Implementing a Secure Abstract Machine

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

[1]Dept. of IT, Uppsala University, Sweden
first.last@it.uu.se

[2]iMinds-Distrinet, KU Leuven, Belgium
first.last@cs.kuleuven.be

ABSTRACT

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 ML, 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.

CCS Concepts

•**Security and privacy** → *Software security engineering; Domain-specific security and privacy architectures;*

Keywords

Abstract Machine; Memory Protection

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, OCaml's bytecode runs on the Zinc abstract machine and the Glasgow Haskell Compiler uses the Spineless Tagless G-machine internally [7].

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 [15].

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) [15]. 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 programs within this protected memory without sacrificing their ability to 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.* [8], with Patrignani *et al.*'s trace semantics of a realistic low-level attacker [13]. 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.

ACM 978-1-4503-3738-0.
http://dx.doi.org/10.1145/0000000.0000000

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.

$$
\begin{array}{lll}
t & ::= v \mid x \mid (t_1\ t_2) \mid t_1\ \texttt{op}\ t_2 \mid \texttt{if}\ t_1\ t_2\ t_3 \mid \texttt{ref}\ t \\
& \quad \mid t_1 := t_2 \mid t_1; t_2 \mid \texttt{let}\ x = t_1\ \texttt{in}\ t_2 \mid !t \mid \texttt{fix}\ t \\
& \quad \mid \texttt{hash}\ t \mid \texttt{letrec}\ x : \tau = t_1\ \texttt{in}\ t_2 \\
\texttt{op} & ::= + \mid - \mid * \mid < \mid > \mid == \\
v & ::= \texttt{unit} \mid l_i \mid \overline{n} \mid (\lambda x : \tau.t) \mid \texttt{true} \mid \texttt{false} \\
\tau & ::= \texttt{Bool} \mid \texttt{Int} \mid \texttt{Unit} \mid \tau_1 \rightarrow \tau_2 \mid \texttt{Ref}\ \tau \\
\texttt{E} & ::= [\cdot] \mid \texttt{E}\ t \mid v\ \texttt{E} \mid \texttt{E}\ \texttt{op}\ t \mid \texttt{op}\ v\ \texttt{E} \mid \ldots
\end{array}
$$

Here $\overline{n}$ indicates the syntactic term representing the number n, τ 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. The `letrec` operator is syntactic sugar for a combination of `let` and `fix`. The operands `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 $\mu ::= \emptyset \mid \mu, l_i = v$, which is assumed to be an ideal store: it never runs out of space. Allocating new locations is done deterministically $l_1, l_2, .., l_n$. 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 tech report [9].

2.2 Contextual Equivalence

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.

DEFINITION 1. **Contextual equivalence** *is defined as:*

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

where $\Uparrow$ denotes divergence, t_1 and t_2 are closed terms and neither the terms 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 τ as a context C observes the same typing rules as the terms.

MiniML's λ-terms introduce many equivalences, there is no context C, for example, that can distinguish the following two λ-terms.

$$(\lambda x : \texttt{Int}.\ \overline{0}) \qquad (\lambda x : \texttt{Int}.(x\ -\ x)) \qquad \text{(Ex-1)}$$

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, still produce equivalences as a context C cannot observe locations that do not leave a term. The following two terms, for example, are thus contextually equivalent.

$$(\texttt{ref false};\ \overline{0}) \qquad (\texttt{ref true};\ \overline{0}) \qquad \text{(Ex-2)}$$

Contextual equivalence can be used to capture security properties such as *confidentiality* and *integrity* at the language level. This is illustrated for the following two examples.

