshalling state as follows.

$$\mu \vDash \overline{C} \mid N; \mu \vDash \Sigma \circ v : \tau \rightarrow \mu \vDash \overline{C} \mid N; \mu \vDash \Sigma \triangleright v : \tau \quad \text{(Setup)}$$

To save space in the following marshalling rules, we have compressed the state $\mu \vDash \overline{C} \mid N; \mu \vDash \Sigma \triangleright m$ into a wrapper $\llbracket m \rrbracket^N_{\mu\Sigma}$ that denotes the only two constructs relevant to the marshalling process: the expected type $\tau$ and the map of shared names $N$.

$$\begin{align*}
\llbracket b \rrbracket^N_{\text{Bool}} &\rightarrow \llbracket b \rrbracket^N_{\text{Bool}} \\
\llbracket \text{unit} \rrbracket^N_{\text{unit}} &\rightarrow \llbracket \text{unit} \rrbracket^N_{\text{unit}} \\
\llbracket \pi \rrbracket^N_{\text{Int}} &\rightarrow \llbracket \pi \rrbracket^N_{\text{Int}} \\
\llbracket (\lambda x : \tau.t) \rrbracket^N_{\tau \rightarrow \tau'} &\rightarrow \llbracket n^2_1 \rrbracket^N_{\tau \rightarrow \tau'} \\
\llbracket (\lambda x : \tau.t) \rrbracket^N_{\tau \rightarrow \tau'} &\rightarrow \llbracket (\text{Ref}\, l) \rrbracket^N_{\tau \rightarrow \tau'} \\
\llbracket \text{Setup} \rrbracket^N_{\text{Setup}} &\rightarrow \llbracket n^1_1 \rrbracket^N_{\text{Ref}} \\
\llbracket \tau \rightarrow \tau_1 \rightarrow \tau_2 \rrbracket^N_{\tau} &\rightarrow \llbracket \lambda x : \tau.t \rrbracket^N_{\tau} \\
\llbracket \emptyset : t : \tau' \rightarrow \tau \rrbracket^N &\rightarrow \llbracket \text{wr} \rrbracket^N_{\tau}
\end{align*}$$

If the marshalling succeeds (there is no type error) the result is shared, otherwise the secure state is cleared and the attacker is updated with wrong: $\text{wr}$.

$$\begin{align*}
\mu \vDash \overline{C}, C \mid N; \mu \vDash \Sigma \triangleright v : \tau \rightarrow \mu \vDash \overline{C} \circ C[v] \mid N; \mu \vDash \Sigma \quad \text{(Share)} \\
\mu \vDash \overline{C}, C \mid N; \mu \vDash \Sigma \triangleright \text{wr} : \tau \rightarrow \mu \vDash \overline{C}, C[\text{wr}] \mid \star; \emptyset \vDash \varepsilon \quad \text{(Type Error)}
\end{align*}$$

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.

$$\mu \vDash \overline{C} \circ v \mid N; \mu \vDash \Sigma, \varepsilon : \tau \rightarrow \tau' \rightarrow \mu \vDash \overline{C} \mid N; \mu \vDash \Sigma \triangleright v : \tau \quad \text{(Input)}$$

The input value must be marshalled to the correct representation before it is plugged into the head of the stack of evaluation contexts $\Sigma$. Note that as denoted in the reduction rule $\text{Input}$ the marshalling rules will verify that the input value matches the argument type $\tau$ of the to be plugged evaluation context. The marshalling 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. To save space in the following marshalling rules, we have compressed the state \( \mu \vdash \mathcal{C} \ | \ N; \mu \vdash \Sigma \prec m \) into a wrapper \( \| m \|_{\tau}^{N} \) that denotes the only two constructs relevant to the marshalling in process: the expected type \( \tau \) and the map of shared names \( N \).

\[
\begin{align*}
\| b \|_{\text{Bool}}^{N} & \rightarrow \| b \|_{\text{Bool}}^{N} & t \neq b \\
\| t \|_{\text{Int}}^{N} & \rightarrow \| \text{wr} \|_{\text{Int}}^{N} & t \neq \text{\texttt{unit}} \\
\| \text{\texttt{unit}} \|_{\text{Unit}}^{N} & \rightarrow \| \text{\text{wr}} \|_{\text{Unit}}^{N} \quad & t \neq \text{\texttt{unit}} \\
\| \text{\texttt{unit}} \|_{\text{Unit}}^{N} & \rightarrow \| \text{\text{wr}} \|_{\text{Unit}}^{N} \\
\| l \|_{\text{Ref}}^{N} & \rightarrow \| \text{\text{wr}} \|_{\text{Ref}}^{N} & t \neq l \\
\| \text{\texttt{unit}} \|_{\text{Unit}}^{N} & \rightarrow \| \text{\text{wr}} \|_{\text{Unit}}^{N} \quad & t \neq \text{\texttt{unit}} \\
\| \text{\texttt{unit}} \|_{\text{Unit}}^{N} & \rightarrow \| \text{\text{wr}} \|_{\text{Unit}}^{N} \quad & t \neq \text{\texttt{unit}} \\
\end{align*}
\]

If the input doesn’t conform with the type annotated to the evaluation context the state \( S \) is cleared and the attacker terminates to \( \text{\texttt{wr}} \), otherwise the marshalled value is used to plug the evaluation context.

\[
\begin{align*}
\mu \vdash \mathcal{C}, \mathcal{C} | | N; \mu \vdash \Sigma \prec \text{\texttt{wr}} : \tau \rightarrow \mu \vdash \mathcal{C}, \mathcal{C}[\text{\texttt{wr}}] | | *; \emptyset \vdash \varepsilon & \quad \text{(Type-Error-In)} \\
\mu \vdash \mathcal{C} | | N; \mu \vdash \Sigma, E : \tau \rightarrow \tau' \prec \nu : \tau \rightarrow \mu \vdash \mathcal{C} | | N', \mu \vdash \Sigma \circ \nu [\varepsilon] : \tau' & \quad \text{(Plug)}
\end{align*}
\]

**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 \( n_{I}^{1} \) and \( n_{I}^{l} \) respectively. Both names are tracked in the MiniML state \( S \) through a map \( N \) that records the associated term and type, as follows.

\[
\begin{align*}
N & ::= * \ | \ N, n_{I}^{1} \mapsto (l, \tau) \ | \ N, n_{l}^{l} \mapsto (l, \tau)
\end{align*}
\]

A fresh name \( n_{I}^{1} \) 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 \( n_{i}^{l} \) will correspond to the index of the location it refers to \( (n_{l}^{l} \mapsto l_{i}) \).

The attacker interacts with these shared names as follows.

\[
\begin{align*}
\mu \vdash \mathcal{C} \bullet \text{\texttt{E[call}}} n_{I}^{1} \text{\texttt{v}} | | N; \mu \vdash \Sigma \rightarrow \mu \vdash \mathcal{C} \bullet \text{\texttt{v}} | | N; \mu \vdash \Sigma, (t [\cdot]) : \tau \rightarrow \tau' & \quad \text{(A-Call)} \\
\text{\text{\texttt{where}} } N(n_{I}^{1}) = (l, \tau \rightarrow \tau') \\
\mu \vdash \mathcal{C} \bullet \text{\texttt{E(set}}} n_{l}^{l} \text{\texttt{v}} | | N; \mu \vdash \Sigma \rightarrow \mu \vdash \mathcal{C} \bullet \text{\texttt{v}} | | N; \mu \vdash \Sigma, (t := [\cdot]) : \tau \rightarrow \text{Unit} & \quad \text{(A-Set)} \\
\text{\text{\texttt{where}} } N(n_{l}^{l}) = (l, \text{Ref} \, \tau)
\end{align*}
\]
A command from the attacker is not a direct 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 \( \Sigma \), but is instead executed on top the stack. Applying a \( \lambda \)-term (A-Call), writing to a shared location (A-Ref) 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 confirm to the typing rules of MiniML, the attacker is updated with \( \text{wr} \) (A-WrD,A-WrC,A-WrS). Dereferencing a shared MiniML location (A-Der) requires but one step as it involves only the shared name \( n_i \) and thus does not need to marshall out a value.

