Abstract machines are both theoretical models used to study language properties and practical models of language implementations. As with all language implementations, abstract machines are subject to security violations by the context in which they reside. This paper presents the implementation of an abstract machine for ML that preserves the abstractions of the language it implements, in possibly malicious, low-level contexts. To guarantee this security result, we make use of a low-level memory isolation mechanism and derive the formalisation of the machine through a methodology, whose every step is accompanied by formal properties that ensure that the step has been carried out properly. We provide an implementation of the abstract machine and analyse its performance for relevant scenarios.

1 Introduction

Abstract machines are both theoretical models used to study language properties and practical models of language implementations. Nowadays, several languages, especially functional ones, are implemented using abstract machines. For example, Scheme-based languages run on SCED and SKI machines, OCaml’s bytecode runs on the Zinc abstract machine and the Glasgow Haskell Compiler uses the Spineless Tagless G-machine [11] internally.

When security-sensitive applications are run by an abstract machine it is crucial that the abstract machine implementation does not leak security sensitive information. Outside of implementation mistakes, abstract machine implementations are also threatened by the low-level context in which they reside. In practice, an abstract machine implementation will use or interact with various, low-level libraries and/or components that may be written with malicious intent or susceptible to code injection attacks. Such malicious low-level components can bypass software based security mechanisms and may disclose confidential data, break integrity checks and so forth [23].

This paper presents the derivation and implementation of an abstract machine for MiniML, a light-weight version of ML featuring references and recursion, that runs on a processor enhanced with the Protected Module Architecture (PMA) [23]. 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. Our abstract machine executes MiniML programs within this protected memory without sacrificing the functionality of MiniML in its interact with the outside world.

To guarantee the security of the implemented abstract machine, we follow a two step methodology to derive the formalisation of the abstract machine. In the first step MiniML is extended with a secure foreign function interface (FFI). This extension to MiniML is derived by improving the theoretical secure operational semantics of Larmuseau et al. [12], with Patrignani et al.’s trace semantics of a realistic low-level attacker [21]. In the second step we apply the syntactic correspondences of Biernacka et al. [2] to this extension of MiniML, to obtain the formalisation of a CESK machine implementation for MiniML. For each of these syntactic correspondences we prove that they do not result in the abstract machine leaking security sensitive information.

After presenting the source language MiniML and the relevant security concepts and formalisations (Section 2), this paper makes the following contributions. It describes our methodology and how we apply it to derive a secure CESK machine for MiniML (Section 3). This paper also details our implementation of the CESK machine as well as the performance of the machine in certain test scenarios (Section 4).

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

2 Overview

This section first presents the source language MiniML (Section 2.1) and the formal method used to reason about its security (Section 2.2). Next it details the security threats to abstract machine implementations (Section 2.3). Lastly we cover the memory isolation mechanism that is used to protect the abstract machine implementation (Section 2.4) and the formal attacker model (Section 2.5).

2.1 The Source Language MiniML

The source language of our secure abstract machine implementation is MiniML: an extension of the typed λ-calculus featuring references and recursion. The syntax is as follows.

\[
\begin{align*}
t & ::= v \mid x \mid (t_1 \ t_2) \mid t_1 \op \ t_2 \mid \text{if} \ t_1 \ t_2 \ t_3 \mid \text{ref} \ t \mid t_1 := t_2 \mid t_1 ; t_2 \\
\text{op} & ::= + \mid - \mid \ast \mid \leq \mid \geq \mid \text{==} \\
v & ::= \text{unit} \mid \ell \mid \pi \mid (\lambda x : \tau. \ t) \mid \text{true} \mid \text{false} \\
\tau & ::= \text{Bool} \mid \text{Int} \mid \text{Unit} \mid \tau_1 \rightarrow \tau_2 \mid \text{Ref} \ \tau
\end{align*}
\]
\[
E ::= [] \mid E \cdot \mid v \cdot E \mid E \cdot \text{op} \cdot t \mid v \cdot \text{op} \cdot E \mid !E \mid \text{ref} \cdot E \mid \text{fix} \cdot E
\]
\[
\text{if} \ E \ t_1 \ t_2 \mid \text{let} \ x = \ E \ \text{in} \ t \mid E := t \mid v := E \mid E; \ t \mid \text{hash} \ E
\]

Here \( \pi \) indicates the syntactic term representing the number \( n \), \( \tau \) denotes the types and \( E \) is a Felleisen-and-Hieb-style evaluation context with a hole \([\cdot]\) that lifts the basic reduction steps to a standard left-to-right call-by-value semantics [6]. The \text{letrec} operator is syntactic sugar for a combination of \text{let} and \text{fix}. The operands \text{op} apply only to booleans and integers. The 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 \( \mu ::= \emptyset | \mu, l_i = v \), which is assumed to be an ideal store: it never runs out of space. Allocating new locations is done deterministically \( l_1, l_2, \ldots, l_n \). The term \text{hash} \( t \) maps a location to its index: \( l_i \mapsto i \), similar to how Java’s \text{.hashCode} method converts references to integers.

The reduction rules of MiniML are as expected.

\[
\begin{align*}
\mu | E[\text{if} \ t_2 \ t_3] & \rightarrow \mu | E[t_2] & \mu | E[\text{if} \ t_2 \ t_3] & \rightarrow \mu | E[t_3] \\
\mu | E[\text{let} x = v \ \text{in} \ t] & \rightarrow \mu | E[v/x] \\
\mu | E[\text{letrec} x : \tau = t_1 \ \text{in} \ t_2] & \rightarrow \mu | E[t'] \\
\mu | E[l_i] & \rightarrow \mu | E[i] \\
\mu' = [l_i \mapsto v] \mu \\
\mu | E[l := v] & \rightarrow \mu' | E[\text{unit}] \\
\mu | E[\pi_1 \ \text{op} \ \pi_2] & \rightarrow \mu | E[\text{rnd}(n_1 \ \text{op} \ n_2)] \\
\mu' = \mu, [l_i \mapsto v] \\
\mu | E[\text{ref} \ v] & \rightarrow \mu' | E[l_i] \\
\mu | E[l_i] & = v \\
\mu | E[l_i] & \rightarrow \mu | E[v] \\
\mu | E[l_i] & \rightarrow \mu | E[l_i] \\
\mu | E[\text{fix}(\lambda x : \tau. t)] & \rightarrow \mu | E[t[\text{fix}(\lambda x : \tau. t)/x]]
\end{align*}
\]

The typing rules of MiniML are entirely standard as well.

\[
\begin{align*}
x : \tau & \in \Gamma \\
\Gamma \vdash x : \tau \\
\Gamma \vdash b : \text{Bool} \\
\Gamma \vdash \bar{n} : \text{Int}
\end{align*}
\]
Definition 1. Contextual equivalence can distinguish them. Contextual equivalence is formalised as follows.

\[ \Gamma \vdash t \equiv \bowtie t \quad \text{for all } \tau, \Gamma, \text{ and contexts } \Gamma \]

where \( \bowtie \) denotes divergence, \( t_1 \) and \( t_2 \) are closed terms and neither the terms nor the contexts feature explicit locations \( l_i \) as they are not part of the static semantics. Note that contextually equivalent MiniML terms are of the same type \( \tau \) as a context \( C \) observes the same typing rules as the programs of the MiniML-calculus.

To formally state and reason about the security concerns of MiniML, contextual equivalence is used. A MiniML context \( C \) is a MiniML term with a single hole \( [\cdot] \), two MiniML terms are contextually equivalent if and only if no context \( C \) can distinguish them. Contextual equivalence is formalised as follows.

Definition 1. Contextual equivalence is defined as:

\[ t_1 \simeq t_2 \iff \forall C. C[t_1] \bowtie \iff C[t_2] \bowtie \]

where \( \bowtie \) denotes divergence, \( t_1 \) and \( t_2 \) are closed terms and neither the terms nor the contexts feature explicit locations \( l_i \) as they are not part of the static semantics. Note that contextually equivalent MiniML terms are of the same type \( \tau \) as a context \( C \) observes the same typing rules as the programs of the MiniML-calculus.

Two MiniML terms \( \mathcal{T} \) and \( \mathcal{T}' \) are, for example, not contextually equivalent as a context \( C = (\text{if } ([\cdot] = \mathcal{T}) \Omega \text{ true}) \), where \( \Omega \) is a diverging term, can distinguish them. MiniML’s \( \lambda \)-terms introduce many equivalences, there is no context \( C \), for example, that can distinguish the following two \( \lambda \)-terms.

\[ (\lambda x : \text{Int}. \overline{0}) \quad (\lambda x : \text{Int}. (x - x)) \quad (\text{Ex-1}) \]
The equivalences over the locations of MiniML are a little more complex. Due to the deterministic allocation order and the inclusion of the hash operation, a context can observe the number of locations as well as their indices. Locations when kept secret, however, can still produce equivalences as a context $C$ cannot contain a location $l_i$ unless it is shared at run-time. The following two terms, for example, are thus contextually equivalent.

$$(\text{ref } \text{false}; \overline{0}) \quad (\text{ref } \text{true}; \overline{0})$$

Contextual equivalence can be used to express security properties such as confidentiality and integrity. This is illustrated by the following two examples.

**Confidentiality** Consider the following two contextually equivalent MiniML terms.

$$\begin{align*}
\text{let } \text{secret} &= \text{ref } \text{false} \in \\
(\lambda x : \text{Int}. \text{secret}++; x)
\end{align*}$$

These two terms differ only in the value they store in the secret reference. Because these terms are contextually equivalent that implies that there is no MiniML context that can read the secret.

**Integrity** Consider the following two contextually equivalent MiniML terms.

$$\begin{align*}
(\lambda x : \text{Int} \rightarrow \text{Int}. \\
\text{let } y = \text{ref } \text{false} \in \\
\text{let } r = (x \overline{1}); \in \\
\text{if } (!y = \overline{0}) r (\overline{0})
\end{align*}$$

These two terms differ only in that the left term does an integrity check: it checks that the reference $y$ was not modified. Because these terms are contextually equivalent that implies that there is no way for a MiniML context to modify the reference $y$.

### 2.3 Security Properties of Abstract Machine Implementations

An abstract machine for MiniML is a program that inputs programs written in MiniML and then executes them according to the semantics for MiniML encoded in the machine. An abstract machine implementation has two security concerns: (i) implementation mistakes and (ii) malicious behaviour by the low-level context in which it resides. In this work we consider the latter concern.

More specifically, we consider the threats posed to an abstract machine implementation by an attacker with kernel-level code injection privileges. Kernel-level code injection is a critical vulnerability that bypasses all existing software-based security mechanisms: disclosing confidential data, disrupting applications and so forth. This attacker poses four threats to abstract machine implementations.


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

Abuse of references An abstract machine interoperates with the outside context, including possibly the attacker, by sharing references to data structures. An attacker that modifies these references can manipulate the internal state of the abstract machine and alter the control flow (as is the case for the Java VM [24]).

Observing implementation details Most abstract machines do not encode the exact semantics of MiniML, but instead use more concrete and more efficient encodings such as continuations and closures. Because these encodings make MiniML abstractions more concrete they may reveal to the attacker implementation details not observable to the MiniML contexts of Section 2.2.

Violation of type safety When interoperating with the outside context, the abstract machine not only shares data, it also receives it. If the abstract machine does not type check this incoming data, as commonly done for performance reasons, the attacker can violate the contextual equivalence of MiniML.

2.4 The Protected Module Architecture

To secure the run-time memory of our abstract machine implementation, addressing the inspection threat of Section 2.3, we make use of 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. 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.

<table>
<thead>
<tr>
<th>From \To</th>
<th>Protected</th>
<th>Unprotected</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Entry</td>
<td>Code</td>
</tr>
<tr>
<td>Protected</td>
<td>r x</td>
<td>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 our current implementation of the abstract machine makes use of a research prototype [23], Intel is developing a new instruction set, referred to as SGX, that enables the usage of PMA in commercial processors [17].
2.5 The Low-Level Attacker Model

The implemented abstract machine, that resides within the protected memory, remains secure in the face of the previously mentioned attacker with kernel level code injection capabilities. To formally reason about the capabilities and behaviour of this attacker, we make use of the fully abstract trace semantics of Patrignani and Clarke for assembly programs enhanced with PMA [21].

These trace semantics transition over a state $\Lambda$ which is a quintuple $(p, r, f, m, s)$, where $p$ is the program counter, $r$ the register file, $f$ a flags register, $m$ represents only the protected memory of PMA and $s$ is a descriptor that details where the protected memory partition starts, as well as the number of entry points and the size of the code and data sections. Additionally, the state $\Lambda$ can be $(\text{unknown, } m, s)$: a state modelling that the attacker is executing in unprotected memory. The trace semantics denote the observations and inputs of the attacker through the following labels $L$.

$$
L ::= \alpha | \tau
\alpha ::= \sqrt{} | \gamma! | \gamma?
\gamma ::= \text{call } p(r) | \text{ret } p(r)
$$

A label $L$ is either an observable action $\alpha$ or a non-observable action $\tau$. Decorations $? \text{ and } !$ indicate the direction of an observable action: from the unprotected memory to the protected memory (?) or vice-versa (!). Observable actions $\gamma$ are function calls or returns to a certain address $p$, combined with the registers $r$. Registers are in the labels as they contain the arguments to the function calls and also convey information on the behaviour of the code executing in the protected memory. Additionally, an observable action $\alpha$ can be a tick $\sqrt{}$ indicating termination.

Note that these labels do not denote low-level reads and writes to memory addresses. These are not required in this work. The memory of the protected memory module cannot be read or written by the low-level attacker, and our secure abstract machine is designed in a manner that removes the need for reads and writes to unprotected memory.