Confidentiality. Consider the following two contextually equivalent MiniML terms.

$$
\begin{array}{ll}
\texttt{let}\ secret = \texttt{ref}\ \overline{0}\ \texttt{in} & \texttt{let}\ secret = \texttt{ref}\ \overline{0}\ \texttt{in} \\
\quad (\lambda x : \texttt{Int}.\ secret{+}{+}; x) & \quad (\lambda x : \texttt{Int}.\ x)
\end{array}
$$

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.

$$
\begin{array}{ll}
(\lambda x : \texttt{Int} \rightarrow \texttt{Int}. & (\lambda x : \texttt{Int} \rightarrow \texttt{Int}. \\
\quad \texttt{let}\ y = \texttt{ref}\ \overline{0}\ \texttt{in} & \quad \texttt{let}\ y = \texttt{ref}\ \overline{0}\ \texttt{in}\ (x\ \overline{1})) \\
\quad\quad \texttt{let}\ r = (x\ \overline{1})\texttt{in} & \\
\quad\quad\quad \texttt{if}\ (!y = \overline{0})\ r\ (\overline{-1})) &
\end{array}
$$

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 The Security Challenges 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 [15]. 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 could break the confidentiality and integrity requirements of MiniML programs.

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 [16]).

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, 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.

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

A variety of PMA implementations exist. While our current implementation of the abstract machine makes use of a research prototype [15], Intel is developing a new instruction set, referred to as SGX, that enables the usage of PMA in commercial processors [11].

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 [13].

These trace semantics transition over a state Λ 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 Λ can be $(\texttt{unknown}, m, s)$: 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.

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

A label L is either an observable action α or a non-observable action τ. Decorations ? and ! indicate the direction of an observable action: from the unprotected memory to the protected memory (?) or vice-versa (!). Observable actions γ are function calls or returns to a certain address p, combined with a sequence of 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 α can be a tick $\surd$ 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 $\mathsf{Tr}(P)$, are computed as follows: $\mathsf{Tr}(P) = \{\overline{\alpha} | \exists \Lambda. \Lambda_0(P) \overset{\overline{\alpha}}{\Longrightarrow\!\!\!\!\!\rightarrow} \Lambda\}$. Where Λ_0 is the initial state and the relation $\Lambda \overset{\overline{\alpha}}{\Longrightarrow\!\!\!\!\!\rightarrow} \Lambda'$ describes the traces generated by transitions between states. Whenever two assembly programs P_1 and P_2 are contextually equivalent they will have the same traces (the same inputs and observations by attackers) as follows:

PROPOSITION 1 (FULLY ABSTRACT TRACES [13]).
$$P_1 \simeq_a P_2 \iff \mathsf{Tr}(P_1) = \mathsf{Tr}(P_2)$$

where $\simeq_a$ denotes contextual equivalence between two assembly programs.

3. DERIVING A SECURE CESK MACHINE

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 (`C`ontrol), a map from variables to values (`E`nvironment), a map from locations to values (`S`tore) and a stack of evaluation contexts (`K`ontinuations) [4].

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.* [8] (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.

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 3.2).

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 3.3).

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.* [8]. The resulting extension of MiniML is formalised by means of a labelled transition system (LTS). Concretely this LTS is a triple $(\mathtt{S}, L, \xrightarrow{L})$, where the state $\mathtt{S}$ 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 $\xrightarrow{L}$ denotes the labelled transitions between the states is $\mathtt{S} \xrightarrow{L} \mathtt{S}'$. 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 $\mathtt{S}$ is a quadruple: $\langle \omega, \mu, \mathtt{N}, \overline{p} \rangle$. The first element ω is a substate that captures the four different execution states of MiniML throughout the FFI. The second element μ is the store of locations (Section 2.1). The third element of the state $\mathtt{S}$ ($\mathtt{N}$) 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 $\mathtt{S}$ 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 the attacker is handled, the transition rules and the formal security of this FFI.

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 τ, (2) marshalling out values to be returned to the attacker, (3) marshalling in input from the attacker that is expected to be of type τ or (4) waiting on input from the attacker.

$$(1)\ \Sigma \circ t : \tau \quad (2)\ \Sigma \rhd m : \tau \quad (3)\ \Sigma \lhd m : \tau \quad (4)\ \Sigma$$

Where $m ::= v \mid w$ 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 Σ: 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:

$$\Sigma ::= \varepsilon \mid \Sigma, \mathtt{E} : \tau \to \tau'$$

where, as explained in Section 2.1, $\mathtt{E}$ 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 λ-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 λ-terms and locations have been shared with the attacker. Larmuseau *et al.* [8] model reference objects for λ-terms and locations through names $\mathtt{n}_i^{\mathtt{f}}$ and $\mathtt{n}_i^{\mathtt{l}}$ respectively. Both names are tracked in a map $\mathtt{N}$ that records the associated term and type, as follows.

