A Secure Compiler for ML Modules

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

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

Abstract. Many functional programming languages compile to low-level languages such as C or assembly. Numerous security properties of those compilers, however, apply only when the compiler compiles whole programs. This paper presents a compilation scheme that securely compiles a *standalone module* of ModuleML, a light-weight version of an ML with modules, into untyped assembly. The compilation scheme is secure in that it reflects the abstractions of a ModuleML module, for every possible piece of assembly code that it interacts with. This is achieved by isolating the compiled module through a low-level memory isolation mechanism and by dynamically type checking its interactions. We evaluate an implementation of the compiler on relevant test scenarios.

1 Introduction

High-level functional programming languages such as ML or Haskell offer programmers numerous security features through abstractions such as type systems, module systems and encapsulation primitives. Motivated by speed, memory efficiency and portability these high-level functional programming languages are often compiled to low-level target languages such as C and assembly [3]. The security features of such low-level target languages, however, rarely coincide with those of high-level source languages. As a result the compiled program might leak confidential information or break integrity when faced with an attacker operating in the low-level target language.

This security risk is rarely considered in existing compilers as it is often assumed that the compiler compiles the whole program, isolating it from malicious attackers. In practice, however, the final executable will consist of more than just the program in the functional language, it will be linked with various, low-level libraries and/or components that may be written with malicious intent or susceptible to code injection attacks. These low-level components have low-level code execution privileges enabling them to inject code into the system and inspect the variables and memory contents of the compiled program.

This paper presents a compilation scheme that compiles ModuleML, a light-weight version of ML featuring references and a module system, into an untyped assembly language running on a machine model enhanced with the Protected Module Architecture (PMA) [19]. PMA is a low-level memory isolation mechanism, that protects a certain memory area by restricting access to that area based on the location of the program counter. Our compilation scheme compiles an input ModuleML module to this protected memory in a way that protects it from low-level attackers while at the same time preserving all of its functionality.

Contributions The security of a compilation scheme between two programming languages, is often discussed in terms of full abstraction [1]. A fully-abstract compilation scheme preserves and reflects *contextual equivalence* between source and target-level components (Section 2). Preservation of contextual equivalence means that the compilation scheme outputs target-level components that behave as their source-level counterparts. Reflection implies that the source-level security properties are not violated by the generated target-level output.

This paper introduces a secure compilation scheme from ModuleML to untyped assembly extended with PMA (Section 3), that is proven to reflect contextual equivalence (Section 4). As is common in secure compilation works that target a realistic low-level language [16], we assume that preservation holds. Preservation coincides with compiler correctness, it establishes that the secure compiler is a correct ModuleML compiler. While we have tested our implementation intensely (Section 5), we consider formally verifying our compiler a separate research subject (Section 6). To better explain the secure compilation scheme, this paper also introduces a pattern referred to as the Secure Abstract Data Type pattern (Section 2.4). This pattern bundles together some of the techniques applied in previous secure compilation and full abstraction works.

This paper is not the first work to securely compile to untyped assembly extended with PMA. Previous work on secure compilation by Patrignani *et al.* [16] has fully abstractly compiled an object-oriented language to PMAs. The secure compilation scheme introduced in this paper differs from that work in the following three ways. Firstly, the secure compilation scheme of Patrignani *et al.* is limited in its usefulness as a real world compilation scheme in that it does not accept any arguments from the attacker outside of basic values, such as integers and booleans, and shared object identities. In this work we develop a more realistic compiler that accepts attacker defined functions, locations and modules.

Secondly, the abstractions of functional languages are more challenging than those of imperative object-oriented languages. In a functional language such as ModuleML, functions are for example higher-order and thus cannot be compiled into a straight-forward sequence of calls and returns. In this work we address these challenges through the use of an interaction counting masking mechanism.

Lastly, the inclusion of functors, higher-order functions mapping modules to modules, in ModuleML presents a novel secure compilation challenge. The modules created through functors are not analogous to objects, from a secure compilation standpoint. Whereas every object produced by a constructor is of the same type and thus subject to the same type checks and security constraints, functors can produce modules with different types and security constraints. In this work we address all security challenges introduced by functors and develop an efficient method of encoding the required checks.

Limitations To simplify the compilation scheme, polymorphic types and type kinds have been left out of ModuleML. The effects of certain low-level errors such as stack overflows or out of memory errors are also not considered.

2 Overview

This section introduces the source language ModuleML (Section 2.1), the target language A+I (Section 2.2), the threat model (Section 2.3) and a secure compilation pattern that we reuse throughout this work (Section 2.4).

2.1 The Source Language ModuleML

The source language ModuleML is divided into a core language and a module language. The core language is an extension of the simply typed λ-calculus featuring booleans, integers, unit, pairs, references, sequences, recursion and integer and boolean comparison operators. The module system is an adaption of Leroy's variant of the SML module system that features manifest types [11]. It consists of signatures, structures and functors, as illustrated below.

```
signature S = sig
 type T
 val func: T → T
end
```

Signature

```
module M : S = struct
 type T = int
 val func x = x +1
end
```

Structure

```
module F = functor(A : S)
struct
 val fd y = (A.func y)
end
module M' = F(M);
```

Functor

A signature is a sequence of signature components that are either value declarations type declarations or module declarations. The signature `S` listed above, for example, defines an abstract type `T` and a value declaration `func` that is a function of type `T` $\rightarrow$ `T`. A structure is a sequence of structure components that are either value bindings, module bindings or type bindings. The structure `M` listed above, binds the type `T` to **`int`** and binds the value `func` to a simple addition function. A functor can be considered as a parametrized module, a possibly higher-order function mapping modules to modules. The module `F` listed above, is a functor that maps a structure conforming to `S` to a new structure that consists only of a value binding `fd` that applies the value binding `A.func` to the argument `y`. The module `M'`, for example, is the result of applying `F` to `M`.