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

\[
\mu \vdash C \mid N; \mu \vdash \Sigma \circ !l : \tau \quad \text{(A-Der)}
\]

\[
\mu \vdash C \mid N; \mu \vdash \Sigma \rightarrow \mu \vdash C \mid N; \mu \vdash \Sigma \circ !l : \tau \quad \text{(A-Ref)}
\]

\[
\mu \vdash C, C \bullet \text{deref } n_i | N; \mu \vdash \Sigma \rightarrow \mu \vdash C \mid C[\text{wr}] | *; \emptyset \vdash \varepsilon \quad \text{(A-WrD)}
\]

\[
\mu \vdash C, C \bullet \text{call } n_i v | N; \mu \vdash \Sigma \rightarrow \mu \vdash C \mid C[\text{wr}] | *; \emptyset \vdash \varepsilon \quad \text{(A-WrC)}
\]

\[
\mu \vdash C, C \bullet \text{set } n_i v | N; \mu \vdash \Sigma \rightarrow \mu \vdash C \mid C[\text{wr}] | *; \emptyset \vdash \varepsilon \quad \text{(A-WrS)}
\]

3.4 \( B^{\alpha} \): 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 \( B^{\alpha} \). To do so we define an applicative bisimulation in the style of of Jeffrey’s and Rathke’s applicative bisimulation for the \( v_{\text{ref}} \)-calculus [7]. 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 \( (\mathcal{S}, \alpha, \rightarrow^\alpha) \) where the MiniML states \( \mathcal{S} \) of MiniML\(^a\) are the states of the LTS, \( \alpha \) the set of labels and \( \rightarrow^\alpha \) the labelled transitions between states. The attacker state \( A \) is thus not represented in these labelled reductions, instead the labels \( \alpha \), denote the observations of the high-level MiniML\(^a\) attacker as follows.
The labelled reductions of the LTS are of the form: $S \overset{\alpha}{\rightarrow} S'$. The transitions are as follows.

\[
\begin{align*}
\alpha & ::= \gamma | \tau | \sqrt{.} \\
\gamma & ::= v? | v! | w_r \mid \gg (\lambda x.t) \mid \gg n^1_i \mid \gg n^2_i \mid \gg \text{ref} \tau \mid \text{!} n^1_i
\end{align*}
\]

The labelled reductions of the LTS are of the form: $S \overset{\gamma}{\rightarrow} S'$. The transitions are as follows.

\[
\begin{align*}
N; \mu \models \Sigma \circ t: \tau & \overset{\tau}{\rightarrow} N; \mu' \models \Sigma \circ t': \tau \quad \text{(S-Inner)} \\
N; \mu \models \Sigma \circ v: \tau & \overset{\tau}{\rightarrow} N; \mu \models \Sigma \triangleright v: \tau \quad \text{(S-Setup)} \\
N; \mu \models \Sigma, E \triangleleft v: \tau & \overset{\tau}{\rightarrow} N; \mu \models \Sigma \circ E[v]: \tau \quad \text{(S-Plug)} \\
N; \mu \models \Sigma \triangleleft m: \tau & \overset{\tau}{\rightarrow} N; \mu \models \Sigma \triangleleft m': \tau \quad \text{(S-MarshIN)} \\
N; \mu \models \Sigma \triangleright m: \tau & \overset{\tau}{\rightarrow} N'; \mu \models \Sigma \triangleright m': \tau \quad \text{(S-MarshOut)} \\
N; \mu \models \Sigma, E: \tau \rightarrow \tau' \overset{v'}{\rightarrow} N; \mu \models \Sigma, E: \tau \rightarrow \tau' \triangleleft v: \tau \quad \text{(A-V)} \\
N; \mu \models \Sigma \triangleright v: \tau & \overset{v'}{\rightarrow} N; \mu \models \Sigma \quad \text{(M-V)} \\
N; \mu \models \Sigma \overset{w_r}{\rightarrow} \tau & \overset{w_r}{\rightarrow} \ast; \emptyset \models \varepsilon \quad \text{(Wr-O)} \quad N; \mu \models \Sigma \overset{w_r}{\rightarrow} ; \ast; \emptyset \models \varepsilon \quad \text{(Wr-I)} \\
N; \mu \models \Sigma, E : \tau \overset{\gg \text{ref} \tau}{\rightarrow} N; \mu \models \Sigma, (\text{ref }[\cdot]) : \tau \rightarrow \text{Ref } \tau \quad \text{(A-R)} \\
N; \mu \models \Sigma \overset{\gg n^1_i}{\rightarrow} N; \mu \models \Sigma \circ l_i : \tau \quad \text{where } N(n^1_i) = (l_i, \text{Ref } \tau) \quad \text{(D-N)} \\
N; \mu \models \Sigma \overset{\gg n^2_i}{\rightarrow} N; \mu \models \Sigma, (t[\cdot]) : \tau \rightarrow \tau' \quad \text{where } N(n^2_i) = (t, \tau \rightarrow \tau') \quad \text{(C-N)} \\
N; \mu \models \Sigma \overset{\gg l_i}{\rightarrow} N; \mu \models \Sigma, (l_i := [\cdot]) : \tau \rightarrow \text{Unit } \quad \text{if } N(n^1_i) = (l_i, \text{Ref } \tau) \quad \text{(S-N)} \\
N; \mu \models \Sigma \circ E[(\tau_1 \rightarrow \tau_2) F(\lambda x.t) \ v] : \tau \overset{\gg (\lambda x.t) \ v}{\rightarrow} N; \mu \models \Sigma, E : \tau_2 \rightarrow \tau \triangleright v : \tau_1 \quad \text{(C-L)}
\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-V). Likewise the inputs or returned values of the secure state, converted to MiniML* 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 $w_r$. 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 3.3 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* (C-L) the argument is marshalled first (S-MarshOut).

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 $S \xrightarrow{\gamma} S'$ as $S \xrightarrow{\tau} S'$ where $\xrightarrow{\tau}$ is the reflexive transitive closure of the silent transitions $\xrightarrow{\tau}$. Our bisimulation $B^a$ over the observations and inputs of the MiniML$^a$ attacker is now defined as follows.

**Definition 3.** The relation $B^a$ is a **bisimulation** iff $S_1 B^a S_2$ implies:

1. Given $S_1 \xrightarrow{\gamma} S_1'$ there is $S_2'$ such that: $S_2 \xrightarrow{\tau} S_2'$ and $S_1' B^a S_2'$
2. Given $S_2 \xrightarrow{\gamma} S_2'$ there is $S_1'$ such that: $S_1 \xrightarrow{\tau} S_1'$ and $S_1' B^a S_2'$

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 $B^a$ 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 4.** Contextual equivalence for MiniML$^+$ ($\simeq^a$) is defined as:

$$\begin{align*}
S_1 \simeq^a S_2 & \iff \forall A. (A \mid || S_1) \uparrow \iff (A \mid || S_2) \uparrow
\end{align*}$$

**Theorem 1 (Congruence of the Bisimilarity).** $S_1 \approx^a S_2 \iff S_1 \simeq^a S_2$.

The proof splits the thesis into two sublemma: completeness and soundness.

**Lemma 1.** **(Completeness)** $S_1 \approx^a S_2 \Rightarrow S_1 \simeq^a S_2$

**Proof.** To prove that contextual equivalence implies bisimilarity we show that the contextual equivalence relation ($\simeq^a$) is itself a bisimulation ($B^a$). Assume that: $S_1 \approx^a S_2$. Because bisimilarity is symmetrical, we divide the proof into two parts.

1. Assume that: $S_1 \xrightarrow{\gamma} S_1'$. We must show that there exists a $S_2$ such that (1) $S_2 \xrightarrow{\gamma} S_2'$ and that (2) $S_1' \approx^a S_2'$. The proof proceeds by case analysis on the labels $\gamma$. For the labels produced by the secure state $S$ we rely on the fact that it follows from the assumption $S_1 \approx^a S_2$ 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 $S_1 = \star; \mu \vdash \varepsilon$ and that $S_1' = \star; \emptyset \vdash \varepsilon$. From the assumption $S_1 \approx^a S_2$ we have that $S_2 = \star; \mu \vdash \varepsilon$ as otherwise there exists a context $A$ that can distinguish $S_1$ and $S_2$ as follows.

  - $N \neq \star$: In this case there is some name $n_1$ that the attacker $A$ can invoke as per the rules $C-N$ and $S-N$.  

15
\( \Sigma \not= \varepsilon \): In this case there is some input \( \nu \) that the secure context \( \Sigma \) accepts from the attacker \( \mathbf{A} \) as per the rule \( A-V \).

\( \star; \mu \vdash \varepsilon \circ t \ | \ \star; \mu \vdash \varepsilon \triangleright m \ | \ \star; \mu \vdash \varepsilon \triangleleft m \): In this case \( \Sigma_2 \) will either produce a value as per \( M-V \) which the attacker can observe, otherwise \( \Sigma_2 \) diverges in which case the attacker distinguishes the states by default.

It follows from the LTS that \( \Sigma_2 = \star; \mu \vdash \varepsilon \triangleright S_2 = \star; \emptyset \vdash \varepsilon \), we conclude that thesis (1) and (2) hold.

