<http://www.diva-portal.org>

Postprint

This is the accepted version of a paper presented at *SEFM 2015, September 7–11, York, UK*.

Citation for the original published paper:

Larmuseau, A., Clarke, D. (2015)

Formalizing a secure foreign function interface.

In: *Software Engineering and Formal Methods* (pp. 215-230). Springer

Lecture Notes in Computer Science

http://dx.doi.org/10.1007/978-3-319-22969-0_16

N.B. When citing this work, cite the original published paper.

Permanent link to this version:

<http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-267089>

Formalizing a Secure Foreign Function Interface

Adriaan Larmuseau and Dave Clarke

Uppsala University, Sweden,
`first.last@it.uu.se`

Abstract. Many high-level functional programming languages provide programmers with the ability to interoperate with untyped and low-level languages such as C and assembly. Research into the security of such interoperation has generally focused on a closed world scenario, one where both the high-level and low-level code are defined and analyzed statically. In practice, however, components are sometimes linked in at run-time through malicious means. In this paper we formalize an operational semantics that securely combines MiniML, a light-weight ML, with a model of a low-level attacker, without relying on any static checks on the attacker. We prove that the operational semantics are secure by establishing that they preserve and reflect the equivalences of MiniML. To that end a notion of bisimulation for the interaction between the attacker and MiniML is developed.

Keywords: Language Interoperation, Full Abstraction, Bisimulation

1 Introduction

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

Preserving language abstractions is commonly formalised through a notion of *full abstraction*: if two source language terms t_1 and t_2 are *contextually equivalent*, indistinguishable to all source language contexts, then the terms must also be indistinguishable to all foreign language contexts that they interact with through the FFI and vice versa. Full abstraction thus preserves the abstractions of the source language in its interactions with the foreign language, ensuring that the programmer need only reason about the abstractions of the source language. Full abstraction does not, however, protect a programmer from writing insecure programs. That concern must be addressed in the design of the source language.

This paper introduces MiniML⁺ a formal model for a fully abstract FFI between a source language MiniML, a light-weight ML featuring references, tuples and recursion and a machine-level language such as assembly or C. The foreign language is not explicated in the model. It is instead simplified to an

attacker model MiniML^a that captures all the threats to full abstraction that a machine-level language may pose. To establish the security result we prove that an MiniML program set within MiniML⁺ preserves and reflects the equivalences of the original MiniML program. Because direct proofs over contextual equivalence are difficult we develop notions of bisimulation that coincide with contextual equivalence over MiniML and MiniML⁺. Full abstraction is then established by systematically relating the states of both notions of bisimulation.

This paper is an extension of our previous work on a formal model for a low-level memory protection mechanism [9]. In contrast to the basic λ -calculus addressed in that work, this paper considers a more complex source language, featuring references and non-trivial data types, and contributes a more complete formal model that considers in detail the exchange of data structures and memory locations. The introduced formal model differs from previous formalisms such as Matthews' and Findler's multi-language semantics [10] in that it is less abstract. In contrast to multi-language semantics where the concrete details of function calls and data exchange are left to the implementation, our model provides insight into how to implement these mechanisms in a secure fashion.

The remainder of this paper is organized as follows. Firstly the paper provides an overview of the source language MiniML and the attacker model MiniML^a (Section 2). The paper describes the secure FFI formalism MiniML⁺ (Section 3) and provides a proof of full abstraction (Section 4). Finally the paper presents related work (Section 5) and concludes (Section 6).

2 The Interoperating Languages

The secure FFI combines MiniML (Section 2.1) and an attacker model MiniML^a for a low-level language such as assembly or C (Section 2.2). The terms, types and contexts of MiniML are typeset in a black font, in contrast the terms and contexts of the attacker model MiniML^a are typeset in a **bold red font**.

2.1 The Source Language MiniML

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

$$\begin{aligned} t &::= v \mid x \mid (t_1 \ t_2) \mid \langle t_i^{i \in 1..n} \rangle \mid \text{op } t_1 \ t_2 \mid t.i \mid \text{if } t_1 \ t_2 \ t_3 \mid \text{ref } t \\ &\mid \text{let } x = t_1 \text{ in } t_2 \mid t_1 := t_2 \mid t_1; t_2 \mid \text{fix } t \mid !t \mid \text{hash } t \\ &\mid \text{letrec } x : \tau = t_1 \text{ in } t_2 \\ \text{op} &::= + \mid - \mid * \mid < \mid > \mid == \\ v &::= \langle v_i^{i \in 1..n} \rangle \mid \text{unit} \mid l_i \mid \bar{n} \mid (\lambda x : \tau. t) \mid \text{true} \mid \text{false} \\ \tau &::= \text{Bool} \mid \text{Int} \mid \text{Unit} \mid \tau_1 \rightarrow \tau_2 \mid \text{Ref } \tau \mid \langle \tau_i^{i \in 1..n} \rangle \\ E &::= [\cdot] \mid E \ t \mid v \ E \mid \text{op } E \ t \mid \text{op } v \ E \mid E.i \mid \dots \end{aligned}$$

Here $\bar{n}$ indicates the syntactic term representing the number n , the **letrec** operator for recursive expressions is syntactic sugar for a **let**-term combined with a **fix** operator 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 [2]. The operands `op` apply only to booleans and integers. Locations l_i are an artefact of the dynamic semantics that do not appear in the syntax used by programmers. The locations are tracked at run-time in a store $\mu ::= \emptyset \mid \mu, l_i = v$, which is assumed to be an ideal store: it never runs out of space. Allocating new locations is done deterministically $l_1, l_2, \dots, l_n$. The term `hash` t maps a location to its index: $l_i \mapsto \bar{i}$, similar to how Java's `hashCode` method converts references to integers. The reduction and type rules are standard and are thus omitted. The interested reader can find the full formalisation of the semantics in an accompanying technical report [8].

Contextual Equivalence The secure FFI aims to preserve the contextual equivalences of MiniML. A MiniML context C is a MiniML term with a single hole $[\cdot]$, two MiniML terms are contextually equivalent if and only if there is no context C that can distinguish them. Contextual equivalence is formalised as follows.