The typing rules for the ModuleML module system are standard. Note that this work uses SML style generative functors which return *fresh* abstract types with each application [4], as this type of functor provides strong data encapsulation. The interested reader can find a complete formalisation of ModuleML in the accompanying technical report [10].

Contextual Equivalence The secure compilation scheme aims to reflect ModuleML contextual equivalence in the target language A+I. A ModuleML context $C : \tau' \rightarrow \tau$ is a well-typed program P of type τ with a single hole $[\cdot]$ that is to be filled with a module `M` of type τ'. Two ModuleML modules $\mathtt{M}_1$ and $\mathtt{M}_2$ 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).

$$\mathtt{M}_1 \simeq \mathtt{M}_2 \stackrel{def}{=} \forall C : \tau' \to \tau.\ C[\mathtt{M}_1]\Uparrow \iff C[\mathtt{M}_2]\Uparrow$$

where $\Uparrow$ indicates divergence.

The following two ModuleML modules $\mathtt{M}_1$ and $\mathtt{M}_2$ are, for example, not contextually equivalent as they are distinguishable by the denoted context C, assuming Ω is a diverging term.

```
module M₁ = struct
  val v₁ = ref 1
end
```

Module A

```
module M₂ = struct
  val v₁ = ref 0
end
```

Module B

```
open M
(if (!(M.v₁) == 0) Ω
 else true)
```

Context C

Note that the `open M` statement implements the hole of the context C.

2.2 The Low-Level Target Language A+I

To model a realistic compilation scheme, the target language should be close to what is used by modern processors. For this reason this paper adopts A+I (acronym of Assembly plus Isolation), a low-level language that models an idealised von Neumann machine enhanced with a low-level memory protection mechanism referred to as Protected Module Architecture (PMA) [19]. 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 Point	*Code*	*Data*	
Protected	r x	r x	r w	r w x
Unprotected	x			r w x

A variety of PMA implementations exist. While most of them are research prototypes [19], Intel is developing a new instruction set, referred to as SGX, that enables the usage of PMA in commercial processors [15].

Trace Equivalence Our secure compiler relates contextually equivalent ModuleML modules to contextually equivalent low-level components. Reasoning about contexts is, however, notoriously complex. Reasoning about untyped low-level contexts is especially complex as they lack any inductive structure. In this work we thus adopt the fully abstract *trace semantics* of Patrignani and Clarke for PMA enhanced programs, to reason about trace equivalence instead [17].

The trace semantics transition over a state $\Lambda = (p, r, f, m, s)$, where 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, Λ can be $(\textsf{\textbf{unknown}}, m, s)$ a state modelling that A+I code, possibly malicious, is executing in unprotected memory. The trace semantics denote the observations of the A+I contexts that interact with the protected memory through labels L as follows.

$$L ::= \alpha \mid \tau \qquad\qquad \alpha ::= \surd \mid \delta! \mid \gamma?$$
$$\gamma ::= \mathit{call}\ p(r; f) \mid \mathit{ret}\ p(r; f) \qquad \delta ::= \gamma \mid \omega(a, v).\delta \qquad \omega ::= \mathit{read} \mid \mathit{write}$$

A label L can be either an observable action α or a non-observable action τ indicates that an unobservable action occurred in protected memory. Decorations ? and ! indicate the direction of the 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 the registers r and flags f. Registers and flags are in the labels as they convey information on the behaviour of the code executing in the protected memory. Observable actions $\omega(a, v)$ from the protected memory to the unprotected memory detail read and writes to the unprotected memory where a is the memory address and v is the value written to the address. The values will always be data, the compiler does not produce code that writes instructions to the unprotected memory. Additionally, an observable action α can be a tick $\surd$ indicating termination.

Formally the trace semantics of an A+I program L, denoted as $\mathsf{Traces}(L)$, are computed as follows: $\mathsf{Traces}(L) = \{\overline{\alpha} \mid \exists \Lambda.\Lambda_0(L) \stackrel{\overline{\alpha}}{\Longrightarrow} \Lambda\}$. Where Λ_0 is the initial state and the relation $\Lambda \stackrel{\overline{\alpha}}{\Longrightarrow} \Lambda'$ describes the traces generated by transitions between states. An important property of this trace equivalence is that the information they convey is so precise that we can rely on the equality between the traces produced by A+I programs as a replacement for contextual equivalence.

Proposition 1 (Fully Abstract Trace Semantics for A+I [17]).

$$L_1 \simeq_l L_2 \iff \mathsf{Traces}(L_1) = \mathsf{Traces}(L_2)$$

Where $\simeq_l$ denotes contextual equivalence between two A+I programs.

2.3 The Attacker

The attacker considered in this work has kernel-level code injection privileges that can be used to introduce malware into a software system. Kernel-level code injection is a critical vulnerability that bypasses all existing software-based security mechanisms: disclosing confidential data, disrupting applications and so forth. For the sake of simplicity, no differentiation between kernel and user code is defined in A+I. Thus, by modelling the attacker as injecting A+I code, we are modelling kernel-level code injection. Note that PMA is a program counter based mechanism that this attacker model cannot bypass [19].