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

A *fresh* name $\mathtt{n}_i^{\mathtt{f}}$ is created deterministically every time a λ-term is shared between MiniML and the attacker. The index i of these names $\mathtt{n}_i^{\mathtt{f}}$ denumerates the shared λ-terms. The first λ-term to be shared will be masked by a name $\mathtt{n}_1^{\mathtt{f}}$, the second λ-term will be masked by a name $\mathtt{n}_2^{\mathtt{f}}$ and so on. In contrast, the index i of the name $\mathtt{n}_i^{\mathtt{l}}$ will correspond to the index of the location it refers to $(\mathtt{n}_i^{\mathtt{l}} \mapsto l_i)$. Larmuseau *et al.* have previously proven that these names and their indices i: do not reveal to an attacker any information not available to MiniML contexts and thus do not violate the contextual equivalences of MiniML [8].

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 $p^e_{\mathtt{deref}}$.

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 $p^e_{\mathtt{retb}}$ that handles the reverting of control.

Attacker input. Input from the attacker is handled in the *marshalling in* state of the FFI: $\langle \Sigma \lhd m : \tau, \mu, \mathtt{N}, \overline{p} \rangle$. Whenever the input does not observe the type τ, the MiniML program errors (**wr**) and subsequently terminates.

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: ${}^{\tau\to\tau'}\mathtt{F}p$, where the type $\tau \to \tau'$ 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 [13] 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 $p^e_{\mathtt{alloc}}$. The attacker is only able to write a value to such a shared MiniML location through the entry point $p^e_{\mathtt{set}}$, which ensures that this occurs in a type safe manner.

Transitions. The most relevant transitions are as follows.

$$\langle \Sigma \circ t : \tau, \mu, \mathtt{N}, \overline{p}\rangle \xrightarrow{\tau} \langle \Sigma \circ t' : \tau, \mu', \mathtt{N}, \overline{p}\rangle \quad \text{(Silent)}$$

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

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

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

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

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

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

$$\langle \Sigma, \mu, \mathtt{N}, \overline{p}\rangle \xrightarrow{call\ p^e_{\mathtt{set}}(w_n, w, p_r)?} \langle (\Sigma, (l_i := [\cdot]) : \tau \to \mathtt{Unit} \lhd w : \tau), \mu, \mathtt{N}, p_r : \overline{p}\rangle \quad \text{(A-Set)}$$
$$where\ \mathtt{N}(w_n) = (l_i, \mathtt{Ref}\ \tau)$$

Transitions within the MiniML program are labelled as silent through the label τ (Silent). Errors, either by faulty calls or type inappropriate input by the attacker (Wr-I) are followed by immediate termination $\surd$. To start the computation of the MiniML program, the low-level attacker calls the entry point $p^e_{\mathtt{start}}$ 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 through the return entry point $p^e_{\mathtt{retb}}$ (A-Ret).

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 example operations listed above, handle function calls (A-Apply) and the setting and dereferencing of shared location (A-Set,A-Deref). Note that during each of these operations the expected type is always inferred at run-time to ensure that the operations happen in a type safe manner.

Full abstraction. 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 *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 α. Define the transition relation $\mathtt{S} \overset{\alpha}{\Longrightarrow} \mathtt{S}'$ as $\mathtt{S} \xrightarrow{\tau}{}^{*} \xrightarrow{\alpha} \mathtt{S}'$ where $\xrightarrow{\tau}{}^{*}$ is the reflexive transitive closure of the silent transitions $\xrightarrow{\tau}$.