These trace semantics provide an accurate model of the attacker as they coincide with contextual equivalence for assembly programs enhanced with PMA. Formally the traces of an assembly program $P$, denoted as $\text{Tr}(P)$, are computed as follows: $\text{Tr}(P) = \{\sigma | \exists A. A_0(P) \xrightarrow{\pi} A \}$. Where $A_0$ is the initial state and the relation $A \xrightarrow{\pi} A'$ describes the traces generated by transitions between states. Whenever two assembly programs $P_1$ and $P_2$ are contextually equivalent they will produce the same sequence of traces (the same inputs and observations by attackers) as follows:

**Proposition 1 (Full Abstraction of the Traces [21]).**

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

where $\simeq_a$ denotes contextual equivalence between two assembly programs.
3 Deriving a Secure CESK Machine for MiniML

Our goal in this paper is to derive and implement a secure CESK machine for MiniML. A CESK machine is a transition system of four elements: the term being evaluated (Control), a map from variables to values (Environment), a map from locations to values (Store) and a stack of evaluation contexts (Kontinuations) [5].

Directly implementing a secure CESK for MiniML is, however, likely to fail. Instead, we derive a formalisation of the secure CESK machine for MiniML through a methodology of two distinct steps that start from the semantics of MiniML. This formalisation is then implemented using the PMA mechanism of Section 2.4, as detailed in Section 4.

To derive the CESK formalisation, we first extend MiniML with a secure foreign function interface (FFI) by adapting the theoretical formalization by Larmuseau et al. [12] (Section 3.1) to the low-level attacker of Section 2.5. This FFI enables MiniML programs to interoperate with the low-level attacker in a manner that prevents the low-level attacker from distinguishing between contextually equivalent MiniML terms. We reason over MiniML extended with this FFI by means of a labelled transition system, and formally prove that it preserves the equivalences of MiniML (Section 3.2).

In the second step of the methodology we derive a CESK formalisation from the previously obtained LTS by making use of syntactical correspondences that preserve the formal properties MiniML (Section 3.3).

Each step of the methodology includes a proposition that is proven to hold, establishing that the contextual equivalences of MiniML are preserved. We validate the methodology by showing that the combination of these propositions ensures that the derived CESK machine is secure (Section 3.4).

3.1 Step 1: Secure Foreign Function Interface

We extend MiniML with a secure FFI to the low-level attacker of Section 2.5 by adapting the theoretical FFI design of Larmuseau et al. [12]. The resulting extension of MiniML is formalised by means of a labelled transition system (LTS). Concretely this LTS is a triple \((S, L, \rightarrow)\), where the state \(S\) is a MiniML program extended with certain FFI specific attributes, \(L\) are the labels of low-level attacker’s fully abstract trace semantics (Section 2.5) and \(\rightarrow\) denotes the labelled transitions between the states is \(S \rightarrow S’\). The LTS does not include a state for the attacker as the labels capture all the relevant details of the low-level attacker.

The state \(S\) is a quadruple: \((\omega, \mu, N, p)\). The first element \(\omega\) is a substate that captures the four different execution states of MiniML throughout the FFI. The second element \(\mu\) is the store of locations (Section 2.1). The third element of the state \(S\) (\(N\)) is a map used to keep track of the reference objects that are used to mask the security relevant terms of MiniML. The final element of the state \(S\) is a stack of low-level pointers \(p\), that enable the MiniML to transfer control to the correct address of the attacker. In what follows we elaborate on these four
different execution states, the employed reference objects, the low-level usage of addresses, how input from and output to the attacker is handled and the transition rules. The formal security of this FFI is handled in Section 3.2.

The four FFI states of MiniML Throughout its interoperation with the low-level attacker MiniML programs adopt four different states. A MiniML program is either (1) executing a term \( t \) of type \( \tau \), (2) marshalling out values to be returned to the attacker, (3) marshalling in input from the attacker that is expected to be of type \( \tau \) or (4) waiting on input from the attacker.

\[
\begin{align*}
(1) \ 
\Sigma \circ t : \tau \\
(2) \ 
\Sigma \triangleright m : \tau \\
(3) \ 
\Sigma \triangleleft m : \tau \\
(4) \ 
\Sigma
\end{align*}
\]

Where \( m ::= v \ | \ w \) as the marshalling states (2) and (3) are responsible for converting MiniML values \( v \) to low-level words \( w \) and vice versa.

Each of these four different states include \( \Sigma \): a type annotated stack of evaluation contexts, which keeps track of the interoperation between the attacker and the MiniML program. This type annotated stack, is formally defined as:

\[
\Sigma ::= \varepsilon \ | \ \Sigma, E : \tau \rightarrow \tau'
\]

where, as explained in Section 2.1, \( E \) is an evaluation context that represent the computation that was halted when control was reverted to the attacker. This stack of evaluation contexts is type annotated, to ensure that no typing information is lost during the interactions with the attacker.

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 FFI and the derived CESK machine, to keep track of which locations or \( \lambda \)-terms and locations have been shared with the attacker. Larmuseau et al. [12] model reference objects for \( \lambda \)-terms and locations through names \( n_f \) and \( n_l \) respectively. Both names are tracked in a map \( N \) that records the associated term and type, as follows.

\[
N ::= \star \ | \ N, n_f^i \mapsto (t, \tau) \ | \ N, n_l^i \mapsto (t, \tau)
\]

A fresh name \( n_f^i \) is created deterministically every time a \( \lambda \)-term is shared between MiniML and the attacker. The index \( i \) of these names \( n_f^i \) denumerates the shared \( \lambda \)-terms. The first \( \lambda \)-term to be shared will be masked by a name \( n_f^1 \), the second \( \lambda \)-term will be masked by a name \( n_f^2 \) and so on. In contrast, the index \( i \) of the name \( n_l^i \) will correspond to the index of the location it refers to (\( n_l^i \mapsto l_i \)). Larmuseau et al. have previously proven that these names and their indices do not reveal to an attacker any information not available to MiniML contexts and thus do not violate the contextual equivalences of MiniML [12].
Entry points and return pointers The interoperation with the low-level attacker involves two kinds of addresses: entry points and return pointers. The entry points are, as detailed in Section 2.4, the interface that the attacker (residing in unprotected memory) uses to interact with a MiniML program and the CESK machine that executes it. To provide the low-level attacker the same level of functionality as MiniML context, we introduce an entry point \( p_e \) for each possible MiniML operation that the low-level attacker may need. Whenever the attacker wants to dereference a shared location, for example, it will use the entry point for dereferencing shared locations \( p_{e \text{deref}} \).

Whenever the attacker makes use of an entry point, control switches to the protected memory where the MiniML program and the CESK machine reside. To correctly revert control to the attacker, the FFI keeps track of the address from which the call to the entry point originates. On the flip side when control reverts from the attacker to the secure abstract machine, control cannot simply jump to the correct address within the protected memory as it is inaccessible from the outside (Section 2.4). Instead, the attacker must make use of a return-back entry point \( p_{e \text{retb}} \) that handles the reverting of control.

Attacker input and output Input from the attacker is handled in the marshalling in state of the FFI: \( \langle \Sigma : m : \tau, \mu, N, p \rangle \). Whenever the input does not observe the type \( \tau \), the MiniML program errors (\( \text{wr} \)) and subsequently terminates. Output to the attack is handled in the marshalling out state of the FFI: \( \langle \Sigma : m : \tau, \mu, N, p \rangle \). again whenever the typing check fails the MiniML program errors (\( \text{wr} \)) and subsequently terminates.

To save space in the following marshalling rules, we compress the marshalling out state \( \langle \Sigma : m : \tau, \mu, N, p \rangle \) into a wrapper \( \uparrow L \downarrow L \langle \Sigma : m : \tau, \mu, N, p \rangle \) into a wrapper \( \uparrow L \downarrow L \langle \Sigma : m : \tau, \mu, N, p \rangle \). 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 their byte word hexadecimal representation as \( \text{hex} \{ \cdot \} \).

\begin{align*}
\uparrow \langle \text{true} \rangle^N_{\text{Bool}} & \rightarrow \uparrow \langle 0x1 \rangle^N_{\text{Bool}} \\
\uparrow \langle m \rangle^N_{\text{Int}} & \rightarrow \uparrow \langle \text{hex}(n) \rangle^N_{\text{Int}} \\
\tau = \tau_1 \rightarrow \tau_2 & \rightarrow \uparrow \langle \text{F} p_f \rangle^N_{\tau} \\
j = |N| + 1 & \rightarrow \Rightarrow (N, n_j \mapsto (\lambda x : \tau.t), \tau \rightarrow \tau')) \\
\uparrow \langle \text{false} \rangle^N_{\text{Bool}} & \rightarrow \uparrow \langle 0x0 \rangle^N_{\text{Bool}} \\
\uparrow \langle \text{unit} \rangle^N_{\text{Unit}} & \rightarrow \uparrow \langle 0xDEAD \rangle^N_{\text{Unit}} \\
\uparrow \langle \text{false} \rangle^N_{\text{Bool}} & \rightarrow \uparrow \langle 0x0 \rangle^N_{\text{Bool}} \\
\uparrow \langle \text{true} \rangle^N_{\text{Bool}} & \rightarrow \uparrow \langle 0x1 \rangle^N_{\text{Bool}} \\
\uparrow \langle \text{unit} \rangle^N_{\text{Unit}} & \rightarrow \uparrow \langle 0xDEAD \rangle^N_{\text{Unit}} \\
\uparrow \langle \text{false} \rangle^N_{\text{Bool}} & \rightarrow \uparrow \langle 0x0 \rangle^N_{\text{Bool}} \\
\uparrow \langle \text{true} \rangle^N_{\text{Bool}} & \rightarrow \uparrow \langle 0x1 \rangle^N_{\text{Bool}} \\
\uparrow \langle \text{unit} \rangle^N_{\text{Unit}} & \rightarrow \uparrow \langle 0xDEAD \rangle^N_{\text{Unit}}
\end{align*}
Besides basic values, the attacker can also share its low-level functions with a MiniML program by sharing a function pointer \( p \). This function pointer is syntactically embedded into a MiniML program as a term: \( \tau \rightarrow \tau' Fp \), where the type \( \tau \rightarrow \tau' \) is included with the pointer to enable the FFI to type check the use of this attacker function at run-time.

The attacker cannot share its locations with the MiniML program. Reading and writing to unprotected memory creates many challenges to contextual equivalence [21] and is thus avoided in our design of the secure CESK machine. Instead, the attacker is given the ability to request the creation of a MiniML location through an entry point \( p^\text{set} \). The attacker is only able to write a value to such a shared MiniML location through the entry point \( p^\text{set} \), which ensures that this occurs in a type safe manner.

**Transitions** The transitions of this FFI LTS are as follows.

\[
\begin{align*}
&w \neq \text{Ox1 or } w \neq \text{Ox0} \\
&\frac{\langle \Sigma \circ t : \tau, \mu, N, \overline{p} \rangle}{\langle \Sigma \circ t' : \tau, \mu', N, \overline{p} \rangle} \quad \text{(S-Silent)} \\
&\langle \Sigma \circ v : \tau, \mu, \overline{n} \rangle \quad \text{(S-Setup)} \\
&\langle \Sigma \triangleleft v : \tau, \mu, \overline{n} \rangle \quad \text{(S-Plug)} \\
&\langle \Sigma \triangleright m : \tau, \mu, \overline{n} \rangle \quad \text{(S-MarshIN)} \\
&\langle \Sigma \triangleright m' : \tau, \mu, \overline{n'} \rangle \quad \text{(S-MarshOut)} \\
&\langle \Sigma \triangleleft \text{wr} : \tau, \mu, \overline{n} \rangle \quad \text{(Wr-I)} \\
&\langle \Sigma, \mu, \overline{n} \rangle \quad \text{(Wr-R)} \\
&\langle \Sigma, \mu, \overline{n} \rangle \quad \text{(Wr-C)}
\end{align*}
\]
\[\langle \varepsilon, \emptyset, \star, \emptyset \rangle \quad \text{(Done)}\]

\[\langle \Sigma \circ t : \tau, \emptyset, \star, \emptyset \rangle \xrightarrow{\text{call } p_{\text{start}}(p_r)} \langle \Sigma \circ t : \tau, \emptyset, \star, p_r : \emptyset \rangle \quad (A-\text{Start})\]

\[\langle \Sigma \triangleright w : \tau, \mu, N, p_r : \emptyset \rangle \xrightarrow{\text{return } p_r \ (w)} \langle \Sigma, \mu, N, \emptyset \rangle \quad (M-\text{Ret})\]

\[\langle \Sigma, \mu, N, \emptyset \rangle \xrightarrow{\text{call } p_{\text{strm}}(w, p_r)} \langle (\Sigma \circ \null \mu : \tau, \mu, N, p_r : \emptyset) \quad (A-\text{Deref})\]

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

\[\langle \Sigma, \mu, N, \emptyset \rangle \xrightarrow{\text{call } p_{\text{strm}}(w_n, w, p_r)} \langle ((\Sigma, \langle \cdot [\ ] \rangle \triangleright w : \tau), \mu, N, p_r : \emptyset) \quad (A-\text{Apply})\]

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

\[\langle \Sigma, \mu, N, \emptyset \rangle \xrightarrow{\text{call } p_{\text{strm}}(w_n, w, p_r)} \langle ((\Sigma, \langle l_i := [\ ] \rangle) \triangleright \text{Ref } \tau \triangleright w : \tau), \mu, N, p_r : \emptyset) \quad (A-\text{Set})\]

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

\[\langle \Sigma, \mu, N, \emptyset \rangle \xrightarrow{\text{call } p_{\text{strm}}(w_n, w, p_r)} \langle ((\Sigma, \langle \text{ref } [\ ] \rangle) \triangleright \text{Ref } \tau \triangleright w : \tau), \mu, N, p_r : \emptyset) \quad (A-\text{Ref})\]

Transitions within the MiniML program, such as for example S-Silent, are not observable the attacker and are thus again labelled as silent (S-\(*\)). Whenever the assembly-language attacker makes a mistake by either providing words that cannot be marshalled (Wr-I) or by calling an inaccessible address or returning to an address that is not the \(p_{\text{strm}}^w\) (Wr-C,Wr-R), the protected memory is terminated to the empty state \(\langle \star ; \emptyset \triangleright \varepsilon \rangle, e, \emptyset \rangle\).