\(- \gamma = \nu! \): It follows from the LTS rule \( M-V \) that \( \Sigma_1 = \mathcal{N}; \mu \vdash \Sigma \triangleright \nu \) and that \( \Sigma'_1 = \mathcal{N}; \mu \vdash \Sigma \). From the assumption \( \Sigma_1 \simeq^a \Sigma_2 \) we have that \( \Sigma_2 = \nu'; \mu' \vdash \Sigma' \triangleright \nu' \) where \( \nu' = \nu \), \( \forall \nu_1 \in \text{Dom}(\mathcal{N}).\mathcal{N}(\nu_1) \simeq \nu'(\nu_1) \) and \( \forall \nu \in \Sigma \). \( \forall \nu' \in \Sigma'.\nu \simeq \nu' \): as otherwise there exists a context \( \mathbf{A} \) that can distinguish \( \Sigma_1 \) and \( \Sigma_2 \) as follows.

\( \Sigma_2 = \nu'; \mu' \vdash \Sigma' \triangleright \nu' \land \nu' \not= \nu \): The attacker \( \mathbf{A} = ([\nu]_{\equiv^a \nu}) \) can distinguish \( \Sigma_1 \) and \( \Sigma_2 \).

\( \Sigma_2 \uparrow \): \( \Sigma_1 \) does not diverge, the attacker does distinguishes them by default.

\( \mathcal{N} \not\equiv \mathcal{N}' \): In this case there is some name \( \nu_1 \) that the attacker \( \mathbf{A} \) can invoke as per the rules \( C-N \) and \( S-N \) to distinguish \( \Sigma_1 \) and \( \Sigma_2 \).

\( \Sigma \not\equiv \Sigma' \): In this case there is some input \( \nu \) that the secure context \( \Sigma \) accepts from the attacker \( \mathbf{A} \) as per the rule \( A-V \) that can be used to distinguish \( \Sigma_1 \) and \( \Sigma_2 \). It follows from the LTS that \( \Sigma_2 = \nu'; \mu' \vdash \Sigma' \triangleright \nu \Rightarrow \Sigma'_2 = \nu'; \mu' \vdash \Sigma' \), we conclude that thesis (1) and (2) hold.

\(- \gamma = \nu? \): It follows from the LTS rule \( A-V \) that \( \Sigma_1 = \mathcal{N}; \mu \vdash \Sigma \) and that \( \Sigma'_1 = \mathcal{N}; \mu \vdash \Sigma \). From the assumption \( \Sigma_1 \simeq^a \Sigma_2 \) we have that \( \Sigma_2 = \nu'; \mu' \vdash \Sigma' \) as otherwise there exists a context \( \mathbf{A} \) that can distinguish \( \Sigma_1 \) and \( \Sigma_2 \). The label \( \nu? \) can be encoded as a context \( \mathbf{A} = (\nu \bullet \nu) \), as contextual equivalence is closed under contexts we have that for the resulting \( \Sigma'_2 = \nu'; \mu' \vdash \Sigma' < \nu \) we can conclude that \( \Sigma'_1 \simeq^a \Sigma'_2 \).

\(- \gamma = \nu\text{ wr}: \) 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 \( \Sigma_2 \) doesn’t produce the failure state for these actions the states \( \Sigma_1 \) and \( \Sigma_2 \) can be distinguished contradicting the assumption \( \Sigma_1 \simeq^a \Sigma_2 \).

- \( \text{Wr-O} \): If \( \Sigma_2 \) does not reduce the program to the failure state the states \( \Sigma_1 \) and \( \Sigma_2 \) can be distinguished contradicting the assumption \( \Sigma_1 \simeq^a \Sigma_2 \).

- \( \text{Wr-I} \): similar to \( \text{Wr-O} \).
Lemma 2. (Soundness) \( S_1 \simeq S_2 \Rightarrow S_1 \approx S_2 \)

Proof. As per Definition 2 we have that the thesis \( S_1 \approx S_2 \) becomes:

\[
\forall A. A \mid| S_1 \iff A \mid| S_2
\]
The proof is divided into two cases, one case for each side of the co-implication.

1. ⇒ In this case the thesis is:

\[ \forall A. A \parallel S_1 \uparrow \Rightarrow A \parallel S_2 \uparrow. \]

The thesis can be redefined as:

\[ \forall A. \forall k \in \mathbb{N}. A \parallel S_1 \rightarrow^k A_1' \parallel S_1' \Rightarrow \forall m \in \mathbb{N}. A \parallel S_2 \rightarrow^m A_2' \parallel S_2'. \]

The proof proceeds by induction on \( m \).

Base case: \( m = 0 \). Straightforward: \( A \parallel S_2 \rightarrow^0 A \parallel S_2 \).

Inductive case: \( m = h + 1 \). The thesis is:

\[ A \parallel S_2 \rightarrow^{h+1} A_2' \parallel S_2'. \]

The inductive hypotheses (IH) is:

\[ \forall A. \forall k \in \mathbb{N}. A \parallel S_1 \rightarrow^k A_1' \parallel S_1' \Rightarrow A \parallel S_2 \rightarrow^h A_2' \parallel S_2'. \]

We know from this IH that:

\[ \exists A, S_1, A \parallel S_1 \rightarrow^h A_1' \parallel S_1' \rightarrow^{k-h} A_1' \parallel S_1'. \]

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 \( h + 1 \). There are two cases: either the attacker in MiniML 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. \( \exists \gamma. S_1^h \approx S_1'^h \).

By the assumption \( S_1 \approx S_2 \) we conclude that \( S_2^h \approx S_2'^h \) and \( S_1'^h \approx S_2'^h \). This, in conjunction with the IH, implies the thesis:

\[ A \parallel S_2 \rightarrow^{h+1} A_2' \parallel S_2'. \]

ii. \( \not\exists \gamma. M_1^h \Rightarrow S_1'^h \).

Per the definition of bisimulation we have that this is only possible if \( S_1^h \) diverges, more particularly the MiniML term \( t_1^h \) that it executes diverges. It follows from the assumption that \( S_1 \approx S_2 \) that \( S_2^h \approx S_2'^h \) and thus that: \( \not\exists \gamma. M_1^h \Rightarrow S_1'^h \).

(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. \( \exists \gamma. N \parallel t_1^h \Rightarrow S_1'^h \).

Because the observable label \( \gamma \) is produced by the respective attackers, we must thus show that: \( A_1^h = A_2^h \), where the existence of \( A_2^h \) derives from the induction hypothesis.

We know by the assumption: \( S_1 \approx S_2 \) that both attacker states were modified by the same stream of observable labels if there are any such labels: \( \exists k \in \mathbb{N}. k \leq h \land A \parallel S_1 \rightarrow^k A_1' \parallel S_1' \land A_1 \parallel S_1 \rightarrow^{h-k} A_1^h \parallel S_1^h \) where \( S_1 \Rightarrow S_1^h \) and that \( \exists k \in \mathbb{N}. k \leq h \land A \parallel S_2 \rightarrow^k A_2' \parallel S_2' \land A_2 \parallel S_2 \rightarrow^{h-k} A_2^h \parallel S_2^h \) where \( S_2 \Rightarrow S_2^h \). Combining the fact that the reduction rules of the MiniML+ calculus are deterministic and with the fact that the MiniML contexts are updated in the same way by identical labels \( \gamma \) we conclude that \( A_1^h = A_2^h \) and that \( N \parallel t_1^h \Rightarrow S_1'^h \). This implies the thesis.
ii. \( \not\exists \gamma. \mathbb{N} \vdash \Sigma^h \Rightarrow S^h \).

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 \( A || S_1 \) and \( A || S_2 \) diverge, which implies the thesis.

2. \( \Leftarrow \) As in case 1, *mutatis mutandis*.

4 A Bisimulation over the Assembly Language Attacker

In this section we introduce a bisimulation over the assembly language that captures its interacts 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 [15] 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. Additionaly \( \Lambda \) can be \((\text{unkown}, m, s)\) when modeling the attacker. The attacker thus does not feature and explicit state, instead the labels \( L \) capture its observations and inputs as follows.

\[
L ::= \alpha \mid \tau \\
\alpha ::= \sqrt{\gamma} \mid \gamma! \mid \gamma? \\
\gamma ::= \text{call } p(r) \mid \text{ret } p(r)
\]

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 \( \sqrt{\cdot} \) 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 Abstract Trace Semantics [15]).**

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

Where \( \simeq_l \) denotes contextual equivalence between low-level programs and where \( \text{Tr}(P) \) computes the traces of a program, with an initial state \( \Lambda(P) \) as follows.
\[ \text{Tr}(P) = \{ \gamma \mid \exists A'. A(P) \xrightarrow{\gamma} A' \} \]

Note that this trace semantics does not include explicit read or writes from the protected memory to the unprotected memory or read 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 \( B' \): 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 \( A \) of that semantics include many low-level details on the protected memory that are not relevant to the result of this paper. We thus define a bisimulation \( B' \) 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 \( (S, e, p) \): the MiniML state of MiniML\(^+\) extended with static set of entry points \( e \) and a stack of return pointers \( p \). The MiniML state \( S \) 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 \( p \) enables the MiniML program to return to the address of the attacker were a call to an entry point originated from.