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

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

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

Two MiniML terms $\bar{1}$ and $\bar{2}$ are, for example, not contextually equivalent as a context $C = (\text{if } ([\cdot] == \bar{1}) \ \Omega \ \text{true})$, where Ω is a diverging term, can distinguish them. MiniML's λ -terms, in contrast, introduce many equivalences. There is no context C , for example, that can distinguish the following terms.

$$(\lambda x : \text{Int}. (+ x x)) \quad (\lambda x : \text{Int}. (* 2 x)) \quad (\text{Ex-1})$$

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. The following two terms, for example, are not contextually equivalent.

$$\begin{array}{ll} \text{let } x = \text{ref true in} & \text{let } x = \text{ref true in} \\ \text{let } y = \text{ref true in } x & \text{let } y = \text{ref true in } y \end{array} \quad (\text{Ex-2})$$

As the context $C = (\text{if } (\text{hash}[\cdot] == \bar{1}) \ \Omega \ \text{true})$ can distinguish both terms. 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.

$$\text{let } x = \text{ref false in } \bar{1} \quad \text{let } x = \text{ref true in } \bar{1} \quad (\text{Ex-3})$$

2.2 The Attacker Model MiniML^a

A malicious machine-level attacker can break the abstractions of a MiniML program that it interoperates with in two ways. Firstly the attacker can break the abstractions of MiniML by inspecting and manipulating the internal state of a running MiniML program. An attacker can achieve this by either reading and

writing to references shared through the FFI (a documented vulnerability in Java's JNI [13]) or by abusing low-level privileges to inspect the memory directly. Secondly, when interoperating a MiniML program will not only share terms over the FFI, but also receive terms. The attacker can take advantage of this by passing terms that do not adhere to the typing rules of MiniML.

Our attacker model incorporates both these threats to full abstraction. The attacker model is formalised as a language MiniML^a that is derived from MiniML by removing typing safety and incorporating reflection. Removing type safety is achieved by both removing the type checking rules as well as adding a new term **wr** that captures non reducible expressions such as the following one.

$$\mathbf{E}[\text{if } v \ t_2 \ t_3] \longrightarrow \mathbf{E}[\mathbf{wr}] \quad \text{where } v \neq \mathbf{true} \text{ or } v \neq \mathbf{false}$$

Reflection is added to MiniML^a by means of a syntactical equality testing operator modulo α -equivalence $\equiv_\alpha$. Given two terms t_1 and t_2 , the term $t_1 \equiv_\alpha t_2$ will thus only reduce to **true** if t_1 and t_2 are syntactically equal in all aspects but the names assigned to the variables.

Contextual Equivalence or Lack Thereof The addition of reflection in MiniML^a through the α -equivalence testing operator, renders the abstractions and source level restrictions of MiniML obsolete [15]. Consider, for example, the equivalent λ -terms of Ex-1 in Section 2.1. The following MiniML^a context:

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

distinguishes them due to the $\equiv_\alpha$ operator's ability to inspect the syntax of MiniML terms. A similar context **C** can thus be built for the contextually equivalent terms of Ex-3 and for every other pair of contextually equivalent terms outside of α -equivalent terms.

3 The MiniML^+ -Calculus: a Secure FFI

The MiniML^+ -calculus models the interoperation between MiniML and MiniML^a in a manner that secures the MiniML program from the MiniML^a attacker (Section 3.1). The MiniML^+ -calculus introduces new syntax (Section 3.2), new semantics (Section 3.3), new typing rules (Section 3.4), a modified notion of type soundness (Section 3.5) and definition of contextual equivalence (Section 3.6).

3.1 Overview

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

Separated Program States Preserving full abstraction when faced with a machine level attacker has been achieved by employing memory isolation mechanisms that prevent the attacker from directly accessing the memory of the program being secured [11]. To that end the program state P is split into two sub- states: the attacker state **A** and the secured program state **M** that incorporates the MiniML program. Formally a program P is defined as: $P = \mathbf{A} \parallel \mathbf{M}$.

Call Stacks To ensure that the program state is separable, the combined language must encode the interaction between both languages. To do so each state is extended with a call stack. The secure state $\mathbf{M}$ encodes this call stack as a type annotated stack of evaluation contexts $\Sigma ::= \bar{\mathbf{E}} : \tau \rightarrow \tau' \mid \epsilon$, where $\bar{\mathbf{E}}$ denotes a sequence of evaluation contexts $\mathbf{E}$ that represent the continuation of computation when a call to the attacker returns and are thus only to be filled in by input originating from the attacker. The stack of evaluation contexts is type annotated. In MiniML^+ these annotated types are incorporated into dynamic type checks to ensure that the input from the attacker does not break the type safety of the original MiniML program.

In contrast the attacker encodes the call stack through a sequence of contexts $\bar{\mathbf{C}}$ not a sequence of evaluation contexts $\mathbf{E}$. An evaluation context $\mathbf{E}$ is derived from call-by-value semantics, which limits the hole $[\cdot]$ to certain sub-terms. The evaluation context $\mathbf{E}$ is thus a less powerful threat to full-abstraction than the context $\mathbf{C}$, where the hole can be anywhere. More specifically, for each possible pair of terms $\mathbf{t}_1$ and $\mathbf{t}_2$ received from the MiniML program there exists a context $\mathbf{C}$ of the form: $(\text{if } (\mathbf{t}_1 \equiv_\alpha [\cdot]) \ \Omega \ \text{true})$ that can distinguish them.

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

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

A name $\mathbf{n}_i^f$ is created deterministically every time a λ -term is shared between the secure state and the attacker. The name $\mathbf{n}_1^f$ refers to the first shared λ -term, the name $\mathbf{n}_2^f$ refers to the second shared λ -term (even if it is the exact same λ -term as the first one) and so forth. In contrast the index i of the name $\mathbf{n}_i^l$ will correspond to the index of the location it refers to ($\mathbf{n}_i^l \mapsto l_i$). This is because the `hash` operation in MiniML allows a MiniML context/attacker to observe the index of the location, as illustrated in Ex-2 of Section 2.1. This observational power should thus not be taken away from the attacker in MiniML^+ .

Note that the names $\mathbf{n}_i^f$ and $\mathbf{n}_i^l$ are terms of the MiniML^a -calculus but not of MiniML. Also note that we don't compile or translate the λ -terms and locations into these names. They simply serve as a sharing mechanism, enabling the MiniML^+ -calculus to mask and track the locations and λ -terms shared by the MiniML program with the MiniML^a attacker.