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

1. *If $\mathtt{S}_1 \overset{\alpha}{\Longrightarrow} \mathtt{S}_1'$, $\exists \mathtt{S}_2'.\ \mathtt{S}_2 \overset{\alpha}{\Longrightarrow} \mathtt{S}_2'$ and $\mathtt{S}_1'\ \mathcal{B}\ \mathtt{S}_2'$.*
2. *If $\mathtt{S}_2 \overset{\alpha}{\Longrightarrow} \mathtt{S}_2'$, $\exists \mathtt{S}_1'.\ \mathtt{S}_1 \overset{\alpha}{\Longrightarrow} \mathtt{S}_1'$ and $\mathtt{S}_1'\ \mathcal{B}\ \mathtt{S}_2'$.*

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

To reason about the security properties of the FFI we must first define a compilation scheme that places a MiniML term t into a state $\mathtt{S}$ of the FFI.

$$[\![t]\!]^{FFI} \overset{\text{def}}{=} \langle \boldsymbol{\varepsilon} \circ t : \tau, \emptyset, \star, \emptyset\rangle$$

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

Proposition 2 (Full Abstraction).
$$[\![t_1]\!]^{FFI} \approx [\![t_2]\!]^{FFI} \iff t_1 \simeq t_2$$

The proof proceeds by relating the bisimilarity $\approx$ to a *congruent* bisimilarity over MiniML, a bisimilarity that coincides with the contextual equivalences of MiniML.

A complete formalisation of the FFI and the proof of full abstraction is listed in the associated tech report [9].

3.2 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: $[\![\cdot]\!]^T$ that compiles the current state $\mathtt{S}$ into the derived machine state $\mathtt{S}^T$, 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).
$$\mathtt{S}_1 \approx \mathtt{S}_2 \iff [\![\mathtt{S}_1]\!]^T \approx^T [\![\mathtt{S}_2]\!]^T$$

Each step of the syntactic correspondence was implemented in Ocaml[1] 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 [9].

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 $(\mathtt{S}^K, L, \xrightarrow{L}{}^K)$. The new state $\mathtt{S}^K$ is a quintuple $\langle \mathtt{c}, \mu, \mathtt{k}, \mathtt{N}, \overline{p} \rangle$ where $\mathtt{c}$ is a control and $\mathtt{k}$ the context. A control $\mathtt{c}$ is either a MiniML term t, a low-level word w (when marshalling) or a stop state.

$$\mathtt{c} ::= t \mid w \mid \mathtt{sp}.$$

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

$$\mathtt{K} ::= [\cdot] \mid \mathtt{K}[[\cdot]\ \mathtt{c}] \mid \mathtt{K}[\mathtt{v}\ [\cdot]] \mid \ldots$$

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

$$\mathtt{k} ::= [\cdot] \mid \mathtt{k}[\mathtt{K} : \tau \to \tau'] \mid \mathtt{k}[\circ\, \mathtt{K} : \tau] \mid \mathtt{k}[\lhd\, \mathtt{K} : \tau] \mid \mathtt{k}[\rhd\, \mathtt{K} : \tau]$$

The new transition rules $\xrightarrow{L}{}^K$ differ from $\xrightarrow{L}$ in that they include explicit transitions to plug and construct the continuation contexts $\mathtt{K}$. 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.

$$\langle (\mathtt{c}_1\ \mathtt{c}_2), \mathtt{k}[\circ\, \mathtt{K} : \tau], \mu, \mathtt{N}, \overline{p} \rangle \xrightarrow{\tau} \langle \mathtt{c}_1, \mathtt{k}[\circ\, \mathtt{K}[[\cdot]\ \mathtt{c}_2] : \tau], \mu, \mathtt{N}, \overline{p} \rangle \quad \text{(M-R1)}$$