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

To save space in the following marshalling rules, we compress the marshalling out state \( N; \mu \vdash \Sigma \frac{m}{\tau} \) into a wrapper \( \uparrow \frac{\mu}{\Sigma \frac{m}{\tau}} \) and the marshalling out state \( N; \mu \vdash \Sigma \frac{m}{\tau} \) into a wrapper \( \downarrow \frac{\mu}{\Sigma \frac{m}{\tau}} \). These denote only two constructs relevant to the marshalling process: the expected type \( \tau \) and the map of shared names \( N \).

Note that we denote the conversion of numbers to hex as \( \text{hex} \{\cdot\} \).

\[
\begin{align*}
\uparrow \text{true}^N_{\text{Bool}} & \rightarrow \uparrow \text{0x1}^N_{\text{Bool}} \\
\uparrow \text{false}^N_{\text{Int}} & \rightarrow \uparrow \text{hex}\{n\}^N_{\text{Int}} \\
\tau & \rightarrow \tau_1 \rightarrow \tau_2 \\
\uparrow \langle P \rangle^N_\tau & \rightarrow \uparrow \langle P \rangle^N_{\tau_1} \\
\uparrow \langle \lambda x : \tau.t \rangle^N_\tau & \rightarrow \uparrow \langle \lambda x : \tau_1.t \rangle^N_{\tau_1} \rightarrow \uparrow \text{hex}\{j\}^N_{\tau_1} \\
\uparrow \langle 0 \vdash t : \tau' \mid \tau \neq \tau' \rangle^N_\tau & \rightarrow \uparrow \langle \text{wr} \rangle^N_{\tau_1} \\
\uparrow \langle l_i \rangle^N_{\text{Ref}} & \rightarrow \uparrow \langle \text{hex}\{i\} \rangle^N_{\text{Ref}} \\
\uparrow \text{true}^N_{\text{Bool}} & \rightarrow \uparrow \text{false}^N_{\text{Bool}}
\end{align*}
\]
The transitions are as follows.

$w \neq \text{0x1 or } w \neq \text{0x0}$

$\downarrow\uparrow^\text{0x1}_\text{Int} \rightarrow \downarrow\uparrow^\text{0x0}_\text{Int}$

$w \neq \text{hex}\{n\}$

$\downarrow\uparrow^\text{hex}\{n\}_\text{Int} \rightarrow \downarrow\uparrow^\text{hex}\{m\}_\text{Int}$

$w \neq \text{0xDEAD}$

$\downarrow\uparrow^\text{0xDEAD}_\text{Int} \rightarrow \downarrow\uparrow^\text{0xDEAD}_\text{Int}$

$\downarrow\uparrow^\text{hex}\{i\}_\text{Int} \rightarrow \downarrow\uparrow^\text{hex}\{j\}_\text{Int}$

$w \neq \text{pF or } w = \text{hex}\{i\} \cap \text{N}_i \neq \text{N}$

$\downarrow\uparrow^\text{hex}\{i\}_\text{Int} \rightarrow \downarrow\uparrow^\text{hex}\{j\}_\text{Int}$

$w = \text{hex}\{i\} \cap \text{N}_i \neq \text{N}$

$\downarrow\uparrow^\text{hex}\{i\}_\text{Int} \rightarrow \downarrow\uparrow^\text{hex}\{j\}_\text{Int}$

Labelled transition system The bisimulation $B^L$ is now over defined an LTS ($\langle S, e, \overline{p}, L, \overset{L}{\rightarrow}\rangle$, where $L$ are the labels of the fully abstract trace semantics an $\overset{L}{\rightarrow}$ denotes the labelled transitions between the states is $\langle S, e, \overline{p}\rangle \overset{L}{\rightarrow} \langle S', e, \overline{p}'\rangle$.

The transitions are as follows.

$\langle\langle N; \mu \mid \Sigma \circ t : \tau \rightarrow N; \mu' \mid \Sigma \circ t' : \tau\rangle\rangle \overset{\tau}{\rightarrow} \langle\langle N; \mu' \mid \Sigma \circ t' : \tau \rangle\rangle$ (S-Inner)

$\langle\langle N; \mu \mid \Sigma \circ v : \tau, p_r : \overline{p}\rangle\rangle \overset{\tau}{\rightarrow} \langle\langle N; \mu \mid \Sigma \circ v : \tau \rangle\rangle$ (S-Setup)

$\langle\langle N; \mu \mid \Sigma \circ E \circ v : \tau, p_r : \overline{p}\rangle\rangle \overset{\tau}{\rightarrow} \langle\langle N; \mu \mid \Sigma \circ E \circ v : \tau \rangle\rangle$ (S-Plug)

$\langle\langle N; \mu \mid \Sigma \circ m : \tau \rightarrow N; \mu \mid \Sigma \circ m' : \tau\rangle\rangle \overset{\tau}{\rightarrow} \langle\langle N; \mu \mid \Sigma \circ m' : \tau \rangle\rangle$ (S-MarshIN)

$\langle\langle N; \mu \mid \Sigma \circ m : \tau \rightarrow N; \mu \mid \Sigma \circ m' : \tau\rangle\rangle \overset{\tau}{\rightarrow} \langle\langle N; \mu \mid \Sigma \circ m' : \tau \rangle\rangle$ (S-MarshOut)

$\langle\langle N; \mu \mid \Sigma \circ t : \tau, e, \emptyset\rangle\rangle \overset{\text{call } p_{\text{start}}(p_r)^?}{\rightarrow} \langle\langle N; \mu \mid \Sigma \circ t : \tau, e, p_r : \emptyset\rangle\rangle$ (A-Start)

$\langle\langle N; \mu \mid \Sigma \circ w : \tau, e, p_r : \overline{p}\rangle\rangle \overset{\text{ret } p_r(w)}{\rightarrow} \langle\langle N; \mu \mid \Sigma \circ w : \tau, e, \overline{p}\rangle\rangle$ (M-Ret)

$\langle\langle N; \mu \mid \Sigma, E : \tau \rightarrow \tau', e, \overline{p}\rangle\rangle \overset{\text{ret } p_{\text{start}}(w)}{\rightarrow} \langle\langle N; \mu \mid \Sigma, E : \tau \rightarrow \tau', e, \overline{p}\rangle\rangle$ (A-R)

$\langle\langle N; \mu \mid \Sigma, e, \overline{p}\rangle\rangle \overset{\text{call } p_{\text{next}}(w_n, p_r)^?}{\rightarrow} \langle\langle N; \mu \mid \Sigma \circ !l_i : \tau, e, p_r : \overline{p}\rangle\rangle$ (A-Deref)

where $\text{N}(w_n) = (l_i, \text{Ref } \tau)$

$\langle\langle N; \mu \mid \Sigma, e, \overline{p}\rangle\rangle \overset{\text{call } p_{\text{next}}(w_n, w, p_r)^?}{\rightarrow}$ (A-Apply)

$\langle\langle N; \mu \mid \Sigma, (t \mid ) : \tau \rightarrow \tau' \circ w : \tau, e, p_r : \overline{p}\rangle\rangle$ where $\text{N}(w_n) = (t, \tau \rightarrow \tau')$

21
Transitions within the MiniML program, such as for example S-Inner, are not observable the attacker and are thus again labelled as silent (S-\textdagger). To start the computation of the MiniML program, the low-level attacker calls the entry point \( p_{\text{entry}}^{\text{star}} \), passing as its only argument \( p \), the address at which it expects the result returned (A-Start). When the MiniML program returns to that address (M-Return), 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 values, \( w \), 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 \( p_{\text{return}}^{\text{star}} \) (A-R). Whereas each operation by the high-level attacker on the MiniML terms shared to it through names \( n \) was denoted with its own label, the assembly language attacker calls a separate entry point for each operation (A-Deref, A-Set, A-Ref, A-Apply) passing a byte word representation of the names \( (w_n) \) as an argument to the call. In the case of A-Ref we make use of a conversion function \( \text{convt} \) to convert a hexadecimal value into a type \( \tau \). 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 \( (\,* ; \, \emptyset \, \vdash \, \varepsilon \, , \, e \, , \, \emptyset \, ) \). 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 \textit{weak} bisimulation, that does not take into account the silent transitions \( \tau \) only the actions \( \alpha \), over the LTS. Define the transition relation \( (S, e, \overline{p}) \xrightarrow{\alpha} (S', e, \overline{p}') \) as \( (S, e, \overline{p}) \xrightarrow{\alpha} (S', e, \overline{p}') \) where \( \xrightarrow{\alpha} \) is the reflexive transitive closure of the silent transitions \( \xrightarrow{\tau} \).