To start the computation of the MiniML program, the low-level attacker calls the entry point \(p_{\text{start}}^w\) passing as its only argument \(p_r\), the address at which it expects the result returned (A-Start). Once the MiniML program has computed a value and marshalled it to a byte word representation \(w\) control will revert to that address \(p_r\) (M-Ret). The low-level attacker, as mentioned earlier, cannot jump to an address of the protected memory outside of the entry points, and must thus return the values it computes to the return entry point \(p_{\text{strm}}^w\) (A-Ref).

The low-level attacker calls a separate entry point \(p^e\) for each type of operation on MiniML terms. These entry points take as an argument a byte word representation of the names that model reference objects \(w_n\) as well as byte words that represents the arguments. The operations listed above, handle function calls (A-Apply), the setting and dereferencing of shared location (A-Set,A-Deref) as well as the allocation of new references (A-Ref). In the case of the latter rule we make use of a conversion function \(\text{convt}\) to convert a hexadecimal value into a type \(\tau\). Note that during each of these operations the expected type is
always inferred at run-time, from the type information stored in \( \Sigma \) and \( \Pi \), to ensure that the operations happen in a type safe manner.

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).

### 3.2 Full abstraction of the FFI

To obtain a secure CESK machine for MiniML, we must formally prove that this FFI preserves the contextual equivalences of MiniML. To do so we first define a notion of \textit{weak} bisimulation over the LTS. In contrast to strong bisimulations, such a bisimulation does not take into account the silent transitions of the LTS, only the actions \( \alpha \). Define the transition relation \( S \xrightarrow{\alpha} S' \) as \( S \xrightarrow{\tau} \ast \xrightarrow{\alpha} S' \) where \( \xrightarrow{\tau} \ast \) is the reflexive transitive closure of the silent transitions \( \xrightarrow{\tau} \).

**Definition 2.** \( \mathcal{B} \) is a \textbf{bisimulation} iff \( S_1 \mathcal{B} S_2 \) implies:

1. If \( S_1 \xrightarrow{\alpha} S'_1 \), \( \exists S'_2 \) s.t. \( S_2 \xrightarrow{\alpha} S'_2 \) and \( S'_1 \mathcal{B} S'_2 \).
2. If \( S_2 \xrightarrow{\alpha} S'_2 \), \( \exists S'_1 \) s.t. \( S_1 \xrightarrow{\alpha} S'_1 \) and \( S'_1 \mathcal{B} S'_2 \).

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

To reason about the security properties of the FFI we must also define a compilation scheme that places a MiniML term \( t \) into a state \( S \) of the FFI.

<table>
<thead>
<tr>
<th>( t )</th>
<th>( \tau )</th>
<th>( \mathcal{F} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \epsilon \circ t )</td>
<td>( \tau )</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>

where \( \tau \) it the type of \( t \). We now state that this secure FFI preserves the contextual equivalences of MiniML as follows.

**Theorem 1 (Full Abstraction).**

\[
\llbracket t_1 \rrbracket_{\mathcal{F}} \approx \llbracket t_2 \rrbracket_{\mathcal{F}} \iff t_1 \simeq t_2
\]

The proof proceeds by relating the bisimilarity \( \approx \) to a \textit{congruent} bisimilarity over MiniML, a bisimilarity that coincides with the contextual equivalences of MiniML. This proof is basically a combination of our previous proofs for related results [14,13]. In what follows we first introduce the bisimulation over MiniML and then proceed to formally proof the theorem.

**A Congruent Bisimulation for MiniML** We define a bisimulation relation \( \mathcal{B} \) over the programs of MiniML that is congruent: it coincides with the contextual equivalence relation \( \simeq \). There have been multiple different bisimulations and trace semantics over typed \( \lambda \)-calculi with references. In this paper we use the applicative bisimulation of Larmuseau \textit{et al.} [12]. Applicative bisimulation is defined through an LTS. The LTS models the interactions between a MiniML context \( C \) and a MiniML program. The LTS is formally defined as a triple \( (\zeta, \mathcal{L}, \xrightarrow{\mathcal{L}}) \). The state \( \zeta = K; \mu \mid t \) is the MiniML run-time state extended with a sequence K
that records the locations \( l_i \) that the opponent has knowledge of. This is needed to capture fact that locations are not part of the static semantics and thus do not appear in contexts unless made available at run-time. The labels \( L' \) of the LTS are defined as follows.

\[
L' ::= \gamma \mid \tau \quad \gamma ::= @v \mid .i \mid l_i := v \mid \text{ref } v \mid !l_i \mid l_i \mid b \mid \pi \mid \text{unit}
\]

The most relevant labelled reductions are as follows.

\[
\begin{align*}
K;\mu \mid t & \xrightarrow{\tau} K;\mu' \mid t' \quad \text{(Sil)} & K;\mu \mid \pi & \xrightarrow{\pi} K;\mu' \mid \pi \quad \text{(O-N)} \\
K;\mu \mid (\lambda x : \tau.t) & \xrightarrow{\lambda v} K;\mu \mid ((\lambda x : \tau.t) v) \quad \text{where } v : \tau \quad \text{(l-App)} \\
K;\mu \mid v \xrightarrow{\text{ref } v'} & K;\mu \mid l_i := v' \quad \text{where } l \in K \text{ and } \vdash l : \tau \text{ and } \vdash v : \tau \quad \text{(l-S)} \\
K;\mu \mid v \xrightarrow{!l} & K;\mu \mid !l \quad \text{if } l_i \in K \quad \text{(l-D)} & K;\mu \mid l_i & \xrightarrow{l_i} K, l_i;\mu \mid v_i \quad \text{(O-L)} \\
K;\mu \mid v & \xrightarrow{\text{ref } v'} K;\mu \mid \text{ref } v' \quad \text{(l-Ref)} & K;\mu \mid \langle v_{i\in 1..n} \rangle & \xrightarrow{i} K;\mu \mid v_i \quad \text{(l-Proj)} \\
K;\mu \mid \text{unit} & \xrightarrow{\text{unit}} K;\mu \mid \text{unit} \quad \text{(O-U)} \\
K;\mu \mid \text{true} & \xrightarrow{\text{true}} K;\mu \mid \text{true} \quad \text{(O-T)} \\
K;\mu \mid \text{false} & \xrightarrow{\text{false}} K;\mu \mid \text{false} \quad \text{(O-F)}
\end{align*}
\]

Reduction steps between terms cannot be observed by a context and are thus labelled as silent through the label \( \tau \) (Sil). Whenever a MiniML program reduces to a value that is not a \( \lambda \)-term or tuple (as it may contain a \( \lambda \)-term), the context can observe that value \((O-N,O-L,O-T,O-F,O-U)\). Observing a label \((O-L)\) is a special case as it adds a new location \( l_i \) to \( K \) : the list of observed locations. A context interacts with a \( \lambda \)-term by applying it to values \((l-App)\), likewise a context queries members of a tuple instead of observing it directly \((l-Proj)\). A context can also dereference observed locations \( l_i \) \((l-D)\), create new ones \((l-Ref)\) and assign them values \((l-S)\).

We again, define a weak bisimulation over this LTS. Define the transition relation \( \zeta \xrightarrow{\ast} \zeta' \) as \( \zeta \xrightarrow{\tau} \ast \zeta \rightarrow \zeta' \) where \( \tau \rightarrow \ast \) is the reflexive transitive closure of the silent transitions \( \tau \rightarrow \). A Bisimulation over this LTS is now formally defined as follows.

**Definition 3.** The relation \( B^i \) is a **bisimulation** iff \( \zeta_1 B^i \zeta_2 \) implies:

1. Given \( \zeta_1 \xrightarrow{\ast} \zeta_1' \) there is a \( \zeta_2' \) such that: \( \zeta_2 \xrightarrow{\ast} \zeta_2' \) and \( \zeta_1' B^i \zeta_2' \)
2. Given \( \zeta_2 \xrightarrow{\ast} \zeta_2' \) there is a \( \zeta_1' \) such that: \( \zeta_1 \xrightarrow{\ast} \zeta_1' \) and \( \zeta_1' B^i \zeta_2' \)

We denote bisimilarity, the largest bisimulation, as \( \approx^i \). This bisimilarity coincides with contextual equivalence as follows.

**Proposition 2 (Congruence of the Bisimilarity [12]).** \( t_1 \approx t_2 \iff t_1 \approx^i t_2 \)

**Proof** First we restate the thesis of Theorem 1 as:

\[
t_1 \approx^i t_2 \iff \llbracket t_1 \rrbracket^{FFI} \approx \llbracket t_2 \rrbracket^{FFI}
\]

The proof splits this thesis into two sublemmas: preservation and reflection.
Lemma 1. (Preservation) $t_1 \approx^i t_2 \Rightarrow [t_1]^{FFI} \approx [t_2]^{FFI}$.

Proof. We must develop a relation $\mathcal{R}$ such that:

$$[t_1]^{FFI} \mathcal{R} [t_2]^{FFI}$$

and that for all $S_1 \mathcal{R} S_2$ we have that:

$$S_1 \overset{\sim}{\Rightarrow} S'_1 \text{ and } \exists S'_2, S_2 \overset{\sim}{\Rightarrow} S'_2 \Rightarrow S'_1 \mathcal{R} S'_2$$

(2)

$$S_2 \overset{\sim}{\Rightarrow} S'_2 \text{ and } \exists S'_1, S_1 \overset{\sim}{\Rightarrow} S'_1 \Rightarrow S'_1 \mathcal{R} S'_2$$

(3)

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

$$\mathcal{R}_0 = \{(\langle \Sigma_1, \mu_1, N_1, p_1 \rangle, \langle \Sigma_2, \mu_2, N_2, p_2 \rangle) \mid \forall N_1, N_2, \Sigma_1, \Sigma_2, \mu_1, \mu_2, p_1, p_2 \text{ such that } N_1 \approx_N N_2 \text{ and } \text{Dom}(\mu_1) = \text{Dom}(\mu_2) \text{ and } \Sigma_1 \approx_{\Sigma} \Sigma_2 \text{ and } p_1 = p_2\}$$

$$\mathcal{R}_1 = \{(\langle \tau_1, \mu_1, N_1, p_1 \rangle, \langle \tau_2, \mu_2, N_2, p_2 \rangle) \mid \forall N_1, N_2, \Sigma_1, \Sigma_2, \mu_1, \mu_2, t_1, t_2, \tau_1, \tau_2, p_1, p_2 \text{ such that } (\langle \Sigma_1, \mu_1, N_1, p_1 \rangle, \langle \Sigma_2, \mu_2, N_2, p_2 \rangle) \in \mathcal{R}_0\text{ and } t_1 \approx t_2 \text{ and } \tau_1 = \tau_2\}$$

$$\mathcal{R}_2 = \{(\langle \Sigma_1, \mu_1, N_1, p_1 \rangle, \langle \Sigma_2, \mu_2, N_2, p_2 \rangle) \mid \forall N_1, N_2, \Sigma_1, \Sigma_2, \mu_1, \mu_2, m_1, m_2, t_1, t_2, \tau_1, \tau_2, p_1, p_2 \text{ such that } (\langle \Sigma_1, \mu_1, N_1, p_1 \rangle, \langle \Sigma_2, \mu_2, N_2, p_2 \rangle) \in \mathcal{R}_0\text{ and } m_1 \approx_m m_2 \text{ and } \tau_1 = \tau_2\}$$

$$\mathcal{R}_3 = \{(\langle \Sigma_1, \mu_1, N_1, p_1 \rangle, \langle \Sigma_2, \mu_2, N_2, p_2 \rangle) \mid \forall N_1, N_2, \Sigma_1, \Sigma_2, \mu_1, \mu_2, m_1, m_2, t_1, t_2, \tau_1, \tau_2, p_1, p_2 \text{ such that } (\Sigma_1, \mu_1 \vdash \Sigma_2, \mu_2 \vdash \Sigma_2) \in \mathcal{R}_0\text{ and } m_1 \approx_m m_2 \text{ and } \tau_1 = \tau_2\}$$

where $\approx_N$ is defined as:

$$N_1 \approx_N N_2 \iff \text{dom}(N_1) = \text{dom}(N_2) \text{ and } \forall n \in \text{dom}(N_1), N_1(n) \approx N_2(n)$$

where $\approx_{\Sigma}$ is defined as:

$$\Sigma_1 \approx_{\Sigma} \Sigma_2 \iff$$

For $(E^1_1, \ldots, E^k_1)$ : $(\tau^1_1, \ldots, \tau^n_1)$ in $\Sigma_1$ and

$(E^1_2, \ldots, E^k_2)$ : $(\tau^1_2, \ldots, \tau^n_2)$ in $\Sigma_2$

we have $n' = n$ and for every $1 \leq i \leq n : \tau^1_i = \tau^2_i$ and

$\forall t', t'', \Gamma \vdash t' : \tau^1_1 \wedge \Gamma \vdash t'' : \tau^2_1 \wedge t' \approx t'' \Rightarrow E^1_1[t'] = E^2_2[t'']$
and where $\approx_m$ is defined as:

\[
\begin{align*}
m_1 \approx_m m_2 & \iff \\
(m_1 = t_1 \land m_2 = t_2 \land t_1 \simeq t_2) & \\
\lor (m_1 = w_1 \land m_2 = w_2 \land w_1 = w_2) & \\
\lor (m_1 = \text{wr} \land m_2 = \text{wr}) & \\
\lor (m_1 = \langle m_i \in 1..n \rangle \land m_2 = \langle m_i' \in 1..n \rangle \land \forall i \in 1..n. m_i \approx_m m_i')
\end{align*}
\]

We now proof the three cases.

– In case (1) we have that $\langle (\star; \emptyset \vdash \varepsilon; \emptyset \circ t_1), e_1, \emptyset \rangle \overset{\mathcal{R}}{\rightarrow} \langle (\star; \emptyset \vdash \varepsilon \circ t_2), e_2, \emptyset \rangle$ as we have that $t_1 \simeq t_2$ from the assumption [12].

– In case (2) we assume $S_1 \overset{\mathcal{R}}{\rightarrow} S_2$ and that $S_1 \Rightarrow S_1$. We proceed by case analysis on $L$.