Proof Sketch of Proposition 3 Given a bisimilarity $\approx^K$ over LTS^K, the following compilation scheme $[\![\cdot]\!]^K$:

$$\begin{aligned}
[\![\langle \Sigma \circ \mathtt{t} : \tau, \mu, \mathtt{N}, \overline{p} \rangle]\!]^K &= \langle \mathtt{t}, [\![\mathtt{k}]\!]^E[\circ\, [\cdot] : \tau], \mu, \mathtt{N}, \overline{p} \rangle \\
[\![\langle \Sigma \lhd m : \tau, \mu, \mathtt{N}, \overline{p} \rangle]\!]^K &= \langle m, [\![\mathtt{k}]\!]^E[\lhd\, [\cdot] : \tau], \mu, \mathtt{N}, \overline{p} \rangle \\
[\![\langle \Sigma \rhd m : \tau, \mu, \mathtt{N}, \overline{p} \rangle]\!]^K &= \langle m, [\![\mathtt{k}]\!]^E[\rhd\, [\cdot] : \tau], \mu, \mathtt{N}, \overline{p} \rangle \\
[\![\langle \Sigma, \mu, \mathtt{N}, \overline{p} \rangle]\!]^K &= \langle \mathtt{sp}, [\![\mathtt{k}]\!]^E[\rhd\, [\cdot] : \tau], \mu, \mathtt{N}, \overline{p} \rangle \\
[\![\Sigma, \mathtt{E} : \tau, \overline{p}]\!]^E &= [\![\Sigma]\!]^E[\mathtt{K} : \tau] \mid [\![\varepsilon]\!]^E = [\cdot]
\end{aligned}$$

where $\mathtt{K}$ *explicitates* $\mathtt{E}$

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

2. Closure Conversion. In this second transformation the λ-terms of MiniML are converted into closures. As a result, the previously derived LTS^K is transformed into a new LTS^C $(\mathtt{S}^C, L, \xrightarrow{L}{}^C)$. The new state $\mathtt{S}^C$ is a quintuple $\langle \mathtt{cl}, \mu, \mathtt{k}, \mathtt{N}, \overline{p} \rangle$ where the control terms are now closures $\mathtt{cl}$ as in the $\lambda_{\hat{\rho}}$-calculus [2]:

$$\mathtt{cl} ::= \mathtt{c}[\mathtt{e}] \mid \mathtt{cl}_1\ \mathtt{cl}_2 \mid \mathtt{if}\ \mathtt{cl}_1\ \mathtt{cl}_2\ \mathtt{cl}_3 \mid \ldots$$

where $\mathtt{e}$ is a map of substitutions: $\mathtt{e} ::= \star \mid \mathtt{e}\cdot(\mathtt{cl}/x)$.

Note that in this closure calculus a λ-term is simply a term, but its closure is a value. Note also that the explicit continuations $\mathtt{K}$ now also use closures $\mathtt{cl}$ instead of controls $\mathtt{c}$.

The new transition rules $\xrightarrow{L}{}^C$ differ from $\xrightarrow{L}{}^K$ in that their control element is a closure. This requires rules that propagate the map of substitutions $\mathtt{e}$ across sub-terms, such as, for example, in the following transition rule (Prop-IF) where $\mathtt{e}$ is propagated across the sub-terms of the term $\mathtt{if}$.

$$\langle (\mathtt{if}\ t_1\ t_2\ t_3)[\mathtt{e}], \mu, \mathtt{k}, \mathtt{N}, \overline{p} \rangle \xrightarrow{\tau} \langle (\mathtt{if}\ t_1[\mathtt{e}]\ t_2[\mathtt{e}]\ t_3[\mathtt{e}]), \mu, \mathtt{k}, \mathtt{N}, \overline{p} \rangle \quad \text{(Prop-IF)}$$

Having closures as controls of the transition system also requires us to update the reduction rules that use substitution. Function application, for example, is updated to make use of the map of substitutions by splitting it into two new rules M-B and M-V. These rules respectively add a new substitution to the map when performing an application and fetch a substitution from that same map when reducing a standalone variable.

$$\langle ((\lambda x : \tau.\mathtt{c})[\mathtt{e}]\ v[\mathtt{e}']), \mu, \mathtt{k}, \mathtt{N}, \overline{p} \rangle \xrightarrow{\tau} \langle \mathtt{c}[\mathtt{e} \cdot (\mathtt{v}[\mathtt{e}']/x)], \mu, \mathtt{k}, \mathtt{N}, \overline{p} \rangle \quad \text{(M-B)}$$

$$\langle x[\ldots(\mathtt{cl}/x)\ldots], \mu, \mathtt{k}, \mathtt{N}, \overline{p} \rangle \xrightarrow{\tau} \langle \mathtt{cl}, \mu, \mathtt{k}, \mathtt{N}, \overline{p} \rangle \quad \text{(M-V)}$$

Where $x[\ldots(\mathtt{cl}/x)\ldots]$, matches the first instance of x within the substitution map $\mathtt{e}$.