\textbf{Definition 5.} \( B^I \) is a bisimulation iff \( (S_1, e_1, \overline{p}_1) \, B \, (S_2, e_2, \overline{p}_2) \) implies:
1. Given \( \langle S_1, e_1, p_1 \rangle \overset{\alpha}{\Rightarrow} \langle S'_1, e_1, p'_1 \rangle \), There is \( \langle S'_2, e_2, p'_2 \rangle \) such that \( \langle S_2, e_2, p_2 \rangle \overset{\alpha}{\Rightarrow} \langle S'_2, e_2, p'_2 \rangle \) and \( \langle S'_1, e_1, p'_1 \rangle B \langle S'_2, e_2, p'_2 \rangle \)

2. Given \( \langle S_2, e_2, p_2 \rangle \overset{\alpha}{\Rightarrow} \langle S'_2, e_2, p'_2 \rangle \), There is \( \langle S'_1, e_1, p'_1 \rangle \) such that
\( \langle S_1, e_1, p_1 \rangle \overset{\alpha}{\Rightarrow} \langle S'_1, e_1, p'_1 \rangle \) and \( \langle S'_1, e_1, p'_1 \rangle B \langle S'_2, e_2, p'_2 \rangle \)

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

### 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).** \( \{t_1\}^\uparrow \approx^a \{t_2\}^\uparrow \iff \{t_1\}^\downarrow \approx^f \{t_2\}^\downarrow \)

Where \( \{t\}^\uparrow \) denotes the start state: \( (\star; 0 \vdash e \circ t : \tau) \) of an MiniML term \( t \) when faced with the MiniML\(^a\) attacker and where \( \{t\}^\downarrow \) denotes the start state: \( ((\star; 0 \vdash e \circ t : \tau), e, 0) \) of an MiniML term when faced with the assembly language attacker.

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

**Lemma 3. (Preservation)** \( \{t_1\}^\uparrow \approx^a \{t_2\}^\uparrow \Rightarrow \{t_1\}^\downarrow \approx^f \{t_2\}^\downarrow \).

**Proof.** We must develop a relation \( \mathcal{R} \) such that:

\[
\{t_1\}^\downarrow \mathcal{R} \{t_2\}^\downarrow
\]  

and that for all \( M_1 \mathcal{R} M_2 \) we have that:

\[
M_1 \xRightarrow{\mathcal{I}} M'_1 \text{ and } \exists M'_2, M_2 \xRightarrow{\mathcal{I}} M'_2 \Rightarrow M'_1 \mathcal{R} M'_2
\]  

and that for all \( M_1 \mathcal{R} M_2 \) we have that:

\[
M_2 \xRightarrow{\mathcal{I}} M'_2 \text{ and } \exists M'_1, M_1 \xRightarrow{\mathcal{I}} M'_1 \Rightarrow M'_1 \mathcal{R} M'_2
\]

We define \( \mathcal{R} \) as \( \mathcal{R} = \mathcal{R}_0 \cup \mathcal{R}_1 \cup \mathcal{R}_2 \cup \mathcal{R}_3 \):

\[
\mathcal{R}_0 = \{(\langle N_1; \mu_1 \vdash E_1 \rangle, e_1, p_1), (\langle N_2; \mu_2 \vdash E_2 \rangle, e_2, p_2) \mid \forall N_1, N_2, E_1, E_2, \mu_1, \mu_2, p_1, p_2 \\
\text{mi such that } N_1 \approx^N N_2 \text{ and } \text{Dom}(\mu_1) = \text{Dom}(\mu_2) \text{ and } E_1 \approx^E E_2 \text{ and } p_1 = p_2 \text{ and } e_1 = e_2\}
\]

\[
\mathcal{R}_1 = \{(\langle N_1; \mu_1 \vdash E_1 \circ t_1 : \tau_1 \rangle, e_1, p_1), (\langle N_2; \mu_2 \vdash E_2 \circ t_2 : \tau_2 \rangle, e_2, p_2) \mid \forall N_1, N_2, E_1, E_2, \mu_1, \mu_2, t_1, t_2, \tau_1, \tau_2, p_1, p_2, e_1, e_2, \text{ such that } \\
(\langle N_1; \mu_1 \vdash E_1 \rangle, (\langle N_2; \mu_2 \vdash E_2 \rangle, e_2, p_2) \in \mathcal{R}_0 \text{ and } t_1 \simeq t_2 \text{ and } \tau_1 = \tau_2\}
\]

23
\[ \mathcal{R}_2 = \{(\langle N_1; \mu_1 \models \Sigma_1 \triangleright m_1 : \tau_1 \rangle, e_1, p_1^1), \langle \langle N_2; \mu_2 \models \Sigma_2 \triangleright m_2 : \tau_2 \rangle, e_2, p_2^1 \rangle) \mid \forall N_1, N_2 \]

\[ \Sigma_1, \Sigma_2, \mu_1, \mu_2, m_1, m_2, \tau_1, \tau_2, p_1^1, p_2^1, e_1, e_2 \]

such that \((\langle N_1; \mu_1 \models \Sigma_1 \rangle, e_1, p_1^1), \langle \langle N_2; \mu_2 \models \Sigma_2 \rangle, e_2, p_2^1 \rangle) \in \mathcal{R}_0 \)

and \(m_1 \approx_m m_2\) and \(\tau_1 = \tau_2\)

\[ \mathcal{R}_3 = \{(\langle N_1; \mu_1 \models \Sigma_1 \triangleright m_1 : \tau_1 \rangle, e_1, p_1^1), \langle \langle N_2; \mu_2 \models \Sigma_2 \triangleright m_2 : \tau_2 \rangle, e_2, p_2^1 \rangle) \mid \forall N_1, N_2 \]

\[ \Sigma_1, \Sigma_2, \mu_1, \mu_2, m_1, m_2, \tau_1, \tau_2, p_1^1, p_2^1, e_1, e_2 \]

such that \((N_1; \mu_1 \models \Sigma_1, N_2; \mu_2 \models \Sigma_2) \in \mathcal{R}_0 \)

and \(m_1 \approx_m m_2\) and \(\tau_1 = \tau_2\)

where \(\approx_N\) is defined as:

\[ N_1 \approx_N N_2 \iff \text{dom}(N_1) = \text{dom}(N_2) \text{ and } \forall n_1 \in \text{dom}(N_1). N_1(n_1) \simeq N_2(n_1) \]

where \(\approx_e\) is defined as:

\[ \Sigma_1 \approx_e \Sigma_2 \iff \]

\[ \text{For } (E_1^1, \ldots, E_n^1) : (\tau_1^1, \ldots, \tau_n^1) \text{ in } \Sigma_1 \text{ and } (E_1^2, \ldots, E_n^2) : (\tau_1^2, \ldots, \tau_n^2) \text{ in } \Sigma_2 \]

we have \(n = n'\) and for every \(1 \leq i \leq n : \tau_i^1 = \tau_i^2\) and

\[ \forall t', t'' : \tau_i^1 \land \Gamma \vdash t'' ; \tau_i^1 \land \Gamma \vdash t'' \implies E_1^1[t'] \approx E_2^2[t''] \]

and where \(\approx_m\) is defined as:

\[ m_1 \approx_m m_2 \iff \]

\[ (m_1 = t_1 \land m_2 = t_2 \land t_1 \approx t_2) \]

\[ \lor (m_1 = w_1 \land m_2 = w_2 \land w_1 = w_2) \]

\[ \lor (m_1 = \text{wr} \land m_2 = \text{wr}) \]

\[ \lor (m_1 = \langle m_i^1 \in 1..n \rangle \land m_2 = \langle m_i^2 \in 1..n \rangle \land \forall i \in 1..n. m_i \approx_m m_i^1) \]

We now proof the three cases.

- In case (1) we have that \(\langle \ast ; \emptyset \models e ; \emptyset \circ t_1 \rangle, e_1, \emptyset \rangle \mathcal{R} \langle \ast ; \emptyset \models e \circ t_2, e_2, \emptyset \rangle\) as we have that \(t_1 \approx t_2\) from the assumption [8].

- In case (2) we assume \(M_1 \mathcal{R} M_2\) and that \(M_1 \Downarrow M_1\). We proceed by case analysis on \(L\).

1. Transition \text{wr-1}:

\[ (M_1 =) \langle \langle N_1; \mu_1 \models \Sigma_1 \triangleright \text{wr} : \tau_1 \rangle, e_1, p_1 \rangle \xrightarrow{\text{wr}} (M'_1 =) \langle \ast ; \emptyset \models e \rangle, \emptyset \rangle \]

It follows from \(M_1 \mathcal{R} M_2\) more specifically \(M_1 \mathcal{R}_3 M_2\) that:

\[ M_2 = \langle \langle N_2; \mu_2 \models \Sigma_2 \triangleright \text{wr} : \tau_2 \rangle, e_2, p_2 \rangle \]

24
Where \( N_1 \approx_N N_2, \) \( \text{Dom}(\mu_1) = \text{Dom}(\mu_2), \Sigma_1 \approx \Sigma_2 \) and \( \overline{p}_1 = \overline{p}_2 \) and \( e_1 = e_2 \) and that \( \tau_1 = \tau_2 \). By the LTS we have that only rule \( \text{Wr-I} \) applies as follows:

\[
M_2 \xrightarrow{\vee} (M_2' =) (((\star; \emptyset \models \varepsilon), e_2, \emptyset))
\]

Given that \( \star \approx_N \star, \) \( \text{Dom}(\emptyset) = \text{Dom}(\emptyset) \) and \( \varepsilon \approx \varepsilon \) and \( \emptyset = \emptyset \) we conclude that \( M'_1 \mathcal{R}_0 M'_2 \).