• $L = \sqrt{
}$: By the LTS we have three sub-cases:

1. Transition Wr-I:

\[
(S_1 =)\langle \Sigma_1 \overset{\text{wr}}{\leftarrow} \tau_1, \mu_1, N_1, p_1 \rangle \rightarrow\langle \varepsilon, \emptyset, (\star, \emptyset) \rangle\]

It follows from $S_1 \overset{\mathcal{R}}{\rightarrow} S_2$ more specifically $S_1 \overset{\mathcal{R}_3}{\rightarrow} S_2$ that:

\[
S_2 = \langle \Sigma_2 \overset{\text{wr}}{\leftarrow} \tau_2, \mu_2, N_2, p_2 \rangle
\]

Where $N_1 \approx_N N_2$, $\text{Dom}(\mu_1) = \text{Dom}(\mu_2)$, $\Sigma_1 \approx_\Sigma \Sigma_2$ and $p_1 = p_2$ and that $\tau_1 = \tau_2$. By the LTS we have that only rule Wr-I applies as follows:

\[
S_2 \rightarrow\langle \varepsilon, \emptyset, (\star, \emptyset) \rangle
\]

Given that $\star \approx_N \star$, $\text{Dom}(\emptyset) = \text{Dom}(\emptyset)$ and $\varepsilon \approx_\varepsilon \varepsilon$, we conclude that $S_1' \overset{\mathcal{R}_0}{\rightarrow} S_2'$.

2. Transition Done:

\[
(S_1 =)\langle \varepsilon, \emptyset, (\star, \emptyset) \rangle \rightarrow\langle \varepsilon, \emptyset, (\star, \emptyset) \rangle
\]

It follows from $S_1 \overset{\mathcal{R}}{\rightarrow} S_2$ more specifically $S_1 \overset{\mathcal{R}_0}{\rightarrow} S_2$ that:

\[
S_2 = \langle \varepsilon, \emptyset, (\star, \emptyset) \rangle
\]

By the LTS we have that only rule the Done applies as follows:

\[
S_2 \rightarrow\langle \varepsilon, \emptyset, (\star, \emptyset) \rangle
\]

Given that $\star \approx_N \star$, $\text{Dom}(\emptyset) = \text{Dom}(\emptyset)$ and $\varepsilon \approx_\varepsilon \varepsilon$ and $\emptyset = \emptyset$, we conclude that $S_1' \overset{\mathcal{R}_0}{\rightarrow} S_2'$.
• $L = \text{ret } p_{\text{retb}}(w)$?: By the LTS we have two sub-cases:

1. Transition A-Ret:

\[
(S_1 = \langle \Sigma_1, E_1 : \tau_1 \rightarrow \tau'_1, \mu_1, N_1, \overline{p_1} \rangle) \xrightarrow{\text{ret } p_{\text{retb}}(w)} (S'_1 = \langle \Sigma_1, E_1 : \tau_1 \rightarrow \tau'_1 < w : \tau_1), \mu_1, N_1, \overline{p_1} \rangle)
\]

It follows from $S_1 \mathcal{R} S_2$ more specifically $S_1 \mathcal{R}_0 S_2$ that:

\[
S_2 = \langle \Sigma_2, E_2 : \tau_2 \rightarrow \tau'_2, \mu_2, N_2, \overline{p_2} \rangle
\]

Where $N_1 \approx_N N_2$, $\text{Dom}(\mu_1) = \text{Dom}(\mu_2)$, $\Sigma_1, E_1 : \tau_1 \rightarrow \tau'_1 \approx_E \Sigma_2, E_2 : \tau_2 \rightarrow \tau'_2$ and thus that $\tau_1 = \tau_2$ and $\tau'_1 = \tau'_2$ and $\overline{p_1} = \overline{p_2}$.

By the LTS we have that many rules including A-Ret apply and thus that $S_2$ can produce the same label:

\[
S_2 \xrightarrow{\text{ret } p_{\text{retb}}(w)} (S'_2 = \langle \Sigma_2, E_2 : \tau_2 \rightarrow \tau'_2 < w : \tau_2), \mu_2, N_2, \overline{p_2} \rangle)
\]

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

2. Transition Wr-R:

\[
(S_1 = \langle \Sigma_1, E_1 : \tau_1 \rightarrow \tau'_1, \mu_1, N_1, \overline{p_1} \rangle) \xrightarrow{\text{ret } p(w)} (S'_1 = \langle \varepsilon, \emptyset, \star, \emptyset \rangle)\text{ where } p \neq p_{\text{retb}}
\]

It follows from $S_1 \mathcal{R} S_2$ more specifically $S_1 \mathcal{R}_0 S_2$ that:

\[
S_2 = \langle \Sigma_2, E_2 : \tau_2 \rightarrow \tau'_2, \mu_2, N_2, \overline{p_2} \rangle
\]

This state does not enforce any restrictions on the abilities of the attacker to call the wrong address as such we conclude that Wr-R applies to $S_2$ as well.

\[
S_2 \xrightarrow{\text{ret } p(w)} (S'_2 = \langle \varepsilon, \emptyset, \star, \emptyset \rangle)\text{ where } p \neq p_{\text{retb}}
\]

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

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

\[
(S_1 = \langle \Sigma_1 \triangleright w_1 : \tau_1), \mu_1, N_1, \overline{p_1} \rangle) \xrightarrow{\text{ret } p_{\text{ret}}(w_1)} (S'_1 = \langle \Sigma_1, \mu_1, N_1, \overline{p_1} \rangle)
\]

It follows from $S_1 \mathcal{R} S_2$ more specifically $S_1 \mathcal{R}_2 S_2$ that:

\[
S_2 = \langle \Sigma_2 \triangleright w_2 : \tau_2), \mu_2, N_2, \overline{p_2} \rangle
\]
Where $N_1 \approx_N N_2$, $Dom(\mu_1) = Dom(\mu_2)$, $\Sigma_1 \approx \Sigma_2$ and $\overline{p}_1 = \overline{p}_2$ and $p_r = p_r$ and that $\tau_1 = \tau_2$. By the LTS we have that only $M$-Ret applies and that it will produce the same label:

$$S_2 \xrightarrow{\text{ref } p_{\text{ref}}(w_1)} (S'_2 =) (\Sigma_2, \mu_2, N_2, \overline{p}_2)$$

Given that $N_1 \approx_N N_2$, $Dom(\mu_1) = Dom(\mu_2)$, $\Sigma_1 \approx \Sigma_2$, and $\overline{p}_1 = \overline{p}_2$, we conclude that $S'_1 \mathcal{R}_0 S'_2$.

- $L = \text{call } p(\overline{w})$: By the LTS we have 6 sub-cases:
  1. Transition $A$-Start:
     $$\begin{align*}
     (S_1 =) & (\Sigma_1 \circ t_1 : \tau_1, \mu_1, N_1, \emptyset) \xrightarrow{\text{call } p_{\text{start}}(p_r)?} \\
     & (S'_1 =) (\Sigma_1 \circ t_1 : \tau_1, \mu_1, N_1, p_r : \emptyset)
     \end{align*}$$
     It follows from $S_1 \mathcal{R} S_2$ more specifically $S_1 \mathcal{R}_1 S_2$ that:
     $$S_2 = (\Sigma_2, \mu_2, N_2, \emptyset)$$
     Where $N_1 \approx_N N_2$, $Dom(\mu_1) = Dom(\mu_2)$, $\Sigma_1 \approx \Sigma_2$, $t_1 \approx t_2$ and $\tau_1 = \tau_2$. By the LTS we have that transition $A$-Start applies to $S_2$ given that $\overline{p}_2 = \emptyset$.
     $$S_2 \xrightarrow{\text{call } p_{\text{start}}(p_r)?} (S'_2 =) (\Sigma_2, \mu_2, N_2, p_r : \emptyset)$$
     Given that $(p_r : \emptyset) = (p_r : \emptyset)$, we conclude that $S'_1 \mathcal{R}_1 S'_2$.
  2. Transition $A$-Ref:
     $$\begin{align*}
     (S_1 =) & (\Sigma_1, \mu_1, N_1, \overline{p}_1) \xrightarrow{\text{call } p_{\text{ref}}(w_r,w,p_r)?} \\
     & (S'_1 =) (\Sigma_1, (\text{ref } [[]]) : \tau \rightarrow \text{Ref } \tau < w : \tau, \mu_1, N_1, p_r : \overline{p}_1)
     \end{align*}$$
     where $\text{conv}(w_1) = \tau_1$
     It follows from $S_1 \mathcal{R} S_2$ more specifically $S_1 \mathcal{R}_0 S_2$ that:
     $$(S_2 =) (\Sigma_2, \mu_2, N_2, \overline{p}_2)$$
     Where $N_1 \approx_N N_2$, $Dom(\mu_1) = Dom(\mu_2)$, $\Sigma_1 \approx \Sigma_2$ and $\overline{p}_1 = \overline{p}_2$. By the LTS we have that several transitions apply including $A$-Ref:
     $$S_2 \xrightarrow{\text{call } p_{\text{ref}}(w_r,w,p_r)?} (S'_2 =) (\Sigma_2, (\text{ref } [[]]) : \tau \rightarrow \text{Ref } \tau < w : \tau, \mu_2, N_2, p_r : \overline{p}_2)$$
     where $\text{conv}(w_1) = \tau$
     It now follows from the fact that $\Sigma_1, (\text{ref } [[]]) : \tau \rightarrow \text{Ref } \tau \approx_{\Sigma} \Sigma_2, (\text{ref } [[]]) : \tau \rightarrow \text{Ref } \tau$ and that $w = w$ that $S'_1 \mathcal{R}_2 S'_2$.  

18
3. Transition A-Set:

\[ (S_1 =) \langle \Sigma_1, e_1, \mu_1, N_1, \overline{p_1} \rangle \xrightarrow{\text{call } p_{\text{set}}^\epsilon(w_n, w, p_r)} (S'_1 =) \langle (\Sigma, (l_i := [\cdot]) : \tau_1 \rightarrow \text{Unit} \triangleleft w : \tau_1), \mu_1, N_1, p_r : \overline{p_1} \rangle \]

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

It follows from \( S_1 \mathcal{R} S_2 \) more specifically \( S_1 \mathcal{R}_N S_2 \) that:

\[ S_2 = \langle (\Sigma_2, E_2 : \tau_2 \rightarrow \tau'_2), \mu_2, N_2, \overline{p_2} \rangle \]

Where \( N_1 \approx_N N_2, \ Dom(\mu_1) = Dom(\mu_2), \Sigma_1, E_1, \tau_1 \rightarrow \tau'_1 \approx_N \Sigma_2, E_2, \tau_2 \rightarrow \tau'_2 \) and thus that \( \tau_1 = \tau_2 \) and \( \tau'_1 = \tau'_2 \) and \( \overline{p_1} = \overline{p_2} \).

From the LTS it follows that the same transition A-Set is possible.

\[ (S_2 =) \xrightarrow{\text{call } p_{\text{set}}^\epsilon(w_n, w, p_r)} (S'_2 =) \langle (\Sigma_2, (l_j := [\cdot]) : \tau_2 \rightarrow \text{Unit} \triangleleft w : \tau_2), \mu_2, N_2, p_r : \overline{p_2} \rangle \]

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 \( S'_1 \mathcal{R}_N S'_2 \).

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

5. Transition A-Apply:

\[ (S_1 =) \langle \Sigma_1, \mu_1, N_1, \overline{p_1} \rangle \xrightarrow{\text{call } p_{\text{set}}^\epsilon(w_n, w, p_r)} (S'_1 = \langle (\Sigma_1, (t_i [\cdot]) : \tau_1 \rightarrow \tau'_1 \triangleleft w : \tau_1), \mu_1, N_1, p_r : \overline{p_1} \rangle \]

where \( N(w_n) = (t_i, \tau \rightarrow \tau') \)

It follows from \( S_1 \mathcal{R} S_2 \) more specifically \( S_1 \mathcal{R}_N S_2 \) that:

\[ S_2 = \langle (\Sigma_2, E_2 : \tau_2 \rightarrow \tau'_2), \mu_2, N_2, \overline{p_2} \rangle \]

Where \( N_1 \approx_N N_2, \ Dom(\mu_1) = Dom(\mu_2), \Sigma_1, E_1, \tau_1 \rightarrow \tau'_1 \approx_N \Sigma_2, E_2, \tau_2 \rightarrow \tau'_2 \) and thus that \( \tau_1 = \tau_2 \) and \( \tau'_1 = \tau'_2 \) and \( \overline{p_1} = \overline{p_2} \).

From the LTS it follows that the same transition A-Apply is possible.

\[ (S_2 =) \xrightarrow{\text{call } p_{\text{set}}^\epsilon(w_n, w, p_r)} (S'_2 =) \langle (\Sigma_2, (t_j [\cdot]) : \tau_2 \rightarrow \tau'_2 \triangleleft w : \tau_2), \mu_2, N_2, p_r : \overline{p_2} \rangle \]

where \( N(w_n) = (t_j, \tau_2 \rightarrow \tau'_2) \)

By the fact that \( N_1 \approx_N N_2 \), it follows that \( t_1 \simeq t_2 \) and thus that \( \tau_2 = \tau_1 \) and \( \tau'_1 = \tau'_2 \). We conclude that \( S'_1 \mathcal{R}_N S'_2 \).
6. Transition Wr-C: similar to the Wr-R case.