3.2 Syntax

While basic values such as numbers and booleans can simply be converted to the correct representation when exchanged, no such conversion is possible for λ -terms and locations l_i . As detailed in Section 3.1, in MiniML^+ the MiniML^a attacker is restricted to reference objects formalized as names $\mathbf{n}_i^f$ and $\mathbf{n}_i^l$ that refer to λ -terms and locations shared by the MiniML program respectively. A MiniML^a attacker can compare these names through its α -equivalence testing operator $\equiv_\alpha$ and can also apply, read and write them in MiniML^+ using the newly added terms: $\text{call } \mathbf{n}_i^f \mathbf{v}$, $\text{deref } \mathbf{n}_i^l$ and $\text{set } \mathbf{n}_i^l \mathbf{v}$ respectively. The attacker can also create new names $\mathbf{n}_i^l$, that point to freshly allocated locations l_i in the MiniML program, through a term $\text{fref}^\tau \mathbf{v}$. Where τ represent the MiniML type that the attacker promises the value $\mathbf{v}$ conforms to. This promise is checked at run-time by MiniML^+ . The syntax of MiniML^a is thus extended as follows.

$$\mathbf{t} ::= \dots \mid \text{call } \mathbf{t}_1 \mathbf{t}_2 \mid \text{set } \mathbf{t}_1 \mathbf{t}_2 \mid \text{deref } \mathbf{t} \mid \text{fref}^\tau \mathbf{t} \quad \mathbf{v} ::= \dots \mid \mathbf{n}_i^f \mid \mathbf{n}_i^l$$

In contrast, the terms of MiniML are only extended with one new value: ${}^\tau\mathbf{F}(\lambda \mathbf{x}. \mathbf{t})$ that embeds a MiniML^a λ -term in MiniML, modelling an attacker function that the MiniML program can use. The type τ is included with the value to enable MiniML^+ to type check the use of this attacker function at run-time. The MiniML-calculus is not extended with a term to embed locations of MiniML^a , as manipulating the attacker memory harms the full abstraction result. This does not harm the interoperability, as the attacker can simply create an MiniML location through the $\text{fref}^\tau \mathbf{t}$ term instead of sharing its own.

The marshallng process of MiniML^+ transitions between terms of MiniML and MiniML^a within the secure state $\mathbf{M}$. The marshallng terms m are as follows.

$$m ::= v \mid \mathbf{v} \mid \langle m_i^{i \in 1..n} \rangle$$

The marshallng converts MiniML values to MiniML^a values and vice versa. Marshallng a tuple of size n is not immediate but takes n steps. To capture the intermediate state where some members are converted and others are not a tuple of terms m is included.

3.3 Operational Semantics

The reduction rules of the MiniML-calculus are denoted as $P \rightarrow P'$. As described in Section 3.1 a program $P = \mathbf{M} \parallel \mathbf{A}$ composes two states $\mathbf{M}$ and $\mathbf{A}$. The secure state $\mathbf{M}$ is either (1) executing a term t of type τ , (2) marshallng out values from MiniML to MiniML^a , (3) marshallng in input from the attacker that is expected to be of type τ or (4) waiting on input from the attacker.

$$(1) \mathbf{N}; \mu \Vdash \Sigma \circ t : \tau \quad (2) \mathbf{N}; \mu \Vdash \Sigma \triangleright m : \tau \quad (3) \mathbf{N}; \mu \Vdash \Sigma \triangleleft m : \tau \quad (4) \mathbf{N}; \mu \Vdash \Sigma$$

The attacker state takes two forms: it executes a term: $\mathbf{A} = \mu \Vdash \overline{\mathbf{C}} \circ \mathbf{t}$ or is suspended waiting on input from the MiniML program: $\mathbf{A} = \mu \Vdash \overline{\mathbf{C}}$.

For every possible program state P , the attacker or the secure state is thus suspended. We divide the reduction rules over the program state P into four

categories: internal computations, marshalling values in, marshalling values out and cross-boundary commands.

Internal Computations These are reduction rules that only affect the terms of one of the two languages. These are thus simply the reduction rules of MiniML and MiniML^a set within the program state of MiniML⁺, denoted as follows.

$$\begin{aligned} \mu \Vdash \bar{\mathbf{C}} \parallel \mathbf{N}; \mu \Vdash \Sigma \circ t : \tau &\rightarrow \mu \Vdash \bar{\mathbf{C}} \parallel \mathbf{N}; \mu' \Vdash \Sigma \circ t' : \tau & (\text{Internal MiniML}) \\ \mu \Vdash \bar{\mathbf{C}} \circ \mathbf{t} \parallel \mathbf{N}; \mu \Vdash \Sigma &\rightarrow \mu' \Vdash \bar{\mathbf{C}} \circ \mathbf{t}' \parallel \mathbf{N}; \mu \Vdash \Sigma & (\text{Internal MiniML}^a) \end{aligned}$$

Marshalling Values Out Whenever the embedded MiniML program reduces to a value v , that value needs to be converted to the appropriate representation before it is shared with the head of the attacker's call stack $\bar{\mathbf{C}}$. If the value is a location or a λ -term then it must be masked with a name $\mathbf{n}_i^l$ or $\mathbf{n}_i^f$, and the association between the name, the term and the term's type recorded in the map $\mathbf{N}$. Otherwise, the value is simply converted to the corresponding MiniML^a value. This conversion happens in a designated marshalling state as follows.

$$\mu \Vdash \bar{\mathbf{C}} \parallel \mathbf{N}; \mu \Vdash \Sigma \circ v : \tau \rightarrow \mu \Vdash \bar{\mathbf{C}} \parallel \mathbf{N}; \mu \Vdash \Sigma \triangleright v : \tau \quad (\text{Setup})$$

To save space in the following marshalling rules, we have compressed the state $\mu \Vdash \bar{\mathbf{C}} \parallel \mathbf{N}; \mu \Vdash \Sigma \triangleright m$ into a wrapper $\llbracket m \rrbracket_\tau^{\mathbf{N}}$ that denotes the only two constructs relevant to the marshalling process: the expected type τ and the map of shared names $\mathbf{N}$. Note the tuple conversion rule: it converts every member individually, ensuring that the embedded λ -terms and locations become names.