2. Transition \( \text{Wr-O} \): similar to the \( \text{Wr-I} \) case.

3. Transition Done:

\[
(M_1 =) (((\star; \emptyset \models \varepsilon), e_2, \emptyset)) \xrightarrow{\vee} (M_1' =) (((\star; \emptyset \models \varepsilon), \emptyset))
\]

It follows from \( M_1 \mathcal{R} M_2 \) more specifically \( M_1 \mathcal{R}_0 M_2 \) that:

\[
M_2 = (((\star; \emptyset \models \Sigma_2), e_2, \emptyset))
\]

where \( e_1 = e_2 \). By the LTS we have that only rule \( \text{Done} \) applies as follows:

\[
M_2 \xrightarrow{\vee} (M_2' =) (((\star; \emptyset \models \varepsilon), e_2, \emptyset))
\]

Given that \( \star \approx_N \star, \) \( \text{Dom}(\emptyset) = \text{Dom}(\emptyset) \) and \( \varepsilon \approx \varepsilon \) and \( \emptyset = \emptyset \), we conclude that \( M'_1 \mathcal{R}_0 M'_2 \).

- \( L = ret\ p^{e}_{\text{retb}}(w)? \): By the LTS we have two sub-cases:
  1. Transition A-R:

\[
(M_1 =) (((N_1; \mu_1 \models \Sigma_1, E_1 : \tau_1 \rightarrow \tau_1'), e_1, \overline{p}_I)) \xrightarrow{\text{ret } p^{e}_{\text{retb}}(w)?} (M_1' =) (((N_1; \mu_1 \models \Sigma_1, E_1 : \tau_1 \rightarrow \tau_1' < w : \tau_1), e_1, \overline{p}_I))
\]

It follows from \( M_1 \mathcal{R} M_2 \) more specifically \( M_1 \mathcal{R}_0 M_2 \) that:

\[
M_2 = (((N_2; \mu_2 \models \Sigma_2, E_2 : \tau_2 \rightarrow \tau_2'), e_2, \overline{p}_2))
\]

Where \( N_1 \approx_N N_2, \) \( \text{Dom}(\mu_1) = \text{Dom}(\mu_2), e_1 = e_2, \Sigma_1 \approx \Sigma_2 \) and \( \tau_1 \rightarrow \tau_2' \) and thus that \( \tau_1 = \tau_2 \) and \( \tau_1' = \tau_2' \) and \( \overline{p}_I = \overline{p}_2 \). By the LTS we have that many rules including A-R apply and thus that \( M_2 \) can produce the same label:

\[
M_2 \xrightarrow{\text{ret } p^{e}_{\text{retb}}(w)?} (S_2' =) (((N_2; \mu_2 \models \Sigma_2, E_2 : \tau_2 \rightarrow \tau_2' < w : \tau_2), e_2, \overline{p}_2))
\]

Given that \( N_1 \approx_N N_2, \) \( \text{Dom}(\mu_1) = \text{Dom}(\mu_2), \Sigma_1 \approx \Sigma_2 \) and \( \overline{p}_1 = \overline{p}_2 \) and \( e_1 = e_2 \) and \( w = w \), we conclude that \( M'_1 \mathcal{R}_3 M'_2 \).

25
2. Transition \( \text{Wr-R} \):

\[
(M_1) = ((N_1; \mu_1 \models \Sigma_1, E_1 : \tau_1 \rightarrow \tau'_1), e_1, \overline{p_1}) \xrightarrow{\text{ret } p(w)^?} (M'_1) = ((\ast; \emptyset \models \varepsilon), e_1, \emptyset) \text{ where } p \neq p^\text{retb}
\]

It follows from \( M_1 \mathcal{R} M_2 \) more specifically \( M_1 \mathcal{R}_0 M_2 \) that:

\[
M_2 = ((N_2; \mu_2 \models \Sigma_2, E_2 : \tau_2 \rightarrow \tau'_2), e_2, \overline{p_2})
\]

This state does not enforce any restrictions on the abilities of the attacker to call the wrong address as such we conclude that \( \text{Wr-R} \) applies to \( M_2 \) as well.

\[
M_2 \xrightarrow{\text{ret } p(w)^?} (M'_2) = ((\ast; \emptyset \models \varepsilon), e_2, \emptyset) \text{ where } p \neq p^\text{retb}
\]

Given that \( \ast \approx_{M_1} \ast, \text{Dom}(\emptyset) = \text{Dom}(\emptyset) \) and \( \varepsilon \approx \varepsilon \) and \( \emptyset = \emptyset \), we conclude that \( M'_1 \mathcal{R}_0 M'_2 \).

\bullet \ L = \text{ret } p(w)!: \text{ By the LTS we have that this label applies only to the transition } \text{M-Ret}:

\[
(M_1) = ((N_1; \mu_1 \models \Sigma_1 \triangleright w_1 : \tau_1), e_1, p_{r_1} : \overline{p_1}) \xrightarrow{\text{ret } p_{r_1} (w_1)^!} (M'_1) = ((N_1; \mu_1 \models \Sigma_1), e_1, \overline{p_1})
\]

It follows from \( M_1 \mathcal{R} M_2 \) more specifically \( M_1 \mathcal{R}_2 M_2 \) that:

\[
M_2 = ((N_2; \mu_2 \models \Sigma_2 \triangleright w_2 : \tau_2), e_2, p_{r_2} : \overline{p_2})
\]

Where \( N_1 \approx_{N} N_2, \text{Dom}(\mu_1) = \text{Dom}(\mu_2), e_1 = e_2, \ \Sigma_1 \approx_{\Sigma} \Sigma_2 \) and \( \overline{p_1} = \overline{p_2} \) and \( p_{r_1} = p_{r_2} \) and that \( \tau_1 = \tau_2 \). By the LTS we have that only \( \text{M-Ret} \) applies and that it will produce the same label:

\[
M_2 \xrightarrow{\text{ret } p_{r_1} (w_1)^!} (M'_2) = ((N_2; \mu_2 \models \Sigma_2), e_2, \overline{p_2})
\]

Given that \( N_1 \approx_{N} N_2, \text{Dom}(\mu_1) = \text{Dom}(\mu_2), \ \Sigma_1 \approx_{\Sigma} \Sigma_2, e_1 = e_2, \text{ and } \overline{p_1} = \overline{p_2} \), we conclude that \( M'_1 \mathcal{R}_0 M'_2 \).

\bullet \ L = \text{call } p(\overline{w})?: \text{ By the LTS we have } 5 \text{ sub-cases:}

1. Transition \( \text{A-Start} \):

\[
(M_1) = ((N_1; \mu_1 \models \Sigma_1 \circ t_1 : \tau_1), e_1, \emptyset) \xrightarrow{\text{call } p^\text{start}(p_r)^?} (M'_1) = ((N_1; \mu_1 \models \Sigma_1 \circ t_1 : \tau_1), e_1, p_r : \emptyset)
\]

It follows from \( M_1 \mathcal{R} M_2 \) more specifically \( M_1 \mathcal{R}_1 M_2 \) that:

\[
M_2 = ((N_2; \mu_2 \models \Sigma_2 \circ t_2 : \tau_2), e_2, \emptyset)
\]

Where \( N_1 \approx_{N} N_2, \text{Dom}(\mu_1) = \text{Dom}(\mu_2), \Sigma_1 \approx_{\Sigma} \Sigma_2, e_1 = e_2, t_1 \approx t_2 \) and \( \tau_1 = \tau_2 \). By the LTS we have that transition \( \text{A-Start} \) applies to \( M_2 \) given that \( \overline{p_2} = \emptyset \).

\[
M_2 \xrightarrow{\text{call } p^\text{start}(p_r)^?} (M'_2) = ((N_2; \mu_2 \models \Sigma_2 \circ t_2 : \tau_2 \rightarrow \tau'_2), e_2, p_r : \emptyset)
\]

Given that \( (p_r : \emptyset) = (p_r : \emptyset) \), we conclude that \( M'_1 \mathcal{R}_1 M'_2 \).