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 $[\![\cdot]\!]^C$:

$$[\![\langle \mathtt{c}, \mu, \mathtt{k}, \mathtt{N}, \overline{p} \rangle]\!]^C = \langle \mathtt{c}[\star], \mu, \mathtt{k}, \mathtt{N}, \overline{p} \rangle$$

compiles states $\mathtt{S}^K$ that are bisimilar in $\approx^K$ into states $\mathtt{S}^C$ that are bisimilar in $\approx^C$ as the additional transitions of LTS^C are silent transitions $\xrightarrow{\tau}$ that are ignored by the weak bisimulations.

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

$$\langle v[\mathtt{e}], \mu, \mathtt{k}[\circ\, \mathtt{K}[(\lambda x : \tau'.\mathtt{c})[\mathtt{e}']\ [\cdot]] : \tau], \mathtt{N}, \overline{p} \rangle \xrightarrow{\tau} \langle \mathtt{c}[\mathtt{e}' \cdot (v[\mathtt{e}]/\mathtt{x})], \mu, \mathtt{k}[\circ\, \mathtt{K} : \tau], \mathtt{N}, \overline{p} \rangle \quad \text{(M-B-O)}$$

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

[1] https://github.com/sylvarant/secure-abstract-machine

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

4. Unfolding Closures. The previously derived LTS^R is transformed into a CESK machine $(\texttt{M}, L, \xrightarrow{L}^M)$ where the state M is a quintuple $\langle \texttt{c}, \texttt{e}, \texttt{k}, \texttt{N}, \overline{p} \rangle$. 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 λ-terms: the application of closures cannot be unfolded. The new transition rules $\xrightarrow{L}^M$ are identical to $\xrightarrow{L}^R$ except for the cosmetically different states they relate.

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 $[\![\cdot]\!]^M$:

$$[\![\langle \texttt{c}[\texttt{e}], \mu, \texttt{k}, \texttt{N}, \overline{p} \rangle]\!]^M = \langle \texttt{c}, \texttt{e}, \mu, \texttt{k}, \texttt{N}, \overline{p} \rangle$$

compiles states $\texttt{S}^R$ that are bisimilar in $\approx^R$ into states M that are bisimilar in $\approx^M$ as the transitions are unmodified.

3.3 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 [12], as follows.

$$\begin{array}{lccc} \text{MiniML:} & t_1 & \simeq & t_2 \\ & & \Updownarrow \textbf{(P.2)} & \\ \text{MiniML} + FFI: & [\![t_1]\!]^{FFI} & \approx & [\![t_2]\!]^{FFI} \\ & & \Updownarrow \textbf{(P.3)} & \\ \text{CESK:} & [\![[\![t_1]\!]^{FFI}]\!]^{\text{CESK}} & \approx^M & [\![[\![t_2]\!]^{FFI}]\!]^{\text{CESK}} \end{array}$$

where P. is short for Proposition and where $[\![\cdot]\!]^{\text{CESK}}$ combines the compilation schemes of Section 3.2 as follows.

$$[\![\cdot]\!]^{\text{CESK}} = [\![[\![[\![\cdot]\!]^K]\!]^C]\!]^M$$

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 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 3.2, 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[2]) and deployed to the Fides implementation of

[2]https://github.com/sylvarant/secure-abstract-machine

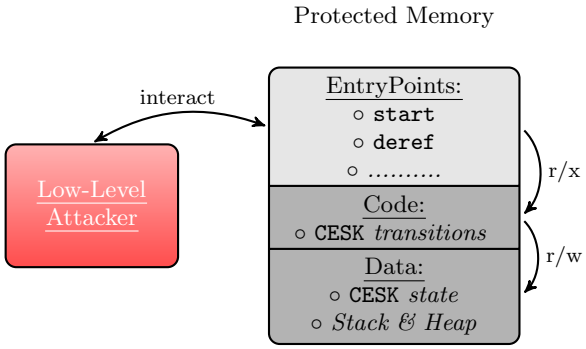

Figure 1: The CESK machine resides within the protected memory. The low-level attacker can only interact with the program running on the machine through the entry points.