$$\begin{array}{c} \overline{\llbracket b \rrbracket_{\text{Bool}}^{\mathbf{N}} \rightarrow \llbracket \mathbf{b} \rrbracket_{\text{Bool}}^{\mathbf{N}}} \quad \overline{\llbracket \text{unit} \rrbracket_{\text{unit}}^{\mathbf{N}} \rightarrow \llbracket \text{unit} \rrbracket_{\text{unit}}^{\mathbf{N}}} \quad \overline{\llbracket \bar{n} \rrbracket_{\text{Int}}^{\mathbf{N}} \rightarrow \llbracket \bar{\mathbf{n}} \rrbracket_{\text{Int}}^{\mathbf{N}}} \\ \frac{j = |\mathbf{N}| + 1 \quad \mathbf{N}' = (\mathbf{N}, \mathbf{n}_j \mapsto ((\lambda x : \tau.t), \tau \rightarrow \tau'))}{\llbracket (\lambda x : \tau.t) \rrbracket_{\tau \rightarrow \tau'}^{\mathbf{N}} \rightarrow \llbracket \mathbf{n}_j^f \rrbracket_{\tau \rightarrow \tau'}^{\mathbf{N}'}} \quad \frac{\mathbf{N}' = (\mathbf{N}, \mathbf{n}_i^l \mapsto (l_i, \text{Ref } \tau))}{\llbracket l_i \rrbracket_{\text{Ref } \tau}^{\mathbf{N}} \rightarrow \llbracket \mathbf{n}_i^l \rrbracket_{\text{Ref } \tau}^{\mathbf{N}'}} \\ \frac{\forall i \in 1..n. \llbracket v_i \rrbracket_\tau^{N_i-1} \rightarrow \llbracket \mathbf{v}_i \rrbracket_\tau^{N_i}}{\llbracket \langle v_i^{i \in 1..n} \rangle \rrbracket_{\tau \times \tau'}^{\mathbf{N}} \rightarrow \llbracket \langle \mathbf{v}_i^{i \in 1..n} \rangle \rrbracket_{\tau \times \tau'}^{N_n}} \quad \frac{\exists i \in 1..n. \llbracket v_i \rrbracket_\tau^{\mathbf{N}} \rightarrow \llbracket \mathbf{wr} \rrbracket_\tau^{\mathbf{N}'}}{\llbracket \langle v_i^{i \in 1..n} \rangle \rrbracket_{\tau \times \tau'}^{\mathbf{N}} \rightarrow \llbracket \mathbf{wr} \rrbracket_{\tau \times \tau'}^{\mathbf{N}'}} \\ \frac{\tau = \tau_1 \rightarrow \tau_2}{\llbracket \tau \mathbf{F}(\lambda \mathbf{x}. \mathbf{t}) \rrbracket_\tau^{\mathbf{N}} \rightarrow \llbracket (\lambda \mathbf{x}. \mathbf{t}) \rrbracket_{\tau}^{\mathbf{N}}} \quad \frac{\emptyset \vdash t : \tau' \quad \tau \neq \tau'}{\llbracket t \rrbracket_\tau^{\mathbf{N}} \rightarrow \llbracket \mathbf{wr} \rrbracket_\tau^{\mathbf{N}}} \end{array}$$

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

$$\begin{aligned} \mu \Vdash \bar{\mathbf{C}}, \mathbf{C} \parallel \mathbf{N}; \mu \Vdash \Sigma \triangleright \mathbf{v} : \tau &\rightarrow \mu \Vdash \bar{\mathbf{C}} \circ \mathbf{C}[\mathbf{v}] \parallel \mathbf{N}; \mu \Vdash \Sigma & (\text{Share}) \\ \mu \Vdash \bar{\mathbf{C}}, \mathbf{C} \parallel \mathbf{N}; \mu \Vdash \Sigma \triangleright \mathbf{wr} : \tau &\rightarrow \mu \Vdash \bar{\mathbf{C}}, \mathbf{C}[\mathbf{wr}] \parallel \star, \emptyset \Vdash \varepsilon & (\text{Type Error}) \end{aligned}$$

Marshalling Values In Whenever the attacker reduces to a value and the secure state's call stack Σ is not empty the value is input into the secure state.

$$\mu \Vdash \bar{\mathbf{C}} \circ \mathbf{v} \parallel \mathbf{N}; \mu \Vdash \Sigma, \mathbf{E} : \tau \rightarrow \tau' \rightarrow \mu \Vdash \bar{\mathbf{C}} \parallel \mathbf{N}; \mu \Vdash \Sigma \triangleleft \mathbf{v} : \tau \quad (\text{Input})$$

The input value must be converted to the correct representation before it is plugged into the head of the stack of evaluation contexts Σ . Note that as denoted in the reduction rule **Input**, the marshalling rules will verify that the input value matches the argument type τ of the to-be-plugged evaluation context. The marshalling reduction rules are analogous to the previously detailed marshalling out rules in that they perform the reverse operation: they convert the input into the appropriate MiniML representation, fetching names from the map N instead of introducing names. The detailed rules are thus omitted here but can be found in the tech report [8]. When the marshalling succeeds the result is plugged into the head of Σ , otherwise M is cleared and the attacker updated to **wr**.

$$\begin{aligned} \mu \Vdash \bar{C} \parallel N; \mu \Vdash \Sigma, E : \tau \rightarrow \tau' \triangleleft v : \tau \rightarrow \mu \Vdash \bar{C} \parallel N', \mu \Vdash \Sigma \circ E[v] : \tau' & \quad (\text{Plug}) \\ \mu \Vdash \bar{C}, C \parallel N; \mu \Vdash \Sigma \triangleleft \mathbf{wr} : \tau \rightarrow \mu \Vdash \bar{C}, C[\mathbf{wr}] \parallel \star, \emptyset \Vdash \varepsilon & \quad (\text{Type Error}) \end{aligned}$$

Cross Boundary Commands The cross boundary commands enable the MiniML program to manipulate shared λ -terms and locations as follows.

$$\begin{aligned} \mu \Vdash \bar{C}, C \parallel N; \mu \Vdash \Sigma \circ E[(\tau_1 \rightarrow \tau_2 F(\lambda x. t) v)] : \tau \rightarrow & \quad (\text{M-Call}) \\ \mu \Vdash \bar{C}, C[(\lambda x. t) [\cdot]] \parallel N; \mu \Vdash \Sigma, E : \tau_2 \rightarrow \tau \triangleright v : \tau_1 & \end{aligned}$$

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

Note that this cross boundary function application serves as an input to the attacker as it is plugged into the top context/attack C of the attacker's call stack $\bar{C}$. This is because the attacker must be able to inspect this function call as accurately as the machine-level attacker who is able to observe which of its functions are called using which arguments.