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

\[
\begin{align*}
(S_1) &= \langle \Sigma_1 \circ E_1[(\tau'' \rightarrow \tau' \ Fp_f \ v_1) : \tau], \mu_1, N_1, P_1 \rangle \\
(S'_1) &= \langle \Sigma_1, E_1 : \tau' \rightarrow \tau_1 \triangleright v_1 : \tau'', \mu_1, N_1, P_1 \rangle \\
\text{(M-Call)}
\end{align*}
\]

\[
S_1 \xrightarrow{\text{call} \ p_f(w)!!} (S'_1) \xrightarrow{\gamma} S''_1
\]

It follows from \( S_1 \mathcal{R} \mathcal{S}_2 \) more specifically \( S_1 \mathcal{R}_1 \mathcal{S}_2 \) that:

\[
(S_2) = \langle (\Sigma_2 \circ t_2 : \tau_2), \mu_2, N_2, P_2 \rangle
\]

Where \( N_1 \approx_N N_2 \) \( \text{Dom}(\mu_1) = \text{Dom}(\mu_2) \), \( E_1 \approx_E E_2 \), \( P_1 = P_2 \), \( \tau_1 = \tau_2 \) and \( t_2 \approx E_1[(\tau'' \rightarrow \tau \ Fp_f \ v_1)] \). It follows from the fact that the bisimulation starts from a state \( \text{compgtFFI} \), where \( t \) is a pure MiniML term, that \( \tau'' \rightarrow \tau \ Fp_f \) derives from the attacker. It now follows from the fact that \( E_1[(\tau'' \rightarrow \tau \ Fp_f \ v_1)] \approx t_2 \) that \( t_2 \) will call the same outside function \( \tau'' \rightarrow \tau \ Fp_f \) the same number of times with contextually equivalent arguments, as otherwise a MiniML context attacker can use a combination of reference counting and value comparison 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 \( S_2 \) will produce the same label \( L \) as follows:

\[
S_2 \xrightarrow{\text{call} \ p_f(w)!!} (S'_2) = \langle (\Sigma_2, E_2 : \tau' \rightarrow \tau), \mu_2, N_2, P_2 \rangle
\]

Where \( S'_2 \mathcal{R} \mathcal{S}_2 \) and allowing us to conclude that the thesis \( S'_1 \mathcal{R} S'_2 \) holds.

- For case (3) we have that: mutatis mutandis.

\[
\square
\]

**Lemma 2. (Reflection)** \( \llbracket t_1 \rrbracket^{\text{FFI}} \approx \llbracket t_2 \rrbracket^{\text{FFI}} \Rightarrow t_1 \approx^i t_2. \)

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

\[
t_1 \neq^i t_2 \Rightarrow \llbracket t_1 \rrbracket^{\text{FFI}} \neq \llbracket t_2 \rrbracket^{\text{FFI}}
\]

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

1. \( \emptyset : \emptyset \mid t_1 \xrightarrow{\text{call}} \xi'_1 \wedge \not \exists \xi'_2 : \emptyset \mid t_2 \xrightarrow{\gamma} \xi'_2. \)

We proceed by case analysis over the label \( \gamma \):

20
- **true**: We have that: \(0; 0 \mid t_1 \rightarrow^* 0; \mu_1 \mid \text{true} \xrightarrow{\text{true}} 0; \mu_1 \mid t'_1\). By the assumption, \(t_1 \neq t_2\) and the LTS we have that either \(t_2\) diverges \(t_2 \uparrow\) or \(t_2\) reduces to a different value: \(0; 0 \mid t_2 \rightarrow^* 0; \mu_2 \mid v_2\) where \(v_2 \neq \text{true}\). The low-level FFI over \([t_1]^{\text{FFI}}\) will produce a label \(\text{ret } p_r(w_1)\) where \(w_1\) is the word marshalled down for \(v_1\) after the low-level attacker calls the \(p_{\text{start}}^r\) entry point.

\[
\begin{align*}
\langle (\varepsilon \circ t_1 : \tau), 0, 0, 0 \rangle \xrightarrow{\text{call } p_{\text{start}}^r(p_r)} &\quad \langle (\varepsilon \circ t_1 : \tau), 0, \star, p_r : 0 \rangle \\
\langle (\varepsilon \circ t_1 : \tau), 0, \star, p_r : 0 \rangle \xrightarrow{\tau^*} &\quad (\varepsilon \triangleright w_1 : \tau), \mu_1, \mu_1, p_r : 0 \rangle \\
(\varepsilon \triangleright w_1 : \tau), \mu_1, \mu_1, p_r : 0 \rangle \xrightarrow{\text{ret } p_r(w_1)} &\quad (\varepsilon, \mu_1, 0, 0) \\
\end{align*}
\]

If \(t_2\) diverges, so does \([t_2]^{\text{FFI}}\):

\[
\begin{align*}
\langle (\varepsilon \circ t_2 : \tau), 0, 0, 0 \rangle \xrightarrow{\text{call } p_{\text{start}}^r(p_r)} &\quad \langle (\varepsilon \circ t_2 : \tau), 0, \star, p_r : 0 \rangle \\
\langle (\varepsilon \circ t_2 : \tau), 0, \star, p_r : 0 \rangle \xrightarrow{\tau^*} &\quad (\varepsilon \triangleright w_2 : \tau), \mu_1, \mu_1, p_r : 0 \rangle \\
\end{align*}
\]

If \(t_2\) does not diverge, we have that it reduces to a word \(w_2\) where \(w_1 \neq w_2\). We thus conclude that \([t_1]^{\text{FFI}} \xrightarrow{\text{call } p_{\text{start}}^r(p_r) \text{ ret } p_r(w_1)} S_1^r\) and that \(\beta S_2^r\) such that: \([t_2]^{\text{FFI}} \xrightarrow{\text{call } p_{\text{start}}^r(p_r) \text{ ret } p_r(w_1)} S_2^r\).

- **false**: Analogous to the **true** case.
- **unit**: Analogous to the **true** case.
- **π**: Analogous to the **true** case.
- **l**: Analogous to the **true** case.

- **@v**: We have that: \(0; 0 \mid t_1 \rightarrow^* 0; \mu_1 \mid (\lambda x : \tau.t_1') v\). By the assumption, \(t_1 \neq t_2\) and the LTS we have that: \(0; 0 \mid t_2 \rightarrow^* 0; \mu_2 \mid v_2\) where \(v_2 \neq (\lambda x : \tau'.t_2')\) or \(v_2 = (\lambda x : \tau'.t_2')\) such that: \(\Gamma \vdash (\lambda x : \tau.t_1') : \tau_1 \land \Gamma \vdash (\lambda x : \tau'.t_2') : \tau_2 \land \tau_1 \neq \tau_2\). The divergence case was considered in the **true** case. By the transition rules of the FFI we have that:

\[
\begin{align*}
[t_1]^{\text{FFI}} \xrightarrow{\text{call } p_{\text{start}}^r(p_r)} &\quad \rightarrow^* (\varepsilon \triangleright (\lambda x : \tau.t_1') : \tau_1, \mu_1, \star, p_r : 0) \\
\end{align*}
\]

and that

\[
\begin{align*}
[t_2]^{\text{FFI}} \xrightarrow{\text{call } p_{\text{start}}^r(p_r)} &\quad \rightarrow^* (\varepsilon \triangleright (\lambda x : \tau.t_2') : \tau_2, \mu_2, \star, p_r : 0) \\
\end{align*}
\]

Where \(\tau_1 \neq \tau_2\). In this case both \([t_1]^{\text{FFI}}\) and \([t_2]^{\text{FFI}}\) will first return the same name \(\eta^2\). However the differing type information is stored in \(\eta\) and thus the outside input for \([t_1]^{\text{FFI}}\) will not be replicable for \([t_2]^{\text{FFI}}\).

- **ref v**: Given that the that the store \(\mu\) never runs out of space, we have that the label applies to any two terms \(t_1\) and \(t_2\), contradicting the assumption.
– $l_i$: By the rules of the LTS we have that $\emptyset; \emptyset | t_1 \nmid \zeta'_1$ as $l_i$ is only applicable if $K_1 \neq \emptyset$. This case thus contradicts the assumption.

– $l_i := v$: Similar as the $l_i$ case.

2. $\emptyset; \emptyset | t_2 \nmid \zeta'_2 \land \beta \zeta'_1$, $\emptyset; \emptyset | t_1 \nmid \zeta'_1$: Similar to case (1).

In the second case there is a sequence of MiniML context actions that result in two states where different LTS transitions apply. In this case we establish the thesis by showing that each MiniML context action can be replicated by in the FFI. We proceed by case analysis over these actions of the MiniML context:

– $\gamma = \oplus v$: By the LTS over $\approx^j$ we have that:

$$K; \mu | (\lambda x : \tau.t) \xrightarrow{\oplus v} K; \mu | ((\lambda x : \tau.t) v) \text{ where } \vdash v : \tau$$

The LTS of the FFI can replicate this action as follows:

$$\begin{align*}
\langle \Sigma, \mu, N, p \rangle &\xrightarrow{\text{call } p_{\text{get}}(w_i, w, p_r) ?} \langle \Sigma, \mu, N, p_r : p, \overline{\tau} \rangle \\
\langle \Sigma, \mu, N, p \rangle &\xrightarrow{\text{call } p_{\text{get}}(w_i, w, p_r) ?} \langle \Sigma, \mu, N, p_r : p, \overline{\tau} \rangle \\
\langle \Sigma, \mu, N, p \rangle &\xrightarrow{\text{call } p_{\text{get}}(w_i, w, p_r) ?} \langle \Sigma, \mu, N, p_r : p, \overline{\tau} \rangle
\end{align*}$$

where $N(w_i) = (t, \tau \rightarrow \tau')$ and $\|v\| = \hat{\langle u \rangle}$.  

– $\gamma = \lnot l_i$: By the LTS over $\approx$ we have that:

$$K; \mu | v \xrightarrow{\lnot l_i} K; \mu | !l \text{ where } l_i \in K$$

Every time a location is shared in MiniML the LTS over $\approx$ adds it to the context knowledge base $K$. Similarly every shared location in the FFI is added to $N$. It follows from the fact that the FFI preserves the semantics of MiniML that every location shared by the MiniML term will be shared and stored to $N$ for $\llbracket \ell \rrbracket_{\text{FFI}}$. The LTS over the FFI can thus replicate this action as follows:

$$\begin{align*}
\langle \Sigma, \mu, N, p \rangle &\xrightarrow{\text{call } p_{\text{get}}(w_i, w, p_r) ?} \langle \Sigma, \mu, N, p_r : p, \overline{\tau} \rangle \\
\langle \Sigma, \mu, N, p \rangle &\xrightarrow{\text{call } p_{\text{get}}(w_i, w, p_r) ?} \langle \Sigma, \mu, N, p_r : p, \overline{\tau} \rangle \\
\langle \Sigma, \mu, N, p \rangle &\xrightarrow{\text{call } p_{\text{get}}(w_i, w, p_r) ?} \langle \Sigma, \mu, N, p_r : p, \overline{\tau} \rangle
\end{align*}$$

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

– $\gamma = l_i := v$: Similar to the $l_i$ case.

\[\Box\]
3.3 Step 2: Deriving the CESK Machine

We now derive a secure CESK machine, from the LTS representation of the FFI, by applying an adapted version of Biernacka et al.’s syntactic correspondence between context-sensitive calculi and abstract machines [2]. This correspondence consists of four transformations that modify state and reduction rules in a way that preserves the contextual equivalences of MiniML. Throughout this section the result of each transformation is annotated with a superscript that indicates to which transformation it belongs.

For each transformation $T$ we prove that the contextual equivalences of MiniML are preserved in two steps. We first develop a bisimilarity $\approx^T$ (Definition 2) over the modified LTS. In a second step we use that bisimilarity to prove that there exist a compilation scheme: $J \cdot K^T$ that compiles the current state $S$ into the derived machine state $S^T$, in a manner that preserves the contextual equivalences of MiniML. This is proven by relating the newly derived bisimilarity $\approx^T$ to the previously derived bisimilarity $\approx$ over the FFI.

**Proposition 3 (Preservation).**

$$S_1 \approx S_2 \iff \langle S_1 \rangle^T \approx^T \langle S_2 \rangle^T$$

Each step of the syntactic correspondence was implemented in Ocaml\(^3\) and tested using MiniML programs. In what follows, we provide a short overview of the transformations as well as proof sketches for Proposition 3. A full formalisation of each of the derived LTSs and as well as the result CESK machine can be found in the accompanying tech report.

1. **Context Sensitive Reduction** In this step explicit continuation contexts are derived to separate reduction contexts from the term being executed. To that end, the LTS obtained in Section 3.1 is transformed into a new LTS $K$ defined as a triple $(S^K, L, -^L \rightarrow^K)$. The new state $S^K$ is a quintuple $\langle c, \mu, k, N, p \rangle$ where $c$ is a control and $k$ the context. A control $c$ is either a MiniML term $t$, a low-level word $w$ (when marshalling) or a stop state.

   $$c ::= t \mid w \mid sp.$$  

   This stop state $sp$ is needed to indicate that the LTS is halted, waiting on input from the attacker. The contexts $k$ of the new machine state are derived by first transforming MiniML’s evaluation contexts $E$ (Section 2.1) into explicit continuations contexts $K$, as follows.

   $$K ::= \langle \cdot \rangle \mid K[[\cdot \mid c] \mid K[v \langle \cdot \rangle] \mid \ldots$$

   Next the annotated evaluation stack $\Sigma$ is converted into an outer continuation that captures the 4 different states of MiniML within the FFI (Section 3.1).

   $$k ::= \langle \cdot \rangle \mid k[K : \tau \rightarrow \tau] \mid k[\circ K : \tau] \mid k[\triangleleft K : \tau] \mid k[\triangleright K : \tau]$$

\(^3\) https://github.com/sylvarant/secure-abstract-machine
The new transition rules $\overset{L \cdot K}{\rightarrow}$ differ from $\overset{L}{\rightarrow}$ in that they include explicit transitions to plug and construct the continuation contexts $K$. The following rule M-R1, for example, enforces the left to right evaluation order of MiniML by selecting the left term $c_1$ as the control and converting the to be evaluated right term $c_2$ into a new continuation.

$$\langle (c_1 \ c_2), K[\ o \ K : \tau], \mu, N, \bar{p} \rangle \overset{\tau}{\rightarrow} \langle c_1, K[\ o \ K[\mid c_2 : \tau], \mu, N, \bar{p} \rangle \quad \text{(M-R1)}$$

**Proof Sketch of Proposition 3** Given a bisimilarity $\approx^K$ over LTS$^K$, the following compilation scheme $\llbracket \cdot \rrbracket^K$:

$$\llbracket \langle \Sigma \circ \tau : \mu, N, \bar{p} \rangle \rrbracket^K = \langle t, \llbracket K \circ \mid : \tau \rangle, \mu, N, \bar{p} \rangle$$

$$\llbracket \langle \Sigma \triangleleft m : \tau, \mu, N, \bar{p} \rangle \rrbracket^K = \langle m, \llbracket K \triangleright \mid : \tau \rangle, \mu, N, \bar{p} \rangle$$

$$\llbracket \langle \Sigma, \mu, N, \bar{p} \rangle \rrbracket^K = \langle \text{sp}, \llbracket K \triangleright \mid : \tau \rangle, \mu, N, \bar{p} \rangle$$

$$\llbracket \Sigma, E : \tau, \bar{p} \rrbracket = \llbracket E[K : \tau] \mid \llbracket E \rrbracket = \mid \rangle \quad \text{where } K \text{ explicates } E$$

compiles states $S$ that are bisimilar in $\approx$ into states $S^K$ that are bisimilar in $\approx^K$ as the new transitions of LTS$^K$ are silent transitions that are ignored by our weak bisimulations.