PMA [15]. 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 [11].

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.

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.

	CESK	**CESK + PMA**
Application	0.28μs	16.71 μs
Callback	0.33μs	40.92μs
Read	0.24μs	16.88 μs

The tests were performed on a Dell Latitude with a 2.67 GHz Intel Core i5 and 4GB of DDR3 RAM. All scenario's incur high overheads due to the Fides hypervisor switching between VMs everytime the system transitions between secure and insecure memory. The *Callback* scenario incurs especially high overheads due to it requiring multiple transitions between the secure and insecure memory.

5. RELATED WORK

Our methodology as detailed in Section 3 improves upon the interoperation semantics of Larmuseau *et al.* [8] 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 [10] 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 [17]. 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 3.2) uses the syntactic correspondence of Biernacka and Danvy [2] between Curien's λ_p-cal-culus 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 [14]. An alternative approach to syntactic correspondence could be to adapt Ager *et al.*'s functional correspondence [1] between evaluators and abstract machines.

Our notions of bisimularity over the derived LTS's and `CESK` machine are based on the bisimulations for the νref-calculus by Jeffrey and Rathke [6]. 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 [5].

6. CONCLUSIONS AND FUTURE WORK

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 Larmuseau *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 it does not introduce security leaks.

References

[1] M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. In *PPDP '03*, pages 8–19. ACM.

[2] M. Biernacka and O. Danvy. A syntactic correspondence between context-sensitive calculi and abstract machines. *Theor. Comput. Sci.*, 375(1):76–108, 2007.

[3] O. Danvy and J. Johannsen. Inter-deriving semantic artifacts for object-oriented programming. *Journal of Computer and System Sciences*, 76(5):302 – 323, 2010.

[4] M. Felleisen. *The Calculi of Lambda-nu-cs Conversion: A Syntactic Theory of Control and State in Imperative Higher-order Programming Languages*. PhD thesis, Indiana University, 1987.

[5] C. Fournet, N. Swamy, J. Chen, P.-E. Dagand, P.-Y. Strub, and B. Livshits. Fully abstract compilation to javascript. In *POPL*, pages 371–384, 2013.

[6] A. Jeffrey and J. Rathke. Towards a theory of bisimulation for local names. Computer Science Report 02-2000, University of Sussex, 2000.

[7] P. Jones and S. L. Implementing lazy functional languages on stock hardware: The spineless tagless g-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.

[8] A. Larmuseau and D. Clarke. Formalizing a secure foreign function interface. In *SEFM 2015*, LNCS, pages 215–230. Springer.

[9] A. Larmuseau, M. Patrignani, and D. Clarke. Implementing a secure abstract machine – extended version. Technical Report 2015-034, Uppsala IT.

[10] J. Matthews and R. B. Findler. Operational semantics for multi-language programs. *TOPLAS*, 31(3), 2009.

[11] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP '13*. ACM, 2013.

[12] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. *TOPLAS*, 21(3), 1999.

[13] M. Patrignani and D. Clarke. Fully Abstract Trace Semantics of Low-level Isolation Mechanisms. In *SAC '14*, pages 1562–1569. ACM, 2014.

[14] M. Pirog and D. Biernacki. A systematic derivation of the stg machine verified in coq. In *Haskell '10*, pages 25–36. ACM, 2010.

[15] R. Strackx and F. Piessens. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *CCS*, 2012.

[16] G. Tan, S. Chakradhar, R. Srivaths, and R. D. Wang. Safe Java native interface. In *ESSoS*, 2006.

[17] S. Zdancewic, D. Grossman, and G. Morrisett. Principals in programming languages: a syntactic proof technique. In *ICFP '99*. ACM, 1999.