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

$$\begin{aligned} \mu \Vdash \bar{C} \circ E[\mathbf{call} \mathbf{n}_i^f \mathbf{v}] \parallel N; \mu \Vdash \Sigma \rightarrow \mu \Vdash \bar{C} \circ \mathbf{v} \parallel N; \mu \Vdash \Sigma, (t [\cdot]) : \tau \rightarrow \tau' & \quad (\text{A-Call}) \\ \text{where } N(\mathbf{n}_i^f) = (t, \tau \rightarrow \tau') & \\ \mu \Vdash \bar{C} \circ E[\mathbf{set} \mathbf{n}_i^l \mathbf{v}] \parallel N; \mu \Vdash \Sigma \rightarrow \mu \Vdash \bar{C} \circ \mathbf{v} \parallel N; \mu \Vdash \Sigma, (l_i := [\cdot]) : \tau \rightarrow \mathbf{Unit} & \\ \text{where } N(\mathbf{n}_i^l) = (l_i, \mathbf{Ref} \tau) & \quad (\text{A-Set}) \\ \mu \Vdash \bar{C} \circ E[\mathbf{deref} \mathbf{n}_i^l] \parallel N; \mu \Vdash \Sigma \rightarrow \mu \Vdash \bar{C} \parallel N; \mu \Vdash \Sigma \circ !l_i : \tau & \quad (\text{A-Der}) \\ \text{where } N(\mathbf{n}_i^l) = (l_i, \mathbf{Ref} \tau) & \\ \mu \Vdash \bar{C} \circ E[\mathbf{fref}^\tau \mathbf{v}] \parallel N; \mu \Vdash \Sigma \rightarrow \mu \Vdash \bar{C} \circ v \parallel N; \mu \Vdash \Sigma, (\mathbf{ref} [\cdot]) : \tau \rightarrow \mathbf{Ref} \tau & \quad (\text{A-Ref}) \end{aligned}$$

A command from the attacker is not an input to the MiniML program, but rather a task it must carry out, and is as such not plugged into the head of the stack

of evaluation contexts Σ , but is instead executed on top the stack. As was the case for the function application by a MiniML program, applying a λ -term (A-Call), writing to a shared location (A-Set) or referencing a new location (A-Ref) requires two steps. In the first step a new evaluation context is constructed. In the second the argument is marshalled out as described previously. Dereferencing a shared MiniML location (A-Der) requires but one step as it involves only the shared name $\mathbf{n}_i^!$ and thus does not need to marshall out a value.

Note that in each of these rules the current evaluation context of the attacker ($\mathbf{E}$) is discarded. While discarding this evaluation context changes the way MiniML^a operates within the FFI, that does not affect its usefulness as an attacker model. On the contrary, we remove it to strengthen the attacker model. As detailed in Section 3.1, the contexts $\mathbf{C}$ of the attackers call stack $\overline{\mathbf{C}}$ pose a real threat to the abstractions MiniML, whereas an evaluation context $\mathbf{E}$ doesn't.

Note also that the rules that deal with type violations by the attacker are omitted as they are analogous to the error rules used by the marshalling rules.

3.4 MiniML⁺ Typing rules

A MiniML⁺ run-time program state P is type checked by type checking each individual evaluation context of the secure state's evaluation stack Σ as well as each association in the state's map N and each location in the secure store μ .

$$\begin{array}{c}
\frac{\Gamma \vdash N; \mu \Vdash \Sigma \quad \Gamma, x : \tau_1 \vdash N \Vdash E[x] : \tau_2}{\Gamma \vdash N; \mu \Vdash \Sigma, E : \tau_1 \rightarrow \tau_2} \quad \frac{\Gamma \vdash N \quad \Gamma \vdash \mu}{\Gamma \vdash N; \mu \Vdash \varepsilon} \\
\frac{\Gamma \vdash N; \mu \Vdash \Sigma \quad \Gamma \vdash t : \tau}{\Gamma \vdash N; \mu \Vdash \Sigma \circ t : \tau} \quad \frac{\Gamma \vdash N; \mu \Vdash \Sigma}{\Gamma \vdash N; \mu \Vdash \Sigma \triangleright m : \tau} \quad \frac{\Gamma \vdash N; \mu \Vdash \Sigma}{\Gamma \vdash N; \mu \Vdash \Sigma \triangleleft m : \tau} \\
\frac{}{\Gamma \vdash \star} \quad \frac{\Gamma \vdash N \quad \Gamma \vdash t : \tau}{\Gamma \vdash N; [\mathbf{n}_i \mapsto (t, \tau)]} \quad \frac{}{\Gamma \vdash \emptyset} \quad \frac{\Gamma \vdash \mu \quad \Gamma \vdash v : \tau}{\Gamma \vdash \mu, \mathbf{l}_i = v}
\end{array}$$

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

$$\frac{}{\Gamma \vdash {}^\tau\mathbf{F}(\lambda \mathbf{x}. \mathbf{t}) : \tau}$$

3.5 Type Soundness

Only the secure state M of a program P is type checked. As such we cannot rely on a traditional notion of type soundness. Instead, similar to Wadler's and Findler's blame calculus [14], we establish that whenever a program gets stuck or reduces to the error $\mathbf{wr}$ the attacker is at fault. As usual type soundness is split into theorems of progress and preservation. Proofs are omitted for space reasons, they are available in the accompanying tech report.

Theorem 1 (Preservation). *Given $\Gamma \vdash P$ and $P \rightarrow P'$ we have $\Gamma \vdash P'$.*

Theorem 2 (Progress). *Given $\Gamma \vdash P$ then if $P \rightarrow \mu \Vdash \mathbf{wr} \parallel \star, \emptyset \Vdash \varepsilon$ or $P \not\rightarrow P'$ then the attacker is the cause.*

3.6 Contextual Equivalence

The MiniML⁺-calculus program state P combines the secure state M and the attacker state A . However, our definition of contextual equivalence will only relate the secure states M that embed the MiniML program as preserving the security properties of MiniML in MiniML⁺ is the goal of this paper. The attacker state A thus serves as the context in which the secure state M operates.

Definition 2. Contextual equivalence over MiniML⁺ ($\simeq^+$) is defined as:

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

Consider, for example, the equivalent λ -terms of Ex-1 in Section 2.1. These λ -terms remain equivalent when placed within two secure states as follows.

$$\star; \emptyset \Vdash \varepsilon \circ (\lambda x : \text{Int}. (+ x x)) \quad \star; \emptyset \Vdash \varepsilon \circ (\lambda x : \text{Int}. (* 2 x))$$

There exists no attacker A that can distinguish these two secure states. The marshalling out rule for λ -terms will convert both λ -terms to the name $\mathbf{n}_1^f$ as they are the first λ -term to be shared with the attacker. An attacker A will observe that name, but cannot observe that the names refer to λ -terms that are not α -equivalent as, due to the dynamic type checking rules of MiniML⁺, it can only apply the name $\mathbf{n}_1^f$ to numbers $\bar{n}$ as in MiniML.