2.4 The Secure Abstract Data Type Pattern

An A+I context must be able to perform the operations of ModuleML on the compiled ModuleML module. Each of these operations is different, but poses a similar secure compilation challenge: how do we enable the A+I context to perform the relevant operations without exposing the implementation details of the abstraction? In this work we introduce the Secure Abstract Data Type (ADT) pattern as a general approach to addressing this challenge. This pattern bundles together the individual techniques applied in certain secure compilation [16] and full abstraction results [14].

An ADT defines both the values of a data type as well as the functions that apply to it, relying on static typing rules to hide the implementation details of the data type. The Secure ADT pattern, in contrast, protects the implementation details of a source language abstraction τ *without* relying on static typing rules. As illustrated in Figure 1, it does this by inserting an ADT like interface between the actual implementation of the abstraction and the target language context. Concretely a secure ADT has the following elements: a secured type $\mathsf{Sec}[\tau]$, an interface that defines the operations applicable to the protected type, marshalling rules that handle the transitions between the different representations for τ, and additional run-time checks if required.

Secured type The Secure ADT pattern states that values of the type τ, the type of the abstraction that the Secure ADT aims to secure, must be *isolated* and can thus not be shared directly. Instead they can be, for example, shared securely by encrypting the value or by providing a reference object, an object that refers to the original value. The type of these securely shared instances is denoted as $\mathsf{Sec}[\tau]$. The Secure ADT pattern considers not only the secure sharing of values of type τ, but also input from the target language context. This input is denoted as $\mathsf{Ins}[\tau']$, where τ' denotes the source language type that the input is expected to conform to. We use τ' and not τ as the outside input can be of a different type then the abstraction that the secure ADT pattern secures.

Interface As illustrated in Figure 1, the interface defines a series of functions (v_i) that provide the outside context with the functionality of ModuleML. These

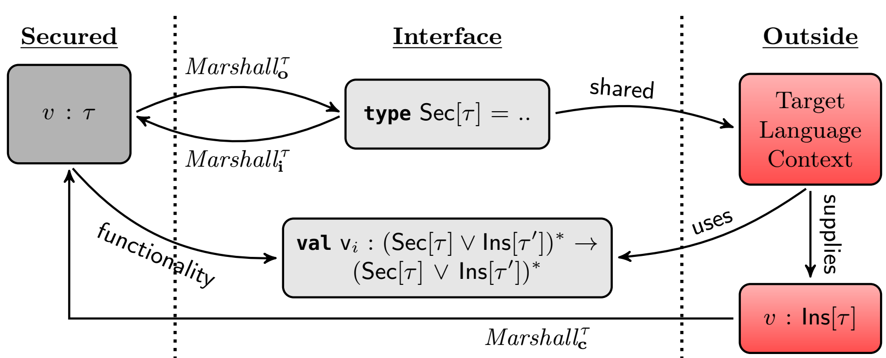

Fig. 1: The Secure ADT pattern isolates an abstraction of type τ through an ADT-like interface that shares secured instances of τ ($\mathsf{Sec}[\tau]$) and accepts outside input ($\mathsf{Ins}[\tau']$).

functions take as arguments some sequence of securely shared values and target language input and return a securely shared or a target language value.

Marshalling The Secure ADT pattern introduces *type directed* marshalling functions to handle the transitions between the values of type τ, which are the securely compiled values, the values of type $\mathsf{Sec}[\tau]$, which are the securely shared instances, and the values of type $\mathsf{Ins}[\tau]$ which are defined by the outside context. The function $\mathit{Marshall}^{\tau}_{\mathbf{o}} : \tau \rightarrow \mathsf{Sec}[\tau]$ converts values into their secured instances. The function $\mathit{Marshall}^{\tau}_{\mathbf{i}} : \mathsf{Sec}[\tau] \rightarrow \tau$ converts the secured instances back into the original value. Note that this function performs an implicit run-time type check. It fails when given an input that does not correspond to a securely shared value of type τ. Certain secure compilation schemes, such as the one considered in this work, also specify a third type of marshalling function: $\mathit{Marshall}^{\tau}_{\mathbf{c}} : \mathsf{Ins}[\tau] \rightarrow \tau$. Such a marshalling function converts values from the target language context into values of the secured type τ, by converting the input value into the correct representation and by wrapping the result with type checks. Note that if the input is of type $\mathsf{Ins}[\tau']$, where $\tau' \neq \tau$, then the input will only be marshalled in if there exists a marshalling function: $\mathit{Marshall}^{\tau'}_{\mathbf{c}} : \mathsf{Ins}[\tau'] \rightarrow \tau'$.

Run-time checks The marshalling rules verify that the input provided by the outside target language context and the output shared to the outside context conform to the typing rules of the source language. This, however, is sometimes not enough to protect the abstractions of the source language. Certain security relevant language properties such as, for example: control-flow integrity, are not always explicitly captured by the typing system. Enforcing these properties must thus be done through *additional* run-time security checks.

3 A Secure Compiler for ModuleML

The secure compilation scheme for ModuleML is a type directed compilation scheme that compiles a standalone ModuleML module and its signature to a protected module (Figure 2). The secure compilation scheme applies the Secure ADT pattern in a general manner. The entry points of the protected module implement an ADT-like interface to the A+I context. The abstractions of ModuleML are isolated by placing all code and data into the data and code sections of the protected module. The protected data section also includes a heap and stack of a fixed size, that can only be accessed by the securely compiled program. This ensures that the run-time memory of the compiled program is also inaccessible.