**Machine Formalisation** The full formalisation is listed below. For each reduction context of MiniML, there is now a continuation $K$. The rule (M-Plug) plugs a value into a continuation and is thus called at least once for every type of communication. The marshalling rules are not repeated within the machine formalisation to save a bit of space.

$$S^K \in \text{State}^K = \text{Ctrl} \times \text{Storage} \times \text{Kont} \times \text{Map} \times \text{Returns}$$

$$\begin{array}{lll}
c \in \text{Ctrl} & ::= & x \ | \ (c \ c) \ | \ \text{if} \ c_1 \ c_2 \ c_3 \ | \ \text{let} \ x = c_1 \ \text{in} \ c_2 \ | \ v \\
c_1; c_2 & | & c_1 \ \text{op} \ c_2 \ | & c_1 := c_2 \ | & \text{hash} \ c \ | & \text{fix} \ c \ | & \text{ref} \ c \\
& & \text{letrec} \ x : \tau = c_1 \ \text{in} \ c_2 ; \ | & \text{sp} \ | \ w \\
v \in \text{Value} & ::= & \tau \rightarrow \tau' \ F \ p \ | & (\lambda x : \tau . c) \ | & b \ | & \text{unit} \ | & l_i \ | & \pi \ | \text{true} \ | \text{false} \\
x \in \text{Var} & ::= & + \ | \ - \ | \ * \ | \ < \ | \ > \ | \ == \\
\tau \in \text{Type} & ::= & \text{Bool} \ | \ \text{Int} \ | \ \text{Unit} \ | \ \tau_1 \rightarrow \tau_2 \ | \ \text{Ref} \ \tau \\
k \in \text{Kont} & ::= & \mid \ | \ K[\ K : \tau \rightarrow \tau'] \ | \ K[\ o \ K : \tau] \ | \ K[\ \mid \mid \ K : \tau] \ | \ K[\ \mid \ K] \\
K \in \text{Kin} & ::= & \mid \ | \ K[\mid c] \ | \ K[\mid \mid \mid \mid] \ | \ K[\mid \mid \mid \mid \mid ; c] \\
& & K[\mid \mid \mid \mid \mid ; c] \ | \ K[\mid \mid \mid \mid \mid ; c] \ | \ K[\mid \mid \mid ; c] \\
& & K[\mid \mid ; c] \ | \ K[\mid \mid \mid \mid ; c] \ | \ K[\mid \mid \mid ; c] \\
& & K[\mid \mid \mid ; c] \ | \ K[\mid \mid \mid ; c] \\
\text{n}_i \in \text{Name} & ::= & \text{an enumerable set.} \\
N \in \text{Map} & ::= & \text{Name} \rightarrow (\text{Ctrl} \times \text{Type})
\end{array}$$
is a closure. This requires rules that propagate the map of substitutions
through the sub-terms of the term
for example, in the following transition rule (A-Apply) where
\[ \langle w, \mu, k \circ k : \tau \rangle, N, p_r : p \rangle \]
A-Set
\[ \langle w, \mu, k \circ k : \tau \rangle, N, p_r : p \rangle \]
A-Ref
\[ \langle \tau \rightarrow \tau \cdot \text{Ref} \rangle, w, \mu, k : \tau, N, p_r : p \rangle \]
M-Call
\[ \langle \tau \cdot \text{Ref} \rangle, w, \mu, k : \tau, N, p_r : p \rangle \]
Using closures as controls of the transition system also requires us to update
the reduction rules that use substitution. Function application, for example, is
updated to make use of the map of substitutions by splitting it into two new rules
M-B and M-V. These rules respectively add a new substitution to the map
when performing an application and fetch a substitution from that same map
when reducing a standalone variable.
\[ \langle \lambda x : \tau \cdot c[x], \mu, k, N, p \rangle \]
M-B
\[ \langle c, \mu, k, N, p \rangle \]
M-V
2. Closure Conversion In this second transformation the \( \lambda \)-terms of MiniML are
converted into closures. As a result, the previously derived LTS\( ^K \) is transformed
into a new LTS\( ^C \) \( (S^C, L, \rightarrow^C) \). The new state \( S^C \) is a quintuple \( \langle c_1, \mu, k, N, p \rangle \)
where the control terms are now closures \( c_1 \) as in the \( \lambda \)-calculus [2]:
\[ c_1 ::= c | c_1 \cdot c_2 | \text{if } c_1 \cdot c_2 \cdot c_3 | ... \]
where \( c \) is a map of substitutions: \( c ::= \star | c \cdot (c_1 / x) \).
Note that in this closure calculus a \( \lambda \)-term is simply a term, but its closure
is a value. Note also that the explicit continuations \( k \) now also use closures \( c_1 \)
instead of controls \( c \).

Proof Sketch of Proposition 3. Using explicit substitutions does not interfere with the contextual equivalences of MiniML, as this internal implementation detail is never observed by the low-level attacker. Given a bisimilarity \( \approx^C \) over LTS\(^C\), the compilation scheme \([\cdot]^C\):

\[
\left[ \langle c, \mu, k, N, \emptyset \rangle \right]^C = \langle c[\emptyset], \mu, k, N, \emptyset \rangle
\]

compiles states \( S^K \) that are bisimilar in \( \approx^K \) into states \( S^C \) that are bisimilar in \( \approx^C \) as the additional transitions of LTS\(^C\) are silent transitions \( \tau \rightarrow \) that are ignored by the weak bisimulations.

Machine Formalisation The full formalisation is listed below. The Closure conversion introduces many propagation rules creating many additional rules.

\[
S^C \in \text{State}^C = \text{Closure} \times \text{Storage} \times \text{Kont} \times \text{Map} \times \text{Returns}
\]

\[
c \in \text{Ctrl} ::= x \mid (c \circ c) \mid \text{if } c_1 \circ c_2 \circ c_3 \mid \text{let } x = c_1 \text{ in } c_2 \mid v \mid (\lambda x : \tau.c)
\]

\[
c_1; c_2 \mid \text{op } c_2 \mid c_1 := c_2 \mid \text{hash } c \mid \text{fix } c \mid \text{ref } c
\]

\[
\text{letrec } x : \tau = c_1 \text{ in } c_2 ; \text{ sp } \mid w
\]

\[
c_1 \in \text{Closure} ::= c[e] (c \circ cl) \mid \text{if } c_1 \circ c_2 \circ c_3 \mid \text{let } x = c_1 \text{ in } c_2
\]

\[
c_1; c_2 \mid \text{cl } c_1 \circ c_2 \mid \text{cl } c_1 := c_2 \mid \text{hash } c \mid \text{fix } c
\]

\[
\text{letrec } x : \tau = c_1 \text{ in } c_2 \mid \text{ref } c
\]

\[
v \in \text{Value} ::= \tau \rightarrow \tau \cdot Fp \mid (\lambda x : \tau.c)[e] \mid b \mid \text{unit } l_i \mid \pi
\]

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

\[
x \in \text{Var} = \text{a set of identifiers.}
\]

\[
op \in \text{Oper} ::= + \mid - \mid \ast \mid < \mid > \mid \text{==}
\]

\[
\tau \in \text{Type} ::= \text{Bool } \mid \text{Int } \mid \text{Unit } \mid \tau_1 \rightarrow \tau_2 \mid \text{Ref } \tau
\]

\[
e \in \text{Env} \quad = \quad \ast \mid e \cdot (c \circ x)
\]

\[
k \in \text{Kont} ::= [\cdot ] \mid k[K : \tau \rightarrow \tau'] \mid k [\circ K : \tau] \mid k [\text{op } K : \tau]
\]

\[
K \in \text{Kin} ::= [\cdot ] \mid k[[\cdot ] \circ cl] \mid k[v[\cdot ]] \mid k[\text{if } \cdot ] c_1 \circ c_3 \mid k[\text{let } \cdot = c_2]
\]

\[
K[[\cdot ]] \mid k[l_i := \cdot ] \mid k[\text{op } c_1 \circ c_2]
\]

\[
K[\text{let } x = \cdot ] c_2 \mid k[\text{fix } c]\]

\[
n_i \in \text{Name} = \text{an enumerable set.}
\]

\[
N \in \text{Map} = \text{Name } \rightarrow (\text{Ctrl } \times \text{Type})
\]

\[
\langle v[e], k [\circ K[K'] : \tau], N \rangle \xrightarrow{\text{M-Plug}} \langle K'[v[e], k [\circ K : \tau], N] \rangle
\]

\[
\langle (c_1 \circ c_2)[e], k [\circ K : \tau], N \rangle \xrightarrow{\text{Prop-App}} \langle (c_1[e], c_2[e], k [\circ K : \tau], N) \rangle
\]

\[
\langle (c_1 \circ c_2), k [\circ K : \tau], N \rangle \xrightarrow{\text{M-R1}} \langle c_1[k] \circ K[\cdot ] c_2 : \tau], N \rangle
\]

\[
\langle (v_1[e], c_2[k [\circ K[v_1[e]] : \tau], N] \rangle \xrightarrow{\text{M-R2}}
\]

27
\[(\lambda x : \tau \cdot c)[e] \downarrow v' \] \(\Rightarrow c[e \cdot (v'[e]/x)], k, N) \tag{M-B} \]

\[x[e], \mu, k, N, p \downarrow \langle c_1, \mu, k, N, p \rangle \tag{M-V} \]

\[\text{if } c_1 \text{ c}_{2 \cdot 2} \text{ c}_{2} \text{ c}_{3}[e] \circ K : \tau, N \downarrow \Rightarrow \text{c}_{1} (k \circ K : \tau), N \] \tag{Prop-If} \]

\[\text{if } \text{true}[\text{c}_{2 \cdot c_{2}}] \text{ c}_{2} \text{ c}_{3}, k, N \downarrow \Rightarrow \text{c}_{2}, k, N \tag{M-IFT} \]

\[\text{if } \text{false}[\text{c}_{2 \cdot c_{2}}] \text{ c}_{2} \text{ c}_{3}, k, N \downarrow \Rightarrow \text{c}_{3}, k, N \tag{M-IFF} \]

\[\text{let } x = c_{1} \text{ in } c_{2}, k[ \circ K : \tau], N \downarrow \Rightarrow \text{c}_{1}, k \circ K : \tau, N \] \tag{Prop-Let} \]

\[\text{let } x = \text{c}_{1} \text{ in } c_{2}, k \circ K : \tau, N \downarrow \Rightarrow \text{c}_{1} \circ K (\text{let } x = [\text{c}_{2}] : \tau), N \] \tag{M-R4} \]

\[\text{let } x = v[e] \text{ in } c_{1}, k[ \circ K : \tau], N \downarrow \Rightarrow \text{c}_{1}[e \cdot (v[e]/x)], k, N \] \tag{M-Let} \]

\[\text{ref } c[e] \circ K : \tau, N, p \downarrow \Rightarrow \langle \text{ref } c[e], \mu, k \circ K : \tau, N, p \rangle \tag{Prop-Ref} \]

\[\text{ref } c[e], k \circ K : \tau, N, p \downarrow \Rightarrow \langle c_{1}, c_{2} \circ \text{ref } c[e], \mu, k \circ K : \tau, N, p \rangle \tag{M-Ra} \]

\[\text{ref } v[e], k \circ K : \tau, N, p \downarrow \Rightarrow \langle 1, \mu, l_{1} \mapsto v[e], k \circ K : \tau, N, p \rangle \tag{M-Ref} \]

\[\text{let } c[e], k \circ K : \tau, N, p \downarrow \Rightarrow \langle \text{let } c[e], k \circ K : \tau, N, p \rangle \tag{Prop-Der} \]

\[\text{let } c[e], k \circ K : \tau, N, p \downarrow \Rightarrow \langle \text{let } c[e], k \circ K : \tau, N, p \rangle \tag{Prop-Seq} \]

\[\text{let } c[e], k \circ K : \tau, N, p \downarrow \Rightarrow \langle \text{let } c[e], k \circ K : \tau, N, p \rangle \tag{M-RS2} \]

\[\text{let } c[e], k \circ K : \tau, N, p \downarrow \Rightarrow \langle \text{let } c[e], k \circ K : \tau, N, p \rangle \tag{M-Set} \]

\[\text{let } c[e], k \circ K : \tau, N, p \downarrow \Rightarrow \langle \text{let } c[e], k \circ K : \tau, N, p \rangle \tag{M-Fix} \]

\[\text{let } c[e], k \circ K : \tau, N, p \downarrow \Rightarrow \langle \text{let } c[e], k \circ K : \tau, N, p \rangle \tag{M-Fix} \]

\[\text{let } c[e], k \circ K : \tau, N, p \downarrow \Rightarrow \langle \text{let } c[e], k \circ K : \tau, N, p \rangle \tag{M-Fix} \]

\[\text{let } c[e], k \circ K : \tau, N, p \downarrow \Rightarrow \langle \text{let } c[e], k \circ K : \tau, N, p \rangle \tag{M-Fix} \]

\[\text{let } c[e], k \circ K : \tau, N, p \downarrow \Rightarrow \langle \text{let } c[e], k \circ K : \tau, N, p \rangle \tag{M-Fix} \]

\[\text{let } c[e], k \circ K : \tau, N, p \downarrow \Rightarrow \langle \text{let } c[e], k \circ K : \tau, N, p \rangle \tag{M-Fix} \]
rules are removed.