26
2. Transition A-Ref:

\[
(M_1 =) ((N_1; \mu_1 \vdash \Sigma_1), \overline{p_f}) \xrightarrow{\text{call } p_{\text{ret}}(w, w, p_r)} (M'_1 =) ((N_1; \mu_1 \vdash \Sigma_1, (\text{ref } [\cdot]): \tau \rightarrow \text{Ref } \tau < w : \tau), p_r : \overline{p_f})
\]

where \( \text{convt}(w_i) = \tau_i \)

It follows from \( M_1 \mathcal{R} M_2 \) more specifically \( M_1 \mathcal{R}_0 M_2 \) that:

\[
(M_2 =) ((N_2; \mu_2 \vdash \Sigma_2), \overline{p_f})
\]

Where \( N_1 \approx_N N_2 \), \( \text{Dom}(\mu_1) = \text{Dom}(\mu_2) \), \( \Sigma_1 \approx \Sigma_2 \) and \( p_1 = p_2 \). By the LTS we have that several transitions apply including A-Ref:

\[
M_2 \xrightarrow{\text{call } p_{\text{ret}}(w, w, p_r)} (M'_2 =) ((N_2; \mu_2 \vdash \Sigma_2, (\text{ref } [\cdot]): \tau \rightarrow \text{Ref } \tau < w : \tau), p_r : \overline{p_f})
\]

where \( \text{convt}(w_i) = \tau \)

It no follows from the fact that \( \Sigma_1, (\text{ref } [\cdot]): \tau \rightarrow \text{Ref } \tau \approx_E \Sigma_2, (\text{ref } [\cdot]): \tau \rightarrow \text{Ref } \tau \) and that \( w = w \) that \( M_1 \mathcal{R}_3 M'_2 \).

3. Transition A-Set:

\[
(M_1 =) ((N_1; \mu_1 \vdash \Sigma_1), e_1, \overline{p_f}) \xrightarrow{\text{call } p_{\text{ret}}(w, w, p_r)} (M'_1 =) ((N_1; \mu \vdash \Sigma_1, (\text{ref } [\cdot]): \tau \rightarrow \text{Unit } < w : \tau_1), e_1, p_r : \overline{p_f})
\]

where \( N(w_n) = (l_i, \text{Ref } \tau_1) \)

It follows from \( M_1 \mathcal{R} M_2 \) more specifically \( M_1 \mathcal{R}_0 M_2 \) that:

\[
M_2 = ((N_2; \mu_2 \vdash \Sigma_2, E_2: \tau_2 \rightarrow \tau'_2), e_2, \overline{p_f})
\]

Where \( N_1 \approx_N N_2 \), \( \text{Dom}(\mu_1) = \text{Dom}(\mu_2) \), \( e_1 = e_2 \), \( \Sigma_1, E_1: \tau_1 \rightarrow \tau'_1 \approx_E \Sigma_2, E_2: \tau_2 \rightarrow \tau'_2 \) and thus that \( \tau_1 = \tau_2 \) and \( \tau'_1 = \tau'_2 \) and \( p_1 = p_2 \). From the LTS it follows that the same transition A-Set is possible.

\[
(M_2 =) \xrightarrow{\text{call } p_{\text{ret}}(w, w, p_r)} (M'_2 =) ((N_2; \mu_2 \vdash \Sigma_2, (\text{ref } [\cdot]): \tau \rightarrow \text{Unit } < w : \tau_2), e_2, p_r : \overline{p_f})
\]

where \( N(w_n) = (l_j, \text{Ref } \tau_2) \)

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 hash operator we have that \( j = i \) and thus that \( (l_j := [\cdot]) \simeq (l_i := [\cdot]) \).

We conclude that \( M'_1 \mathcal{R}_3 M'_2 \).

4. Transition A-Deref: similar to A-Set.
5. Transition A-Apply:

\[(M_1 =) ((N_1; \mu_1 \vdash \Sigma_1), e_1, \{n\}) \xrightarrow{\text{call } \rho_{\text{app}}(w_n,w,p_r)} (M_1') =) ((N_1; \mu_1 \vdash \Sigma_1, (t_1 \mid \cdot) : \tau_1 \rightarrow \tau_1' \triangleleft w : \tau_1), e, p_r : \{n\})\]

where \(N(w_n) = (t, \tau \rightarrow \tau')\)

It follows from \(M_1 \mathcal{R} M_2\) more specifically \(M_1 \mathcal{R}_0 M_2\) that:

\[(M_2 =) \xrightarrow{\text{call } \rho_{\text{app}}(w_n,w,p_r)} (M_2' =) ((N_2; \mu_2 \vdash \Sigma_2, E_2 : \tau_2 \rightarrow \tau_2') \vdash e, p_r : \{n\})\]

where \(N(w_n) = (t_2, \tau_2 \rightarrow \tau_2')\)

By the fact that \(N_1 \approx_N N_2\), it follows that \(t_1 \approx t_2\) and thus that \(\tau_2 = \tau_1\) and \(\tau_1' = \tau_2'\). We conclude that \(M_1' \mathcal{R}_0 M_2'\).

- \(L = \text{call } p(w)\): By the LTS we that this label applies only to the transition M-Call:

\[(M_1 =) ((N_1; \mu_1 \vdash \Sigma_1 \circ E_1[\tau' \mapsto \tau' F_{p_f} v_1] : \tau), \{n\})\]

\[(M_1' =) ((N_1; \mu_1 \vdash \Sigma_1, E_1 : \tau' \rightarrow \tau), \{n\})\]

\[(\text{M-Call}) \quad N_1; \mu_1 \vdash \Sigma_1 \circ E_1[\tau' \mapsto \tau' F_{p_f} v_1] : \tau_1 \rightarrow N_1; \mu_1 \vdash \Sigma_1, E_1 : \tau' \rightarrow \tau_1 \triangleright v_1 : \tau''\]

\[(\text{M-Call}) \quad N_1; \mu_1 \vdash \Sigma_1, E_1 : \tau' \rightarrow \tau_1 \triangleright v_1 : \tau'' \rightarrow^* N_1; \mu_1 \vdash \Sigma_1, E_1 : \tau' \rightarrow \tau_1 \triangleright w_1 : \tau''\]

\[(M_1 \xrightarrow{\text{call } p_f(w)} M_1')\]

It follows from \(M_1 \mathcal{R} M_2\) more specifically \(M_1 \mathcal{R}_1 M_2\) that:

\[(M_2 =) ((N_2; \mu_2 \vdash \Sigma_2 \circ t_2 : \tau_2), \{n\})\]

Where \(N_1 \approx_N N_2\), \(\text{Dom}(\mu_1) = \text{Dom}(\mu_2), E_1 \approx_E E_2, \Sigma_1 = \Sigma_2, \{n\} = \{n\}, \tau_1 = \tau_2\) and \(t_2 \approx E_1[\tau'' \mapsto \tau' F_{p_f} v_1]\). It follows from the fact that the bisimulation starts from a state \(\{t\}^+\), where \(t\) is a pure MiniML term, that \(\tau'' \rightarrow^* \tau' F_{p_f}\) derives from the attacker. It now follows from the fact that \(E_1[\tau'' \mapsto \tau' F_{p_f} v_1] \approx \tau_2\) that \(t_2\) will call the same outside function \(\tau'' \rightarrow^* \tau' F_{p_f}\) the same number of times with contextually equivalent arguments, as otherwise an attacker such as the one discussed in Ex-1 of Section 2.2 can be used to distinguish between the MiniML terms \(t_1\) and \(t_2\). Per Lemma 1 we have that contextually equivalent MiniML terms will be
marshalled into equal low-level values \( w \), we thus conclude that \( M_2 \) will produce the same label \( L \) as follows.

\[
M_2 \xrightarrow{\text{call } p_f(w)!} (S'_2 = \langle (N_2; \mu_2 \vdash \Sigma_2, E_2 : \tau' \rightarrow \tau), \mu_2 \rangle)
\]

Where \( M'_2 \vdash N'_2 M'_2 \) and allowing us to conclude that the thesis \( S'_2 \vdash S'_2 \) holds.
- For case (3) we have that: mutatis mutandis.

\[\square\]

**Lemma 4. (Reflection)** \( \{t_1\}^\dagger \approx^l \{t_2\}^\dagger \Rightarrow \{t_1\}^\dagger \approx^a \{t_2\}^\dagger \).

**Proof.** We prove the lemma by the contrapositive, the lemma is restated as:

\[
\{t_1\}^\dagger \not\approx^a \{t_2\}^\dagger \Rightarrow \{t_1\}^\dagger \not\approx^l \{t_2\}^\dagger
\]

The proof has two cases. In the first case the bisimulation fails immediately as the embedded \( t_1 \) and \( t_2 \) produce differently labelled transitions after silent reduction.

1. \( \langle \{t_1\} \rangle^\dagger \Rightarrow S'_1 \land \exists S'_2. \{t_2\} \Rightarrow \exists S'_2 \Rightarrow \langle \{t_1\} \rangle^\dagger \Rightarrow M'_1 \land \exists M'_2. \{t_2\} \Rightarrow \exists M'_2 \).