The inner workings of how ModuleML is compiled to assembly is of little relevance to this result of this paper. Instead this section focusses on the security relevant aspects of the compilation scheme. This section details how we apply the Secure ADT pattern of Section 2.4 to securely compile abstract types (Section 3.2), structures and signatures (Section 3.3), functions (Section 3.4), locations (Section 3.5) and functors (Section 3.6). Basic types such as integers or pairs are not compiled using the Secure ADT pattern, but must still be marshalled when interacting with the A+I context (Section 3.1).

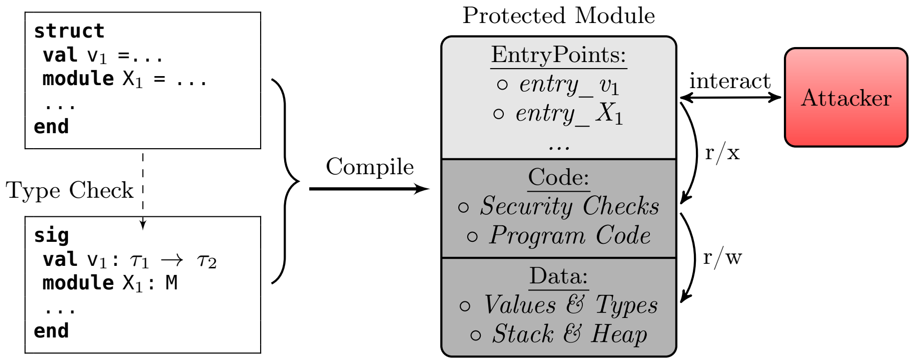

Fig. 2: Our scheme compiles the module and its type into the protected memory.

3.1 Booleans, Integers and pairs

The securely compiled module shares and inputs not only abstractions such as functions, but also basic ModuleML values: booleans, integers and pairs. Booleans and integers are exchanged with the A+I context using their respective A+I representation. The marshalling functions for integers are thus defined as $Marshall_{\mathbf{c}}^{\mathbf{int}} : \mathsf{Ins}[\mathbf{int}] \to \mathbf{int}$, which converts A+I integers into ModuleML integers, and $Marshall_{\mathbf{o}}^{\mathbf{int}} : \mathbf{int} \to \mathsf{Ins}[\mathbf{int}]$, which converts ModuleML integers to A+I integers. The marshalling functions for booleans are analogous.

Marshalling pairs is different. When marshalling, for example, a pair $\langle v_1, v_2\rangle$ the marshalling functions for pairs marshall each value with the type appropriate marshalling function as dictated by the Secure ADT pattern. Marshalling *out* the pair $\langle 1, 2\rangle$, for example, will thus produce a value of type $\langle \mathsf{Ins}[\mathbf{int}], \mathsf{Ins}[\mathbf{int}]\rangle$, while marshalling out the pair of lambdas $\langle(\lambda x : \tau.t), (\lambda x : \tau.t')\rangle$ will produce a value of type $\langle \mathsf{Sec}[\tau \to \tau'], \mathsf{Sec}[\tau \to \tau'']\rangle$.

3.2 Abstract types

Abstract types are, as the name indicates, abstract in that associated values are unobservable to an ModuleML context. Consider, for example, the following module A that conforms to the signature S. This signature defines an abstract type T that abstracts the value bindings v_1 and v_2.

```
module A : S = struct
 type T = bool
 val v1 = true
 val v2 = v1
end
```

```
signature S = sig
 type T
 val v1 : T
 val v2 : T
end
```

An A+I context should not be able to observe that $A.v_1$ and $A.v_2$ both return the value `true`. To achieve this our compilation scheme applies the Secure ADT pattern to compile values of an abstract type. Instead of directly sharing the value of an abstract type T with the $\mathsf{A+I}$ context, we share a secured instance of type $\mathsf{Sec}[\mathsf{T}]$ instead. These secured instances are implemented as indices to a table $\mathcal{A}$. This table $\mathcal{A}$ maps natural numbers to values and their types in a deterministic manner, simply denumerating its entries. Note that this map is *not* a set: it may map different numbers to duplicate elements.

As illustrated in Figure 3, every time a value of an abstract type is returned the securely compiled module will share a new index i that corresponds to the number of requests that the $\mathsf{A+I}$ context has made to abstract types. Note that each member of a pair (Section 3.1) counts as a separate request. The marshalling functions $\mathit{Marshall}_{\mathbf{o}}^{\mathsf{T}}$ and $\mathit{Marshall}_{\mathbf{i}}^{\mathsf{T}}$ are thus implemented as extending the table $\mathcal{A}$ and looking up an index in $\mathcal{A}$ respectively, as illustrated in Figure 3.

We have formally proven in prior work [9], by means of a full abstraction proof, that these request counting indices do not reveal any information to the $\mathsf{A+I}$ context other than the number of times the $\mathsf{A+I}$ context has requested a value of an abstract type. This is information that the context of any source language with state can reproduce and thus does not harm full abstraction. In the case of ModuleML, a context can count its interactions with the protected module by making use of references (a detail that returns in our proof of Section 4).