3. Refocusing and Transition Compression The previous two transformations introduced various additional reduction steps into the semantics of the original LTS. In this third step, the previously derived LTS\(^C\) is transformed into a new LTS\(^R\) (S\(^C\),\(\mathcal{L},\mathcal{R}\)), where the new rules \(\mathcal{L}\rightarrow\mathcal{R}\) are a refocused and compressed version of \(\mathcal{L}\rightarrow\mathcal{C}\) by following the methodology of Biernacka [2]. For example, M-B introduced in the previous transformation is optimized into the following:

\[
\begin{align*}
\langle v[e], \mu, k \circ K(\lambda x : \tau \cdot c)[e'] [\cdot] : \tau \rangle, N, p_r : \mathcal{P} & \xrightarrow{\tau} \langle c[e' \cdot (v[e] / x)], \mu, k \circ K : \tau \rangle, N, p_r \\
\end{align*}
\]

whereas the previous rule required the value to be plugged into the context before a reduction step could take place, this rule applies directly the value of \(k\).

Proof Sketch of Proposition 3. Given a bisimilarity \(\approx\) over LTS\(^R\), the identity relation compiles states \(S\(^C\)\) that are bisimilar in \(\approx\) into states \(S\(^R\)\) that are bisimilar in \(\approx\) as only the silent transitions of LTS\(^R\) differ from those of LTS\(^C\), the states remain unchanged in this transformation.

Machine Formalisation The full formalisation is listed below. The propagation and plug rules are removed, replaced by smarter reduction rules that transition immediately on the kontinuations. This also ensures that the propogation rules are removed.

\[
\begin{align*}
\langle w[e], \mu, k \triangleright : \tau \rangle, N, p_r : \mathcal{P} & \xrightarrow{\text{ret } p_x, w!} \langle \text{sp}[\emptyset], \mu, k \circ K \rangle \quad \text{(M-Marshout)} \\
\langle v[\emptyset], \mu, k \leftarrow : \tau \rangle, N, p_r : \mathcal{P} & \xrightarrow{\text{call } p_x, t\text{est}(w, p_r)} \langle \mathcal{L}[\emptyset], \mu, k \circ \cdot : \tau \rangle, N, p_r : \mathcal{P} \quad \text{(M-Marsin)} \\
\langle cl, \mu, k \circ K : \tau \rangle, N, p_r : \mathcal{P} & \xrightarrow{\tau} \langle cl, \mu, k \circ K : \tau \rangle, N, p_r \quad \text{(M-Wait)} \\
\langle \text{sp}[\emptyset], \mu, k, N, p_r \rangle & \xrightarrow{\text{ret } p_x, w!} \langle w[\emptyset], \mu, k \leftarrow : \tau \rangle, N, p_r \quad \text{(A-Retb)} \\
\langle \text{sp}[\emptyset], \mu, k, N, p_r \rangle & \xrightarrow{\text{call } p_x, \text{appl}(w, p_r)} \langle \mathcal{L}[\emptyset], \mu, k \circ \cdot : \tau \rangle, N, p_r : \mathcal{P} \quad \text{(A-Deref)} \\
\langle w[\emptyset], \mu, k \circ \cdot : \tau \rangle, N, p_r : \mathcal{P} & \xrightarrow{\text{call } p_x, \text{appl}(w, p_r)} \langle \mathcal{L}[\emptyset], \mu, k \circ \cdot : \tau \rangle, N, p_r : \mathcal{P} \quad \text{(A-Set)} \\
\langle \text{sp}[\emptyset], \mu, k, N, p_r \rangle & \xrightarrow{\text{call } p_x, \text{appl}(w, p_r)} \langle w[\emptyset], \mu, k \circ \cdot : \tau \rangle, N, p_r : \mathcal{P} \quad \text{(A-Ref)} \\
\langle \text{sp}[\emptyset], \mu, k, N, p_r \rangle & \xrightarrow{\text{call } p_x, \text{appl}(w, p_r)} \langle w[\emptyset], \mu, k \circ \cdot : \tau \rangle, N, p_r : \mathcal{P} \quad \text{(M-Call)} \\
\end{align*}
\]
\[ S^R \in \text{State}^8 = \text{Closure} \times \text{Storage} \times \text{Kont} \times \text{Map} \times \text{Returns} \]

\[ c \in \text{Ctrl} ::= x \mid (c \; c \; c) \mid \text{let } x = c_1 \text{ in } c_2 \mid v \mid (\lambda x : \tau . c) \]
\[ c_1; c_2 \mid c_1 \circ op \; c_2 \mid c_1 := c_2 \mid \text{hash } c \mid \text{fix } c \mid \text{ref } c \]
\[ \text{letrec } x : \tau = c_1 \text{ in } c_2; \mid \text{sp } \mid w \]

\[ cl \in \text{Closure} ::= c[e] \mid (cl \; cl) \mid \text{let } x = cl_1 \text{ in } cl_2 \]
\[ cl_1; cl_2 \mid cl_1 \circ op \; cl_2 \mid cl_1 := cl_2 \mid \text{hash } cl \mid \text{fix } cl \]
\[ \text{letrec } x : \tau = cl_1 \text{ in } cl_2 \mid \text{ref } cl \]

\[ v \in \text{Value} ::= \tau \rightarrow Fp \mid (\lambda x : \tau . c)[e] \mid b \mid \text{unit } \mid \text{l} \mid \overline{n} \]

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

\[ x \in \text{Var} \quad \text{a set of identifiers.} \]

\[ \text{op} \in \text{Oper} ::= + \mid - \mid * \mid < \mid > \mid == \]

\[ \tau \in \text{Type} ::= \text{Bool } \mid \text{Int } \mid \text{Unit } \mid \tau_1 \rightarrow \tau_2 \mid \text{Ref } \tau \]

\[ e \in \text{Env} = \star \mid e \cdot (cl/x) \]

\[ k \in \text{Kont} ::= [\cdot] \mid k[K : \tau \rightarrow \tau'] \mid k[\circ K : \tau] \mid k[\cdot K : \tau] \mid k[\cdot K : \tau] \]

\[ K \in \text{Kin} ::= [\cdot] \mid K[[\cdot] \; cl] \mid K[\; v[\cdot]] \mid K[(\text{if } [\cdot] \; cl_2 \; cl_3)] \]
\[ K[(\text{ref }[\cdot]) \mid K[(\text{hash }[\cdot])] \mid K[(\text{fix }[\cdot])] \mid K[(\text{op } cl_2)] \]
\[ K[(\text{let } x = [\cdot] \; cl_2)] \mid K[(\text{fix }[\cdot])] \mid K[(cl_1 \; op[\cdot])] \]

\[ n_1 \in \text{Name} = \text{an enumerable set.} \]

\[ N \in \text{Map} = \text{Name } \rightarrow (\text{Ctrl } \times \text{Type}) \]

\[ \langle x[e], \mu, k, N, p \rangle \xrightarrow{\tau} \langle cl, e, \mu, k, N, p \rangle \quad \text{where } e(x) = cl \quad \text{(M-V)} \]
\[ \langle (\lambda x : \tau.c)[e], \mu, k, N, p \rangle \xrightarrow{\tau} \langle (\lambda x : \tau.c)[e], \mu, k, N, p \rangle \quad \text{(M-PrL)} \]
\[ \langle (c_1 \; c_2), e, \mu, k \circ K : \tau, N, p \rangle \xrightarrow{\tau} \langle c_1[e], \mu, k \circ K[\cdot] \; c_2[e] : \tau, N, p \rangle \quad \text{(M-R1)} \]
\[ \langle v[e], \mu, k \circ K[[\cdot] \; c[e'][\cdot]] : \tau, N, p \rangle \xrightarrow{\tau} \langle c[e'], \mu, k \circ K[v[\cdot]] : \tau, N, p \rangle \quad \text{(M-R2)} \]
\[ \langle c[e' \cdot (v/x)], \mu, k \circ K : \tau, N, p \rangle \xrightarrow{\tau} \quad \text{(M-B)} \]
\[ \langle \text{if } c_1 \; c_2 \; c_3[e], \mu, k \circ K : \tau, N, p \rangle \xrightarrow{\tau} \quad \text{(M-R3)} \]
\[ \langle c_1[e], \mu, k \circ K[\text{if } [\cdot] \; c_2[e] \; c_3[e]] : \tau, N, p \rangle \xrightarrow{\tau} \quad \text{(M-IH)} \]
\[ \langle c_2[e'], \mu, k \circ K : \tau, N, p \rangle \xrightarrow{\tau} \quad \text{(M-IF)} \]
\[ \langle \text{false } e, \mu, k \circ K[\text{if } [\cdot] \; c_2[e'] \; c_3[e']] : \tau, N, p \rangle \xrightarrow{\tau} \quad \text{(M-R4)} \]
\[ \langle c_3[e'], \mu, k \circ K : \tau, N, p \rangle \xrightarrow{\tau} \quad \text{(M-Let)} \]
\[ \langle \text{let } x = c_1 \text{ in } c_2[e], \mu, k \circ K : \tau, N, p \rangle \xrightarrow{\tau} \quad \text{(M-Ref)} \]
\[ \langle c_1[e], \mu, k \circ K[\text{let } x = [\cdot] \text{ in } c_2[e]] : \tau, N, p \rangle \xrightarrow{\tau} \quad \text{(M-Let)} \]
\[ \langle c_2[e' \cdot (v/x)], \mu, k \circ K : \tau, N, p \rangle \xrightarrow{\tau} \quad \text{(M-Ref)} \]

30
4. Unfolding Closures

The previously derived LTS\(^R\) is transformed into a CESK machine \((M, L, \frac{L}{\lambda}M)\) where the state \(M\) is a quintuple \((c, e, k, N, p)\). The control \(c\) and substitution map \(e\) are obtained by unfolding the closures \(cl\) of LTS\(^R\). Note that closures remain the values used to encode \(\lambda\)-terms; the application of closures cannot be unfolded. **Proof Sketch of Proposition 3.** Given a bisimilarity \(\approx^M\) over the CESK machine (The CESK machine is also an LTS), the following compilation scheme \([\cdot]M:\)

\[
\begin{array}{c}
\langle e[\cdot], \mu, k \circ K[(\tau_1 \rightarrow \tau_2 Fp_f : \tau)] : \tau, N, p \rangle \\
\end{array}
\]

\[
\xrightarrow{\text{call } p_f(w)} \\
\langle \text{sp}[\cdot], \mu, k[K : \tau_2 \rightarrow \tau, N, p] \rangle
\]

compiles states \(S^R\) that are bisimilar in \(\approx^R\) into states \(M\) that are bisimilar in \(\approx^M\) as the transitions are unmodified.

**Machine Formalisation**

The formalisation of the machine is listed below. The most notable change here, besides the separate environment substate, is the change in the kontinuations. The kontinuations now store an explicit environment substate, to return to when control is returned to the kontinuation.

\[
M \in \text{State}'^M = \text{Ctrl} \times \text{Env} \times \text{Storage} \times \text{Kont} \times \text{Map} \times \text{Returns}
\]

\[
c \in \text{Ctrl} ::= x \mid (\lambda x : \tau.c) \mid (c \ c) \mid \text{if } c_1 c_2 c_3 \mid \text{let } x = c_1 \text{ in } c_2 \mid v
c_1; c_2 \mid c_1 \text{ op } c_2 \mid c_1 := c_2 \mid \text{ hash } c \mid \text{ fix } c \mid \text{ ref } c
\]

\[
v \in \text{Value} ::= \tau \rightarrow \tau \text{Fp} \mid (\lambda x : \tau.c)[e] \mid b \mid \text{unit } l_i \mid \pi
\]

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

\[
x \in \text{Var} \quad \text{ a set of identifiers.}
\]

\[
op \in \text{Oper} ::= + \mid - \mid * \mid < \mid > \mid ==
\]

\[
\tau \in \text{Type} ::= \text{Bool } \mid \text{ Int } \mid \text{ Unit } \mid \tau_1 \rightarrow \tau_2 \mid \text{ Ref } \tau
\]

\[
e \in \text{Env} = * \mid e \cdot (v \times)
\]

\[
k \in \text{Kont} ::= [] \mid k[K : \tau \rightarrow \tau'] \mid k[ \circ K : \tau] \mid k[ < K : \tau] \mid k[ > K : \tau]
\]

\[
K \in \text{Kin} ::= [] \mid K[(\cdot)[c(e)]] \mid K[v[\cdot]] \mid K[(\text{if }[\cdot] c_2 c_3)[e]]
\]

\[
K[(\text{ref}[])] \mid K[(\text{hash}[])] \mid K[(\cdot)[\cdot]] \mid K[(\text{fix}[])] \mid K[(\text{op}[])]
\]

\[
\text{N} \in \text{Name} \quad \text{ an enumerable set.}
\]

\[
\text{N} \in \text{Map} = \text{Name} \rightarrow (\text{Ctrl} \times \text{Type})
\]