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

$$\star, \mathbf{n}_1^f \mapsto (\lambda x : \text{Ref Int}. \bar{1}); \emptyset \Vdash \varepsilon \quad \star, \mathbf{n}_1^f \mapsto (\lambda x : \text{Ref Int}. !x); \emptyset \Vdash \varepsilon$$

As an attacker $A = (\emptyset \Vdash (\text{if } (\bar{1} == [\cdot]) \ \Omega \ \text{true}), \text{call } \mathbf{n}_1^f [\cdot] \circ \text{fref}^\tau \bar{2})$ can distinguish them. Reducing $\text{fref}^\tau \bar{2}$ will result in both secure states returning a name $\mathbf{n}_1^f$ associated with a location l_1 where: $l_1 \mapsto \bar{2}$. This name $\mathbf{n}_1^f$ serves as input to the second context where the name $\mathbf{n}_1^f$ is applied to it. The name $\mathbf{n}_1^f$ refers to terms that are not equivalent in MiniML, a fact that is subsequently observed.

4 Full Abstraction

To establish that the MiniML⁺-calculus is a secure FFI, we show that the FFI preserves the equivalences between MiniML terms despite the presence of the attacker. Direct proofs over contextual equivalence are, however, difficult as one needs to reason about every reduction in every context. To that end we develop notions of bisimulation that coincide with contextual equivalence for MiniML (Section 4.1) and for MiniML⁺ (Section 4.2). Proving that the FFI is fully abstract is done by relating these bisimulations (Section 4.3).

4.1 Bisimulation for MiniML

We define a bisimulation relation $\mathcal{S}$ 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 λ -calculi with references. In this paper we use an applicative bisimulation that is a combination of Jeffrey's and Rathke's applicative bisimulation for the *vref*-calculus [6] and

the fully abstract trace semantics for the λ_μ hashref-calculus by Jagadeesan [5] 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 $(\zeta, \alpha, \xrightarrow{\alpha})$. 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 α of the LTS are defined as follows.

$$\alpha ::= \gamma \mid \tau \quad \gamma ::= @v \mid .i \mid l_i := v \mid \mathbf{ref} \ v \mid !l_i \mid l_i \mid b \mid \bar{n} \mid \mathbf{unit}$$

The most relevant labelled reductions are as follows.

$$\begin{aligned} K; \mu \mid t &\xrightarrow{\tau} K; \mu' \mid t' & (\text{Sil}) & \quad K; \mu \mid \bar{n} \xrightarrow{\bar{n}} K; \mu' \mid \bar{n} & (\text{O-N}) \\ K; \mu \mid (\lambda x : \tau. t) &\xrightarrow{@v} K; \mu \mid ((\lambda x : \tau. t) \ v) & \text{where } \vdash v : \tau & (\text{I-App}) \\ K; \mu \mid v &\xrightarrow{l_i := v'} K; \mu \mid l_i := v' & \text{where } l \in K \text{ and } \vdash l_i : \tau \text{ and } \vdash v : \tau & (\text{I-S}) \\ K; \mu \mid v &\xrightarrow{!l_i} K; \mu \mid !l & \text{if } l_i \in K & (\text{I-D}) \quad K; \mu \mid l_i \xrightarrow{l_i} K, l_i; \mu \mid v_i & (\text{O-L}) \\ K; \mu \mid v &\xrightarrow{\mathbf{ref} \ v'} K; \mu \mid \mathbf{ref} \ v' & (\text{I-Ref}) \quad K; \mu \mid \langle v_i^{i \in 1..n} \rangle \xrightarrow{\cdot i} K; \mu \mid v_i & (\text{I-Proj}) \end{aligned}$$

Reduction steps between terms cannot be observed by a context and are thus labelled as silent through the label τ (Sil). Whenever a MiniML program reduces to a value that is not a λ -term or tuple (as it may contain a λ -term), the context can observe that value (O-N, O-L). 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 λ -term by applying it to values (I-App), likewise a context queries members of a tuple instead of observing it directly (I-Proj). A context can also dereference observed locations l_i (I-D), create new ones (I-Ref) and assign them values (I-S).

We define a weak bisimulation over this LTS. In contrast to a strong bisimulation, such a bisimulation does not use the silent transitions between two states. Define the transition relation $\zeta \xRightarrow{\gamma} \zeta'$ as $\zeta \xrightarrow{\tau}^* \xrightarrow{\gamma} \zeta'$ where $\xrightarrow{\tau}^*$ is the reflexive transitive closure of the silent transitions $\xrightarrow{\tau}$. A Bisimulation over this LTS is now formally defined as follows.

Definition 3. *The relation $\mathcal{S}$ is a **bisimulation** iff $\zeta_1 \mathcal{S} \zeta_2$ implies:*

- (1) *Given $\zeta_1 \xRightarrow{\gamma} \zeta'_1$ there is a ζ'_2 such that: $\zeta_2 \xRightarrow{\gamma} \zeta'_2$ and $\zeta'_1 \mathcal{S} \zeta'_2$*
- (2) *Given $\zeta_2 \xRightarrow{\gamma} \zeta'_2$ there is a ζ'_1 such that: $\zeta_1 \xRightarrow{\gamma} \zeta'_1$ and $\zeta'_1 \mathcal{S} \zeta'_2$*

We denote bisimilarity, the largest bisimulation, as $\approx$. We now establish that the bisimilarity $\approx$ coincides with contextual equivalence $\simeq$.

Theorem 3 (Congruence). $t_1 \simeq t_2 \Leftrightarrow t_1 \approx t_2$.

Where $t_1 \approx t_2$ is short for $\emptyset; \emptyset \mid t_1 \approx \emptyset; \emptyset \mid t_2$. A proof of this theorem is an adaptation of existing results [6,5], as such we leave it to the accompanying tech report [8]. The proof splits the theorem into two sublemma: contextual equivalence implies bisimilarity (Completeness) and bisimilarity implies contextual equivalence (Soundness). The former is proven as per Gordon [4]. The latter is proven by induction over the number of reduction steps.

4.2 Bisimulation for the MiniML⁺-Calculus

We define a notion of bisimulation ($\mathcal{S}^+$) that coincides with the contextual equivalence relation ($\simeq^+$). Again we rely on an applicative bisimulation defined through an LTS. The LTS is a triple $(\mathbf{M}, \alpha^+, \xrightarrow{\alpha^+})$ where the secure states $\mathbf{M}$ are the states, α^+ the set of labels and $\xrightarrow{\alpha^+}$ the labelled transitions between states. The labels α^+ , which denote the observations of the attacker, are defined as follows.

$$\alpha^+ ::= \gamma^+ \mid \tau^+ \mid \checkmark$$

$$\gamma^+ ::= \mathbf{v}^? \mid \mathbf{v}^! \mid \mathbf{wr} \mid \gg (\lambda \mathbf{x}. \mathbf{t}) \mid \gg \mathbf{n}_i^! \mid \gg \mathbf{n}_i^f \mid \gg \mathbf{ref}^\tau \mid !\mathbf{n}_i^!$$

The labelled reductions of the LTS are of the form: $\mathbf{M} \xrightarrow{\gamma^+} \mathbf{M}'$. Although the attacker state $\mathbf{A}$ is not represented in these labelled reductions, the changes to the attacker state can be derived from the labels. The transitions are as follows.