3.3 Structures and Signatures

Our compiler compiles both structures and signatures into records stored within the data section of the protected memory. As dictated by the Secure ADT pattern these records are not exposed directly to the $\mathsf{A+I}$ context. Instead the compilation scheme defines an ADT-like interface of entry points to the protected memory that provide access to the value and structure bindings exposed by the module's signature. Note that, as in previous works [16], these entry points are sorted to obscure their implementation order. The compiler also includes a *load entry point* that evaluates each of the expressions defined within a structure. Our compilation scheme defines marshalling rules that both share secure structures as well as convert in structures created by the $\mathsf{A+I}$ context.

Load entry point As is the case in most ML implementations, the value bindings of ModuleML map names to *expressions* not values. These expressions must be

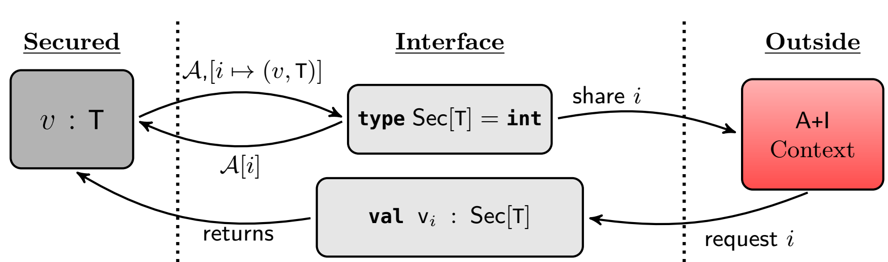

Fig. 3: We use request counting to obscure the value of an abstract type.

reduced to values before the value bindings of a structure can be queried. Our compiler, however, compiles a standalone ModuleML module not a full program, it thus does not have any control over when or if the expressions are evaluated. Instead our compilation scheme provides the A+I context the ability to load the module through a load entry point. This entry point takes no arguments and executes each of the expressions defined throughout the compiled module, storing the result in the appropriate record. Because it is up to the low-level context to invoke this load entry point, a malicious A+I context may attempt to query bindings before the module is loaded or attempt to load the module multiple times. To prevent this, the compiler introduces an additional run-time check in the form of a global flag L_f, that encodes whether or not the module has been loaded. What follows is a pseudo code implementation of the load entry point.

1. Check the flag L_f. Abort if set.
2. For each value binding v_i with an associated expression e:
   (a) Evaluate e and store the result in the appropriate record.
3. Set L_f.

Value binding entry points For each value binding v_i reduced to a value $v : \tau$ declared within the signature of a structure, the compilation scheme creates an entry point of type: $\mathsf{v}_i : \mathsf{Sec}[\tau]$, if τ is an abstraction that must be secured, or $\mathsf{v}_i : \mathsf{Ins}[\tau]$ if τ is a basic type such as **int**. Both are implemented as follows.

1. Check the flag L_f. Abort if not set.
2. Fetch the value v and its type τ from the data section.
3. Return $\mathit{Marshall}^{\tau}_{\mathbf{o}}(v)$

Entry points to structures For each module binding M_i to a structure s with signature S that is declared within the signature of the outer structure, our compiler creates an entry point of type: $\mathsf{M}_i : \mathsf{Sec}[\mathsf{S}]$ that takes no arguments and returns a marshalled instance of the structure of secured type $\mathsf{Sec}[\mathsf{S}]$, as follows.

1. Check the flag L_f. Abort if not set.
2. Return $\mathit{Marshall}^{\mathsf{S}}_{\mathbf{o}}(s)$

$\mathit{Marshall}^{\mathsf{S}}_{\mathbf{o}}$ and $\mathit{Marshall}^{\mathsf{S}}_{\mathbf{i}}$ As dictated by the Secure ADT pattern, structures are not shared directly but instead marshalled out using a type directed function $\mathit{Marshall}^{\mathsf{S}}_{\mathbf{o}} : \mathsf{S} \to \mathsf{Sec}[\mathsf{S}]$. This function converts a structure of signature S into a secured instance $\mathsf{Sec}[\mathsf{S}]$: a record that contains an index i to the table $\mathcal{M}$ and references to the entry points of each value/module binding in S. The references to the entry points are included to inform the A+I context of the functionality that the structure provides, simplifying interoperation. Like the table $\mathcal{A}$ of Section 3.2, the table $\mathcal{M}$ maps numbers to structures and their signatures. This index i thus enables the marshalling in function $\mathit{Marshall}^{\mathsf{S}}_{\mathbf{i}} : \mathsf{Sec}[\mathsf{S}] \to \mathsf{S}$ to retrieve the original

structure and its signature from $\mathcal{M}$. Note that this marshalling function thus performs an implicit type check as the function fails whenever the retrieved signature is not a *subtype* of S.

$\mathit{Marshall}^{\mathsf{S}}_{\mathbf{c}}$ Our compilation scheme enables the A+I context to supply its own structures as arguments to the functors of Section 3.6. These structures are marshalled in by a function $\mathit{Marshall}^{\mathsf{S}}_{\mathbf{c}} : \mathsf{Ins}[\mathsf{S}] \rightarrow \mathsf{S}$, that iterates through the components of the expected signature S, querying the A+I context's structure for the names of the bindings, marshalling in the results or aborting if a name isn't found. When a value binding is marshalled in it is marshalled in using the type appropriate function. When a module binding is marshalled the marshalling function recurses. Note that this function performs a sub-type check: $\mathsf{Ins}[\mathsf{S}] <: \mathsf{S}$.