32
\[ \langle x, e, \mu, k, N, \overline{p} \rangle \rightarrow (v, e, \mu, k, N, \overline{p}) \quad \text{where } e(x) = v \] (M-V)

\[ \langle \lambda x. \tau. c, e, \mu, k, N, \overline{p} \rangle \rightarrow \langle \lambda x. \tau. c[e], e, \mu, k, N, \overline{p} \rangle \] (M-PrfL)

\[ \langle c_1 \ c_2, e, \mu, k \circ K : \tau, N, \overline{p} \rangle \rightarrow \langle c_1, e, \mu, k \circ K[c_2[e]] : \tau, N, \overline{p} \rangle \] (M-R1)

\[ \langle v, e, \mu, k \circ K[\lambda x. \tau. c][e'] : \tau, N, \overline{p} \rangle \rightarrow \langle c, e', \mu, k \circ K[v \rightarrow \tau] : \tau, N, \overline{p} \rangle \] (M-B)

\[ \langle v, e, \mu, k \circ K(\lambda x. \tau. c)[e'] : \tau, N, \overline{p} \rangle \rightarrow \] (M-R2)

\[ \langle c, e' \cdot (v/x), \mu, k \circ K : \tau, N, \overline{p} \rangle \] (M-R3)

\[ \langle \text{if } c_1 \ c_2 \ c_3, e, \mu, k \circ K : \tau, N, \overline{p} \rangle \rightarrow \] (M-If)

\[ \langle c_1, e, \mu, k \circ K[(\text{if } c_2 \ c_3)[e']] : \tau, N, \overline{p} \rangle \] (M-IfF)

\[ \langle \text{let } x = c_1 \ c_2, e, \mu, k \circ K : \tau, N, \overline{p} \rangle \rightarrow \] (M-Let)

\[ \langle c_1, e, \mu, k \circ K[(\text{let } x = \lfloor \in c_2 \lfloor)[e'] : \tau, N, \overline{p} \rangle \rightarrow \langle c_2, e' \cdot (v/x), \mu, k \circ K : \tau, N, \overline{p} \rangle \] (M-R4)

\[ \langle \text{ref } c, e, \mu, k \circ K : \tau, N, \overline{p} \rangle \rightarrow \langle c, e, \mu, k \circ K[(\text{ref } c)[e'] : \tau, N, \overline{p} \rangle \] (M-Ra)

\[ \langle v, e, \mu, k \circ K[\text{ref } c] : \tau, N, \overline{p} \rangle \rightarrow \langle \lambda x. (\mu, 1 \rightarrow v), k \circ K : \tau, N, \overline{p} \rangle \] (M-Ref)

\[ \langle \text{l}, e, \mu, k \circ K : \tau, N, \overline{p} \rangle \rightarrow \langle c, e, \mu, k \circ K[(\text{l})[e'] : \tau, N, \overline{p} \rangle \] (M-Rd)

\[ \langle l_1, e, \mu, k \circ K[(\text{l})[e'] : \tau, N, \overline{p} \rangle \rightarrow \langle v, e, \mu, k \circ K : \tau, N, \overline{p} \rangle \] (M-Der)

\[ \text{where } \lambda (l_1) = v \]

\[ \langle \text{hash } c, e, \mu, k \circ K : \tau, N, \overline{p} \rangle \rightarrow \langle c, e, \mu, k \circ K[(\text{hash } c)[e'] : \tau, N, \overline{p} \rangle \] (M-Rh)

\[ \langle l_1, e, \mu, k \circ K[(\text{hash } c)[e'] : \tau, N, \overline{p} \rangle \rightarrow \langle \overline{l}, e, \mu, k \circ K : \tau, N, \overline{p} \rangle \] (M-Hash)

\[ \langle c_1 : c_2, e, \mu, k \circ K : \tau, N, \overline{p} \rangle \rightarrow \] (M-RSq)

\[ \langle c_1, e, \mu, k \circ K[\lfloor \in c_2 \rfloor[e'] : \tau, N, \overline{p} \rangle \rightarrow \langle c_2, e', \mu, k \circ K : \tau, N, \overline{p} \rangle \] (M-Seq)

\[ \langle c_1 \ := \ c_2, e, \mu, k \circ K : \tau, N, \overline{p} \rangle \rightarrow \] (M-Rs)

\[ \langle c_1, e, \mu, k \circ K[\lfloor \in \rfloor : \circ c_2[e'] : \tau, N, \overline{p} \rangle \rightarrow \langle c_2, e', \mu, k \circ K : \tau, N, \overline{p} \rangle \] (M-Set)

\[ \langle c_1, e, \mu, k \circ K[(\text{l})[e'] : \tau, N, \overline{p} \rangle \rightarrow \] (M-Set2)

\[ \langle v, e, \mu, k \circ K[(\text{l})[e'] : \tau, N, \overline{p} \rangle \rightarrow \langle \text{unit } e', \mu, \lambda x. (\mu, 1 \rightarrow v), k \circ K[e'] : \tau, N, \overline{p} \rangle \] (M-Rf)

\[ \langle (\text{fix } c), e, \mu, k \circ K : \tau, N, \overline{p} \rangle \rightarrow \langle c, e, \mu, k \circ K[(\text{fix } c)[e'] : \tau, N, \overline{p} \rangle \] (M-Fix)

\[ \langle \lambda x. \tau'. c'[e'], e, \mu, k \circ K[\text{fix } c] : \tau, N, \overline{p} \rangle \rightarrow \] (M-Fix)

\[ \langle c_1 \ \text{op } c_2, e, \mu, k \circ K : \tau, N, \overline{p} \rangle \rightarrow \langle c_1, e, \mu, k \circ K[\text{op } c_2[e']] : \tau, N, \overline{p} \rangle \] (M-Ro)

\[ \langle c_2, e', \mu, k \circ K[(\text{op } c_2)[e'] : \tau, N, \overline{p} \rangle \rightarrow \] (M-Op)

\[ \langle v, e, \mu, k \circ K[(\text{v } \text{op } c_2)[e'] : \tau, N, \overline{p} \rangle \rightarrow \langle v', e, \mu, k \circ K : \tau, N, \overline{p} \rangle \] (M-Op2)

\[ \langle v, e, \mu, k \circ [\cdot : \tau, N, \overline{p} \rangle \rightarrow \langle v, e, \mu, k \circ [\cdot : \tau, N, \overline{p} \rangle \] (M-Done)

33
schemes of Section 3.3 as follows.

This chain highlights the preservation and reflection of contextual equivalence in the source language MiniML down to its CESK implementation by means of bisimulations.

This chain of propagation through bisimulations may however, seem less efficient than a direct full abstraction result between MiniML and the derived

\[ \langle w, e, \mu, k \mid \tau \rangle, N, p_r : \bar{p} \xrightarrow{\text{ret } p_r} \langle \text{sp}, e, \mu, kN, \bar{p} \rangle \]  

(M-Marshout)

\[ \langle v, e, \mu, k \mid \tau \rangle, N, \bar{p} \xrightarrow{\tau} \langle v, e, \mu, k, N, \bar{p} \rangle \]  

(M-Marsin)

\[ \langle c, e, \mu, k[K : \tau], N, \bar{p} \rangle \xrightarrow{\tau} \langle c, e, \mu, k \circ K : \tau], N, \bar{p} \rangle \]  

(M-Wait)

\[ \langle \text{sp}, e, \mu, k, N, \bar{p} \rangle \xrightarrow{\text{call } p_{\text{ref}}(w)?} \langle w, e, \mu, k \mid \tau \rangle, N, \bar{p} \]  

(A-Retb)

\[ \langle \text{sp}, e, \mu, k, N, \bar{p} \rangle \xrightarrow{\text{call } p_{\text{ref}}(w,u,p_r)?} \langle l_i, \text{Ref } \tau \rangle, N(w_u) = (l_i, \text{Ref } \tau) \]  

(A-Deref)

\[ \langle w, e, \mu, k \mid \tau \rangle, N, p_r : \bar{p} \]  

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

\[ \langle \text{sp}, e, \mu, k[N, \bar{p}] \rangle \xrightarrow{\text{call } p_{\text{ref}}(w_u,w,p_r)?} \]  

(A-Set)

\[ \langle w, e, \mu, k \mid \tau \rangle, N, p_r : \bar{p} \]  

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

\[ \langle \text{sp}, e, \mu, k[N, \bar{p}] \rangle \xrightarrow{\text{call } p_{\text{ref}}(w_u,w,p_r)?} \]  

(A-Ref)

\[ \langle w, e, \mu, k[N, \bar{p}] \rangle \xrightarrow{\text{call } p_{\text{ref}}(w)?} \langle \text{sp}, e, \mu, k[K : \tau], N, \bar{p} \rangle \xrightarrow{\text{call } p_f(w)!} \]  

(M-Call)

3.4 Validation

Once combined, the formal properties of each of two steps of the methodology form a chain of bi-implications, similar to ones found in work on verifying multi-transformation compilers [19], as follows.

\[ \text{MiniML:} \quad t_1 \xrightarrow{\approx} t_2 \]

\[ \text{MiniML + FFI:} \quad \llbracket t_1 \rrbracket^{\text{FFI}} \xrightarrow{\approx} \llbracket t_2 \rrbracket^{\text{FFI}} \]

\[ \text{CESK:} \quad \llbracket t_1 \rrbracket^{\text{FFI}} \cdot \text{CESK} \xrightarrow{M} \llbracket t_2 \rrbracket^{\text{FFI}} \cdot \text{CESK} \]

where P. is short for Proposition and where \( \llbracket \cdot \rrbracket^{\text{CESK}} \) combines the compilation schemes of Section 3.3 as follows.

\[ \llbracket \cdot \rrbracket^{\text{CESK}} = \llbracket \cdot \rrbracket^{\text{K}} \cdot \text{CESK} \]

This chain highlights the preservation and reflection of contextual equivalence in the source language MiniML down to its CESK implementation by means of bisimulations.
CESK. However, the CESK is merely the final product of a series of systematic transformations that are applied to an LTS that models a FFI for MiniML. Because, as detailed in Section 3.3, these transformations affect only the internal reductions of the original LTS, these bisimilarity relations are not only obtained easily but also obviously coincide with the original bisimilarity $\approx$.

4 Implementation

The derived CESK machine has been implemented in C (available online\(^4\)) and deployed to the Fides implementation of PMA [23]. Fides implements PMA through use of a hypervisor that runs two virtual machines: one that handles the secure memory module and one handles the outside memory. One consequence of this architecture is that, as the low-level context interacts with the abstract machine, the Fides hypervisor will be forced to switch between the two virtual machines for each call and callback between the context and the module, producing a lot of overhead. Note that the new Intel SGX instruction set, which will provide this PMA technology in commercially available processors is likely to substantially reduce this overhead [17].

As illustrated in Figure 1, the CESK is compiled into the protected memory of the PMA mechanism. The machine state as well as the run-time stack and heap are placed in the protected data section, restricting the access to them to the transition rules of the CESK that reside in the protected code section. As explained in Section 3.1, access to these transition rules is provided to the attacker by means of entry points that enable the attacker to interact with the program running on the CESK in a controlled and secure manner.

Note that in our current implementation the MiniML program to be executed is loaded into the protected memory module at compile time. A possible extension to this work is thus to extend the implemented CESK with the secure authentication capabilities of Fides to enable trusted third parties to upload MiniML programs to the CESK machine.

\(^4\) https://github.com/sylvarant/secure-abstract-machine
To get an indication of the performance overhead of our secure implementation, we have benchmarked the overhead produced by our implementation for three scenarios. In the first scenario (Application) the low-level attacker applies a secure MiniML function to a boolean value. In the second scenario (Callback) the attacker applies a higher-order secure function to an attacker function, triggering a callback. In the third scenario (Read) the attacker dereferences a shared location.

<table>
<thead>
<tr>
<th></th>
<th>CESK</th>
<th>CESK + PMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>0.28µs</td>
<td>16.71 µs</td>
</tr>
<tr>
<td>Callback</td>
<td>0.33µs</td>
<td>40.92 µs</td>
</tr>
<tr>
<td>Read</td>
<td>0.24µs</td>
<td>16.88 µs</td>
</tr>
</tbody>
</table>

The tests were performed on a Dell Latitude with a 2.67 GHz Intel Core i5 and 4GB of DDR3 RAM. The Callback scenario incurs high overheads due to it involving more than one transition between secure and insecure memory.

5 Related Work

Our methodology relies on an assembly-level attacker model as detailed by Patrignani et al. [21]. An alternative attacker model would be a typed assembly language as developed by Morrisett et al. [18]. In that setting, however, the attacker model is weakened as it must be well-typed. An other option is the memory probing λ-calculus of Jagadeesan et al. [9].

Our methodology as detailed in Section 3 improves upon the interoperation semantics of Larmuseau et al. [12] to introduce a foreign function interface between the low-level attacker and MiniML. There exist multiple alternative foreign function interface designs: Matthews’ and Findler’s multi-language semantics [15] 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 [25]. These alternative interoperation techniques, however, rely on type checking to enforce security properties which does not defend against the presented low-level attacker model.

Step 2 of our methodology (Section 3.3) uses the syntactic correspondence of Biernacka and Danvy [2] between Curien’s λ-calculus and CESK machines. Other syntactic correspondences have focused on calculi with lazy evaluation and calculi with objects [3] and targeted other abstract machine types such as the Spineless Tagless G-machine [22]. An alternative approach to syntactic correspondence could be to adapt Ager et al.’s functional correspondence [1] between evaluators and abstract machines. In contrast to syntactic correspondence, functional correspondence derives new specifications from evaluators instead of the source language. This approach has been applied to multiple different languages and abstract machines, including Landin’s SECD machine [4] and multi-level languages [8].
Our notions of bisimularity over the derived LTS’s and CESK machine are based on the bisimulations for the \( \nu \)ref-calculus by Jeffrey and Rathke [10]. The proof of full abstraction for the introduced FFI is inspired by the full abstraction proof used for Fournet et al.’s secure compiler to Javascript [7].

A different line of research focusses on developing security architectures with access control mechanisms comparable to PMA: Flicker [16], Fides [23] and the Intel SGX [17]. 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. [20] for Fides-like PMA architectures, no results comparable to ours have been proven for these systems.

6 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: first extends Larmanseau et al.’s secure FFI with a realistic low-level attacker and subsequently applies Biernacka et al.’s syntactic correspondence. A concatenation of formal properties for each step of the methodology ensures that the result is secure.

There are different directions for future work. One is to investigate different abstract machine implementations and what changes (if any) must be done to the methodology to scale to them. Another direction is the integration of a secure abstract machine with runtime aspects of advanced programming languages such as, for example, garbage collection. Two challenges arise in this setting: implementing secure garbage collection and proving that this garbage collection does not introduce security leaks.

References

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