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

$$\mathbf{N}; \mu \Vdash \Sigma \circ v : \tau \xrightarrow{\tau} \mathbf{N}; \mu \Vdash \Sigma \triangleright v : \tau \quad (\text{S-Setup})$$

$$\mathbf{N}; \mu \Vdash \Sigma, \mathbf{E} \triangleleft v : \tau \xrightarrow{\tau} \mathbf{N}; \mu \Vdash \Sigma \circ \mathbf{E}[v] : \tau \quad (\text{S-Plug})$$

$$\mathbf{N}; \mu \Vdash \Sigma \triangleleft m : \tau \xrightarrow{\tau} \mathbf{N}; \mu \Vdash \Sigma \triangleleft m' : \tau \quad (\text{S-MarshIN})$$

$$\mathbf{N}; \mu \Vdash \Sigma \triangleright m : \tau \xrightarrow{\tau} \mathbf{N}'; \mu \Vdash \Sigma \triangleright m' : \tau \quad (\text{S-MarshOut})$$

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

$$\mathbf{N}; \mu \Vdash \Sigma \triangleright \mathbf{v} : \tau \xrightarrow{\mathbf{v}^!} \mathbf{N}; \mu \Vdash \Sigma \quad (\text{M-V})$$

$$\mathbf{N}; \mu \Vdash \Sigma \triangleright \mathbf{wr} : \tau \xrightarrow{\mathbf{wr}} \star; \emptyset \Vdash \varepsilon \quad (\text{Wr-O}) \quad \mathbf{N}; \mu \Vdash \Sigma \xrightarrow{\mathbf{wr}} \star; \emptyset \Vdash \varepsilon \quad (\text{Wr-C})$$

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

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

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

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

$$\mathbf{N}; \mu \Vdash \Sigma \xrightarrow{\gg \mathbf{n}_i^!} \mathbf{N}; \mu \Vdash \Sigma, (l_i := [\cdot]) : \tau \rightarrow \mathbf{Unit} \quad \text{if } \mathbf{N}(\mathbf{n}_i^!) = (l_i, \mathbf{Ref} \tau) \quad (\text{S-N})$$

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

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

Setting and creating shared locations (S-N, A-R) or applying shared λ -terms (C-N, C-L), are as detailed in Section 3.3 two step operations which are captured

by two labels. In the first step, whose label is decorated with $\gg$, a new context is constructed that encodes the shared term and the operation to be performed on it. In the second step the argument is passed across the boundary as captured by the value sharing rules (A-V, M-V). Note that when the secure state applies a MiniML^a (C-L) the argument is marshalled first (S-MarshOut).

As in Section 4.1 we define weak bisimulation. Define the transition relation $M \xRightarrow{\gamma^+} M'$ as $M \xrightarrow{\tau^+}^* \xrightarrow{\gamma^+} M'$ where $\xrightarrow{\tau^+}$ is the reflexive transitive closure of the silent transitions $\xrightarrow{\tau^+}$. Bisimulation is now defined as follows.

Definition 4. *The relation $\mathcal{S}^+$ is a **bisimulation** iff $M_1 \mathcal{S}^+ M_2$ implies:*

- (1) *Given $M_1 \xRightarrow{\gamma^+} M'_1$ there is M'_2 such that: $M_2 \xRightarrow{\gamma^+} M'_2$ and $M'_1 \mathcal{S}^+ M'_2$*
- (2) *Given $M_2 \xRightarrow{\gamma^+} M'_2$ there is M'_1 such that: $M_1 \xRightarrow{\gamma^+} M'_1$ and $M'_1 \mathcal{S}^+ M'_2$*

Again, we denote bisimilarity as $\approx^+$ and prove that it is a congruence.

Theorem 4 (Congruence of the Bisimilarity). $M_1 \simeq^+ M_2 \iff M_1 \approx^+ M_2$.

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

Lemma 1. (Completeness) $M_1 \simeq^+ M_2 \Rightarrow M^1 \approx^+ M^2$.

Proof Sketch To prove that contextual equivalence implies bisimilarity we show that the contextual equivalence relation is itself a bisimulation. Assume that: $M_1 \simeq^+ M_2$. and that: $M_1 \xRightarrow{\gamma^+} M'_1$ (or its symmetry). We must show that there exists a M'_2 such that $M_2 \xRightarrow{\gamma^+} M'_2$ and that $M'_1 \simeq^+ M'_2$. The proof proceeds by case analysis on the labels γ^+ . The labels are divided into two camps: those produced by M and those produced by the attacker **A**. In the former case we prove the contra positive: if $M_1 \xRightarrow{\gamma^+} M'_1 \wedge M_2 \xRightarrow{\gamma^+} M'_2 \implies M_1 \not\approx^+ M_2$, by showing that for every scenario where the states produce different labels there exists a context **C** that can distinguish M_1 and M_2 . In the latter case we simply show that every label produced by the attacker can be encoded as a context **C**, because contextual equivalence is closed under contexts that suffices to imply the thesis.

Lemma 2. (Soundness) $M_1 \approx^+ M_2 \Rightarrow M_1 \simeq^+ M_2$.

The proof proceeds by induction on the number of reduction steps. We show that given $P_1 = \mathbf{A} \parallel M_1$ and $P_2 = \mathbf{A} \parallel M_2$ that P_2 diverges if and only if P_1 diverges. Full proofs for both lemmas are provided in the tech report.

4.3 The FFI Securely Embeds a MiniML Program

To prove that the FFI is secure we prove that injecting a MiniML term t into a secure state of MiniML⁺ as follows: $\{t\}^\uparrow \stackrel{\text{def}}{=} \star; \mu \Vdash \varepsilon \circ t : \tau$ where $\Gamma \vdash t : \tau$, will preserve the abstractions of t irrespective of which attacker **A** it faces. Formally stated: bisimilar terms in MiniML remain bisimilar when injected to MiniML⁺.

Theorem 5 (A Secure FFI). $t_1 \approx t_2 \iff \{t_1\}^\uparrow \approx^+ \{t_2\}^\uparrow$.

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

Lemma 3. (Preservation) $t_1 \approx t_2 \Rightarrow \{t_1\}^\uparrow \approx^+ \{t_2\}^\uparrow$.