3.4 Higher-Order Functions

To compile the λ-terms of ModuleML the compiler uses closure conversion [18] to eliminate free variables by using an explicit environment that stores bindings between variables and values. As is required by the Secure ADT pattern, these closures are not made available to the A+I context but are instead shared as secured instance of type $\mathsf{Sec}[\tau_1 \rightarrow \tau_2]$: indices to a table $\mathcal{C}$ that maps numbers to closures and their types. As was the case for the indices of Section 3.2, these numbers simply denumerate the requests made by the A+I context. The marshalling functions $\mathit{Marshall}^{\tau_1 \rightarrow \tau_2}_{\mathbf{o}}$ and $\mathit{Marshall}^{\tau_1 \rightarrow \tau_2}_{\mathbf{i}}$ are thus implemented as extending the table $\mathcal{C}$ and looking up the closure and its type in $\mathcal{C}$ respectively.

Closure application entry point As is required by the Secure ADT pattern we enable the A+I context to apply shared closures through an entry point of type: $\mathtt{appl} : \mathsf{Sec}[\tau_1 \rightarrow \tau_2] \rightarrow (\mathsf{Ins}[\tau_1] \vee \mathsf{Sec}[\tau_1]) \rightarrow (\mathsf{Ins}[\tau_2] \vee \mathsf{Sec}[\tau_2])$, where the result is $\mathsf{Ins}[\tau_2]$ if τ_2 is a basic type and $\mathsf{Sec}[\tau_2]$ otherwise. This entry point takes as its arguments an index i to the table $\mathcal{C}$ and as a value v of the appropriate representation for type τ_1. The entry point is implemented as follows.

1. Check the flag L_f. Abort if not set.
2. $c = \mathit{Marshall}^{\tau_1 \rightarrow \tau_2}_{\mathbf{i}}(i)$
3. Depending on the representation of v:
   (a) If $\mathsf{Ins}[\tau_1]$: $r = \mathit{Marshall}^{\tau_1}_{\mathbf{c}}(v)$
   (b) If $\mathsf{Sec}[\tau_1]$: $r = \mathit{Marshall}^{\tau_1}_{\mathbf{i}}(v)$
4. Apply c to v, store the result in r'.
5. Return $\mathit{Marshall}^{\tau_2}_{\mathbf{o}}(r')$

Note that the marshalling rules of 3(a) and 3(b) implement the typing rule for function applications, by ensuring that the input value v is of type τ_1.

$\mathit{Marshall}^{\tau_1 \rightarrow \tau_2}_{\mathbf{c}}$ Our compilation scheme enables the A+I context to supply its own functions as arguments to the securely compiled entry points that accept an argument of type: $\mathsf{Ins}[\tau_1 \rightarrow \tau_2]$. These A+I functions are marshalled by a function $\mathit{Marshall}_{\mathbf{c}}^{\tau_1 \to \tau_2} : \mathsf{Ins}[\tau_1 \to \tau_2] \to (\tau_1 \to \tau_2)$, that takes in a reference to the A+I function f and wraps that function into a new function that performs the following steps, whenever the A+I function f is applied to a ModuleML value v within the securely compiled module.

1. $a = \mathit{Marshall}_{\mathbf{o}}^{\tau_1}(v)$
2. Apply f to a. Store the result in r.
3. Return $\mathit{Marshall}_{\mathbf{c}}^{\tau_2}(r)$

3.5 Locations

As is the case in most commonly used ML variants [13], memory locations do not explicitly appear in the syntax used by programmers. Locations are thus not directly observable to an ModuleML context, leading to many equivalences. Consider, for example, the following two contextually equivalent implementations of the value binding v_1.

```
val v₁ = (let x = (ref true) in
 let y = (ref true) in y)
```

```
val v₁ = (let x = (ref true) in
 let y = (ref true) in x)
```

No ModuleML context can observe that the left implementation differs from the right implementation in that it returns the second location it created, stored within variable y, and not the first location stored within the variable x.

Again our compilation scheme applies the Secure ADT pattern to protect ModuleML's locations and the operations available on them. Locations are shared with the A+I context in the same manner as higher-order functions (Section 3.4) and abstract types (Section 3.2): as indices into a table $\mathcal{L}$ that maps numbers to locations and their types. As was the case previously, these numbers simply denumerate the requests made by the A+I context for access to ModuleML locations. The marshalling functions $\mathit{Marshall}_{\mathbf{o}}^{\mathbf{ref}\ \tau}$ and $\mathit{Marshall}_{\mathbf{i}}^{\mathbf{ref}\ \tau}$ are thus implemented as extending the table $\mathcal{L}$ and looking up an index in $\mathcal{L}$ respectively.

Write and read entry points To enable the low-level A+I context to write and read to shared locations in the same way that an ModuleML context can, we introduce a *write location* entry point of type: `write` : $\mathsf{Sec}[\mathbf{ref}\ \tau] \to (\mathsf{Ins}[\tau] \vee \mathsf{Sec}[\tau]) \to \mathbf{unit}$, and a *read location* entry point of type `read` : $\mathsf{Sec}[\mathbf{ref}\ \tau] \to (\mathsf{Ins}[\tau] \vee \mathsf{Sec}[\tau])$. The write location entry points takes two arguments: an index i to the table $\mathcal{L}$ and a value v of the appropriate representation for type τ. It securely writes v to the appropriate location, as follows.

1. Check the flag L_f. Abort if not set.
2. $l = \mathit{Marshall}_{\mathbf{i}}^{\mathbf{ref}\ \tau}(i)$.
3. Depending on the representation of v:
   (a) If $\mathsf{Ins}[\tau]$: $r = \mathit{Marshall}_{\mathbf{c}}^{\tau}(v)$
   (b) If $\mathsf{Sec}[\tau]$: $r = \mathit{Marshall}_{\mathbf{i}}^{\tau}(v)$
4. Write r to l.