We proceed by case analysis over the label \( \gamma \). Only label applies \( \gamma = v! \) as \( t_1 \) and \( t_2 \) are two well typed MiniML terms that either diverge or reduce to a value \( v \) that is then marshalled out as MiniML\( ^a \) value \( v \).

- \( \gamma = v! \): We can expand the assumption \( \{t_1\}^\dagger \not\approx^a \{t_2\}^\dagger \) as:

\[
\star: \varepsilon \vdash t_1 : \tau \xrightarrow{\tau}^* \vdash N_1; \mu_1 \vdash \varepsilon \triangleright v_1 : \tau
\]

\[
N_1; \mu_1 \vdash \varepsilon \triangleright v_1 : \tau \xrightarrow{v_1!} N_1; \mu_1 \vdash \varepsilon
\]

were either \( \{t_2\}^\dagger \) diverges

\[
\star: \varepsilon \vdash t_2 : \tau^\dagger
\]

or \( \{t_2\}^\dagger \) reduces and marshalls to a value \( v_2 \), where \( v_1 \neq v_2 \).

\[
\star: \varepsilon \vdash t_2 : \tau \xrightarrow{\tau}^* \vdash N_2; \mu_2 \vdash \varepsilon \triangleright v_2 : \tau
\]

\[
N_2; \mu_2 \vdash \varepsilon \triangleright v_2 : \tau \xrightarrow{v_2!} N_2; \mu_2 \vdash \varepsilon
\]

The low-level bisimulation over \( \{t_1\}^\dagger \) will produce a label \( \text{ret } p_r(w_1) \) were \( w_1 \) is the word marshalled down for \( v_1 \) after the low-level attacker calls the \( p'_\text{start} \) entry point.

\[
\langle (\star; \emptyset \vdash \varepsilon \circ t_1 : \tau), \emptyset \rangle \xrightarrow{\text{call } p'_\text{start}(p_r)} \langle (\star; \emptyset \vdash \varepsilon \circ t_3 : \tau), p_r : \emptyset \rangle
\]

\[
\langle (\star; \emptyset \vdash \varepsilon \circ t_1 : \tau), p_r : \emptyset \rangle \xrightarrow{\tau}^* \langle (N_1; \mu_1 \vdash \varepsilon \triangleright w_1 : \tau), p_r : \emptyset \rangle
\]

\[
\langle (N_1; \mu_1 \vdash \varepsilon \triangleright w_1 : \tau), p_r : \emptyset \rangle \xrightarrow{\text{ret } p_r(w_1)} \langle (N_1; \mu_1 \vdash \varepsilon), \emptyset \rangle
\]

29
If \( \{t_2\}^+ \) diverges, so does \( \{t_2\}^\dagger \):

\[
\langle(\star): \emptyset \vdash e \circ t_2 : \tau), \emptyset \rangle \xrightarrow{\text{call } p_{\text{start}}(p_r)} \langle(\star): \emptyset \vdash e \circ t_2 : \tau), p_r : \emptyset \rangle
\]

\[
\langle(\star): \emptyset \vdash e \circ t_2 : \tau), p_r : \emptyset \rangle \uparrow
\]

If \( \{t_2\}^\dagger \) does not diverge, we have that it reduces to a word \( w_2 \) where \( w_1 \neq w_2 \). We thus conclude that \( \{t_1\}^\dagger \xrightarrow{\text{call } p_{\text{start}}(p_r) \text{ ret } p_r(w_1)} M'_1 \) and that \( \not\exists M'_2 \) such that:

\[
\{t_2\}^\dagger \xrightarrow{\text{call } p_{\text{start}}(p_r) \text{ ret } p_r(w_1)} M'_2.
\]

2. \( \{t_2\}^\dagger \Rightarrow S'_2 \land \not\exists S'_1 \Rightarrow \Rightarrow S'_1 \Rightarrow \Rightarrow \{t_2\}^\dagger \Rightarrow \Rightarrow M'_2 \land \not\exists M'_1 \Rightarrow \Rightarrow M'_1 \).

Similar to case (1).

In the second case there is a sequence of high-level context actions that result in two states where different LTS transitions apply. In this case we establish the thesis by showing that each MiniML\(^a\) attack can be replicated by an assembly language attacker. We proceed by case analysis over the actions of the MiniML\(^a\) attacker:

- \( \Rightarrow n'_1 \): by the LTS for \( \approx^a \) we have that:

\[
N; \mu \vdash \Sigma \xrightarrow{n'_1} N; \mu \vdash \Sigma, (t \cdot []) : \tau \rightarrow \tau' \quad \text{where } N(n'_1) = (t, \tau \rightarrow \tau')
\]

It follows from the semantics of MiniML\(^a\) that this transition is always followed by an input by the attacker.

\[
N; \mu \vdash \Sigma, (t \cdot []) : \tau \rightarrow \tau' \xrightarrow{\nu} \Sigma, (t \cdot []) : \tau \rightarrow \tau' < \nu : \tau
\]

The assembly language attacker replicates this action as follows:

\[
\langle(N; \mu \vdash \Sigma), e, p \rangle \xrightarrow{\text{call } p_{\text{start}}(w_n, w_r)} \langle(N; \mu \vdash \Sigma, (t \cdot []) : \tau \rightarrow \tau' < w : \tau), e, p_r : p \rangle \quad \text{where } N(w_i) = (t, \tau \rightarrow \tau')
\]

where \( \llbracket \nu \rrbracket = \{w_i\} \).

- \( !n'_1 \) by the LTS for \( \approx^a \) we have that:

\[
\mu \vdash \text{C} \bullet \text{E[deref } n'_1] \quad \mid \ N; \mu \vdash \Sigma \rightarrow \mu \vdash \text{C} \mid \ N; \mu \vdash \Sigma \circ ![i] : \tau \quad \text{where } N(n'_1) = (l_i, \text{Ref } \tau)
\]

The assembly language attacker replicates this action as follows:

\[
\langle(N; \mu \vdash \Sigma), e, p \rangle \xrightarrow{\text{call } p_{\text{start}}(w_n, p_r)} \langle(N; \mu \vdash \Sigma \circ ![i] : \tau), e, p_r : p \rangle \quad \text{where } N(w_i) = (l_i, \text{Ref } \tau)
\]

- \( \Rightarrow n'_1 \): similar to the \( !n'_1 \) case.
– $\gg ref^\tau$: by the LTS for $\approx^a$ we have that:

\[
N; \mu \models \Sigma \gg ref^\tau \rightarrow N; \mu \models \Sigma, (\text{ref } []) : \tau \rightarrow \text{Ref } \tau
\]

Again, it follows from the semantics of MiniML$^a$ that this transition is always followed by an input by the attacker.

\[
N; \mu \models \Sigma, (t [\cdot]) : \tau \rightarrow \tau' \xrightarrow{w^2} \Sigma, (\text{ref } []) : \tau \rightarrow \text{Ref } \tau \triangleleft v : \tau
\]

The assembly language attacker replicates this action as follows:

\[
\langle (N; \mu \models \Sigma), \overline{p} \rangle \xrightarrow{\text{call } p_{\text{ref}}^w(w_t, w, p_r)^2} \langle (N; \mu \models \Sigma), (\text{ref } []) : \tau \rightarrow \text{Ref } \tau \triangleleft w : \tau), p_r : \overline{p} \rangle \quad \text{where } \text{convt}(w_t) = \tau
\]

\[\square\]

### 6 Related Work

Our attacker model is based on the insights of Wand [19] on the nature of programming language reflection. Alternative attacker models are Jagadeesan et al.’s attacker language with low-level memory access operators [6] or erasure function approach of several non-interference works [10]. The former is only 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. [8] 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 [11] 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 [20]. These alternatives, however, do not provide separated program states or explicated marshalling rules both required to model the assembly language attacker.

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 [1]. 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 [7] and gordon’s bisimulation for FPC [5]. An alternative approach could be the environmental bisimulations of Sumii and Pierce [18], 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.

A different line of research focusses on developing security architectures with access control mechanisms comparable to PMA: Flicker [12], Fides [17] and the
Intel SGX [13]. The existence of industry prototypes alongside research ones underlines the feasibility of bringing efficient and secure low-level memory access control to commodity hardware. Besides the secure compilation works of Patrignani and Agten et al. [14] for Fides-like PMA architectures, no results comparable to ours have been proven for these systems.

7 Conclusions

This paper presented the implementation of a secure CESK machine for MiniML. The CESK machine is made secure by applying a low-level memory isolation mechanism (PMA) and by following a methodology that extends a previous developed secure FFI with a realistic low-level attacker and applies Biernacka et al.’s syntactic correspondence. A concatenation of formal properties for each step of the methodology ensures that the result is secure.

References

8. A. Larmuseau and D. Clarke. Formalizing a secure foreign function interface. In SEFM 2015, LNCS. Springer. To appear, online at: https://db.tt/y87tcQ0V.

32