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

(1) $\{t_1\}^\uparrow \mathcal{R} \{t_2\}^\uparrow$ and (2) that $\mathcal{R}$ relates M_1 and M_2 as would $\mathcal{S}^+$. We define $\mathcal{R}$ as $\mathcal{R} = \mathcal{R}_0 \cup \mathcal{R}_1 \cup \mathcal{R}_2 \cup \mathcal{R}_3$: one relation for each possible kind of M . The relation $\mathcal{R}_0$ relates two waiting states: $N; \mu \Vdash \Sigma$ and $N'; \mu' \Vdash \Sigma'$ and enforces that the name maps are equivalent: $Dom(N) = Dom(N') \wedge \forall \mathbf{n}_i. N(\mathbf{n}_i) \simeq N'(\mathbf{n}_i)$, and that the evaluation stacks are equivalent: $|\Sigma| = |\Sigma'| \wedge \forall E, E', t. E[t] \simeq E'[t]$. The relation $\mathcal{R}_1$ relates two states reducing terms t and t' requiring that $t_1 \approx t_2$ in addition to $\mathcal{R}_0$. The relations $\mathcal{R}_2$ and $\mathcal{R}_3$ relate the marshallng states, they require that $\mathcal{R}_0$ holds and that the marshalled terms are equal if they are terms of MiniML^a and contextually equivalent otherwise. Case (1) now follows from the assumption. Case (2) proceeds by analysis on the label γ^+ .

Lemma 4. (Reflection) $\{t_1\}^\uparrow \approx^+ \{t_2\}^\uparrow \Rightarrow t_1 \approx t_2$.

Proof Sketch We prove the contrapositive: $t_1 \not\approx t_2 \Rightarrow \{t_1\}^\uparrow \not\approx^+ \{t_2\}^\uparrow$. The proof has two cases. In the first case the bisimulation fails immediately as t_1 and t_2 produce different transitions after silent reduction: $\emptyset; \emptyset \mid t_1 \xrightarrow{\gamma} \zeta'_1 \wedge \# \zeta'_2. \emptyset; \emptyset \mid t_2 \xrightarrow{\gamma} \zeta'_2$ (or it's symmetry). In this case we derive the thesis by case analysis over the labels γ . In the second case there is a sequence of context actions ($@v, \mathbf{ref} \ v, !l_i, l_i := v$) that result in two states where different LTS transitions apply. In this case we establish the thesis by showing that each MiniML context action can be replicated by an MiniML⁺ attacker action.

Full proofs for both lemmas are provided in the tech report [8].

5 Related Work

This paper extended and refined a secure interoperation semantics for the λ -calculus introduced in previous work [9] with references, non-trivial data types and data marshallng. Formalisations that capture foreign function interface implementations have been developed before. Matthews' and Findler's multi-language semantics [10] enable two languages to interoperate in a way that preserves termination and type safety. In their work however, they aim to abstract away low-level details and instead focus on semantic properties. Our formalism in contrast, focusses on lifting low-level interoperation details into the formalism to study their effect on security. Furr and Foster investigate a sound FFI between OCaml and C, by developing a multi-language type system that embeds OCaml types in C and vice-versa [3]. They, however, assume that the C code is not an attacker and will thus not circumvent their typing system. Tan et al. propose a framework that adds type safety to the default Java FFI [13]. Their system however, requires both static and dynamic checks on the C code. Our formalism in contrast, details an FFI that does not enforce any static checks.

The notions of applicative bisimulations for MiniML and MiniML⁺ are based on the applicative bisimulation for the ν ref-calculus by Jeffrey and Rathke [6], the fully abstract trace semantics for the λ_μ hashref-calculus by Jagadeesan [5] and the trace semantics for general references by Laird [7]. The labels of our

bisimulation differ from the labels used in the latter as our labels do not explicitly state the shared location store. The proof of congruence for the bisimulations over MiniML^+ relies on Gordon’s proof of congruence for FPC [4]. A possible alternative to the applicative bisimulation for MiniML^+ are the environmental bisimulations of Sangiorgi et al. [12]. Our definition of bisimulation is however much simpler than their respective definitions, as the names used in the formalism of MiniML^+ are not local but global and denumerable.

6 Conclusions

This paper introduced a formal model for a foreign function interface between a light-weight ML-like programming language and a low-level attacker. The FFI is secure in that it preserves the abstractions of the ML-like in its interactions with the low-level attacker. This security property was proven by establishing that contextually equivalent terms in the ML-like remain contextually equivalent when interoperating with the low-level attacker through the FFI.

References

1. M. Abadi. Protection in programming-language translations. In *Secure Internet Programming*, volume 1603 of *LNCS*, 1999.
2. M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2):235–271, Sept. 1992.
3. M. Furr and J. S. Foster. Checking type safety of foreign function calls. *TOPLAS*, 30(4), July 2008.
4. A. Gordon. *Bisimilarity as a Theory of Functional Programming: Mini-course*. BRICS notes series. Computer Science Department, 1995.
5. R. Jagadeesan, C. Pitcher, J. Rathke, and J. Riely. Local memory via layout randomization. In *CSF ’11*. IEEE, 2011.
6. A. Jeffrey and J. Rathke. Towards a theory of bisimulation for local names. Computer Science Report 02-2000, University of Sussex, 2000.
7. J. Laird. A fully abstract trace semantics for general references. In *Automata, Languages and Programming*, volume 4596 of *LNCS*, pages 667–679. Springer, 2007.
8. A. Larmuseau and D. Clarke. Formalizing a secure foreign function interface - extended version. Technical Report 2015-015, Uppsala University, May 2015.
9. A. Larmuseau, M. Patrignani, and D. Clarke. Operational semantics for secure interoperation. In *PLAS Workshop ’14*. ACM, 2014.
10. J. Matthews and R. B. Findler. Operational semantics for multi-language programs. *TOPLAS*, 31(3), Apr. 2009.
11. M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens. Secure compilation to protected module architectures. *ACM TOPLAS*, 2015.
12. D. Sangiorgi, N. Kobayashi, and E. Sumii. Environmental bisimulations for higher-order languages. *ACM TOPLAS*, 33(1):5:1–5:69, Jan. 2011.
13. G. Tan, S. Chakradhar, R. Srivaths, and R. D. Wang. Safe Java native interface. In *ESSoS*, pages 97–106, March 2006.
14. P. Wadler and R. B. Findler. Well-typed programs can’t be blamed. In *ESOP ’09*, pages 1–16. Springer-Verlag, 2009.
15. M. Wand. The theory of fexprs is trivial. *Lisp and Symbolic Computation*, 10(3):189–199, 1998.