Note again, that the marshalling rules 3(a) and 3(b) implement the assign location typing rule, by ensuring that the input value v is of type τ.

The implementation of the read location entry point is straight-forward: it retrieves the location from $\mathcal{L}$, dereferences it and marshalls the value.

$Marshall_{\mathbf{c}}^{\mathsf{ref}\,\tau}$ A ModuleML context can allocate new locations and share them with the ModuleML module embedded within the context's hole. We thus enable the A+I context to supply its own locations as arguments to entry points that accept an argument of type $\mathsf{Ins}[\mathbf{ref}\,\tau_1]$. As specified by the Secure ADT pattern these locations are marshalled by a function $Marshall_{\mathbf{c}}^{\mathbf{ref}\,\tau} : \mathsf{Ins}[\mathbf{ref}\,\tau] \to \mathbf{ref}\,\tau$, that takes in a location l_f of the A+I context and wraps it with two functions. The first function enables a ModuleML expression to read the foreign location, the second function enables an ModuleML expression to write to the foreign location. The implementation of the latter is analogous to the implementation of the write entry point. The implementation of the former is simply: $Marshall_{\mathbf{c}}^{\tau}(!l_f)$, where $!l_f$ denotes the dereference of the A+I location l_f.

3.6 Functors

As noted earlier, a ModuleML functor is a higher-order function that maps modules (structures or functors) to modules. Consider the following example.

```
signature Sa = sig
 type U
 val v1: int →int
 val vs : U
end
```

```
signature Sr = sig
 type T
 val fd: int → int
 val F1:functor(X:Sa)→Sa
 val M1: sig
  val v1: T
 end
end
```

```
module F = functor(A : Sa)
struct
 type T = int
 val fd y = (A.v1 y)
 module F1 = functor(X:Sa)=A
 module M1 = A
end : Sr
module M' = F(Mi)
```

Module F is a functor that maps a structure that conforms to signature S_a, to a new structure that consists of: a value binding fd, that applies the argument's value binding v_1 to an argument y, and an inner functor F_1 and an inner structure M_1 that copies the argument. This new structure is ascribed with the signature S_r which seals the value binding $\mathsf{M}_1.\mathsf{v}_1$ with the abstract type T. When compiling functors the compiler operates in two modes. The first mode considers the *static* functor applications within the compiled module, such as, for example, the application of F to an example module M_i in the above listing. Compiling these applications is straightforward, the compiler performs the application and compiles the result in the same way that it compiles any other module.

The second mode considers those functors that are part of the interface to the A+I context. In this case we must securely compile functors into run-time constructs. As is dictated by the Secure ADT pattern we do not share these run-time representations directly with the A+I context, but instead share them (again) as indices into a table $\mathcal{F}$ that maps numbers to functors and their types. As was the case previously, these numbers simply denumerate the requests made by the A+I context for access to ModuleML functors. The marshalling functions

```
module F: functor (A : Sa) →
sig
 type T
 val fd: int → int
 module F1: functor(X : S) →(sig
  type U
  val v1: Int
  val vs: U
 end)
 module M1: (sig
  val v1: T end)
end
```

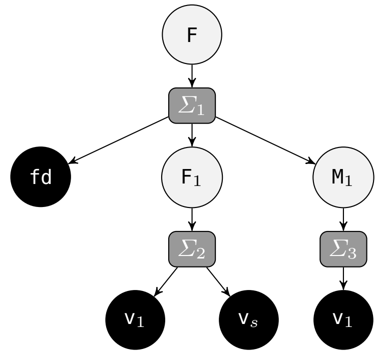

Fig. 4: The secure compiler compiles the signature of F into a tree of unique stamps Σ_i, that enable the functor entry points to identify their arguments.

$Marshall_{\mathbf{o}}^{\mathbf{functor}(\mathsf{X}_i:\mathsf{S})\to\mathsf{S}'}$ and $Marshall_{\mathbf{i}}^{\mathbf{functor}(\mathsf{X}_i:\mathsf{S})\to\mathsf{S}'}$, where $\mathbf{functor}(\mathsf{X}_i : \mathsf{S}) \to \mathsf{S}'$ is the expected type of the functor, are thus implemented by extending the table $\mathcal{F}$ and looking up an index in $\mathcal{F}$ and confirming the type respectively. Our compilation scheme also provides a marshalling rule $Marshall_{\mathbf{c}}^{\mathbf{functor}\,(\mathsf{X}_i:\mathsf{S})\to\mathsf{S}'}$ that converts structures of the A+I context.

Compiling run-time functors Functors are compiled into run-time constructs in a manner similar to the way in which λ-terms are compiled to closures. The functor body is compiled into a function that takes as its arguments a module and an environment of module bindings and returns a new module that conforms to the specification of the functor body. In addition to being compiled into a function, every functor is also compiled into a tree structure of the accessible bindings, assigning a unique stamp Σ_i to each non-leaf node (Figure 4). These stamps Σ_i are used by the entry points for these bindings to authenticate its arguments.

The module that results from applying a run-time functor is stored as a record that incorporates the resulting module as well as additional run-time data. Additionally the record stores a stamp Σ_i, that identifies the functor that produced it, a module binding environment e, which includes the argument to the functor, and environment of abstract type identifiers e_t. The latter is required to keep track of the abstract types that are created by functors that seal their result, as they generate a new abstract type each time they are applied.

Functor application entry point To enable the low-level A+I context to apply functors to modules in the same way that a ModuleML context can, we introduce a functor application entry point into the protected memory that has type: $\mathtt{fappl} : \mathsf{Sec}[\mathbf{functor}(\mathsf{X}_i : \mathsf{S}) \to \mathsf{S}'] \to (\mathsf{Ins}[\mathsf{S}] \vee \mathsf{Sec}[S]) \to \mathsf{Sec}[\mathsf{S}']$. The first argument to this entry point is an index to the table $\mathcal{F}$, the second argument m is a shared module or a module defined by the A+I context. The entry point securely applies the appropriate functor f with associated stamp Σ_f to the argument a, as long as a conforms to the signature S, as follows.

1. Check the flag L_f. Abort if not set.
2. $f = \mathit{Marshall}_{\mathbf{i}}^{\textbf{functor}(\mathsf{X}_i:\mathsf{S})\to\mathsf{S}'}(i)$.
3. Depending on the representation of m:
   (a) If $\mathsf{Ins}[\mathsf{S}]$: $a = \mathit{Marshall}_{\mathbf{c}}^{\mathsf{S}}(a)$
   (b) If $\mathsf{Sec}[\mathsf{S}]$: $a = \mathit{Marshall}_{\mathbf{i}}^{\mathsf{S}}(a)$
4. Apply f to a. Store the result in r.
5. Stamp r with Σ_f.
6. Return $\mathit{Marshall}_{\mathbf{o}}^{S'}(r)$.

Note that as specified in Section 3.3, the marshalling rules of 3(a) and 3(b) perform the sub-typing check required by the functor application rule.

Functor entry points The secure compilation scheme outputs entry points that enable the A+I context to gain access to the functor as well as interact with the result of the functor application. The entry points to functor bindings that are not embedded within another functor have a type: $\mathsf{M}_i : \mathsf{Sec}[\textbf{functor}(\mathsf{X}_i : \mathsf{S}) \to \mathsf{S}']$ and marshall out the associated functor through an index to a table $\mathcal{F}$.

The entry points to the bindings of structures that are defined within the body of a functor, differ from the previously detailed entry points for value, structure and functor bindings in that they take an argument: an index i to the table $\mathcal{M}$. As detailed in the previous paragraph, the functor application entry point marshalls out its result through the marshalling function $\mathit{Marshall}_{\mathbf{o}}^{S}$, which, as explained in Section 3.3, stores the result into the structure requests counting table $\mathcal{M}$. The implementations of these entry points extend the previously discussed entry point implementations in that their result is not statically defined but depends on the structure associated with the input index i. The entry points will thus look up index i in $\mathcal{M}$ and check that the retrieved structure is stamped with the correct stamp Σ_i, as follows.

1. $d = \mathit{Marshall}_{\mathbf{i}}^{\mathsf{S}}(i)$.
2. Check that stamp of $d = \Sigma_i$. If not Abort.

To illustrate the necessity of this stamp check, we reconsider the example functor F introduced at the beginning of this section. This functor is assigned the stamp Σ_1 (Figure 4) and each of its bindings F.fd, F.F$_1$ and F.M$_1$, check that the structure associated with input index i is stamped by Σ_1. If they did not do so the A+I context could, for example, violate the typing rules of ModuleML by passing a structure created using F to the bindings of the following functor F$_b$.

```
signature S_b = sig
 type U
 val v_1: int → int
 val v_s: int
end
```

```
module F_b = functor (A : S_b) struct
 type T = int
 val fd y = (A.v1 y)
 module F_1 = functor(X:S_a)struct type U = int
  val v_s = 0; val v_1 = A.v_s end
 module M_1 = A : S_a
end : S_r
```

While both F_b and F produce a structure with signature S_r, the argument of F_b conforms to the signature S_b not the signature S_a, which seals the binding v_s whereas S_b does not. Without the stamp checking mechanism the A+I context could break the abstractions of ModuleML by passing a module produced by applying F to the entry point for $\mathsf{F}_b.\mathsf{F}_1$ as the implementation of $\mathsf{F}_b.\mathsf{F}_1$ exposes the value binding $\mathsf{A.v}_s$, as highlighted in gray in the listing for F_b.

The entry points for $\mathsf{F.F}_1$ and $\mathsf{F.M}_1$ stamp their result with a stamp Σ_2 and Σ_3 respectively. This further specialization of the stamps within the inner modules is necessary to prevent similar attacks.

$\mathit{Marshall}_{\mathbf{c}}^{\mathbf{functor}(\mathsf{X}_i:\mathsf{S})\to\mathsf{S}'}$ Our compilation scheme enables the A+I context to supply its own functors as arguments to the functor application entry point. These foreign functors are marshalled into ModuleML functors by a function $\mathit{Marshall}_{\mathbf{c}}^{\mathbf{functor}(\mathsf{X}_i:\mathsf{S})\to\mathsf{S}'} : \mathsf{Ins}[\mathbf{functor}(\mathsf{X}_i : \mathsf{S}) \to \mathsf{S}'] \to (\mathbf{functor}(\mathsf{X}_i : \mathsf{S}) \to \mathsf{S}')$, that takes in a reference to an A+I function f and wraps that function into a new function that performs the following steps, whenever the foreign functor is applied to a ModuleML module M, within the securely compiled module.

1. $a = \mathit{Marshall}_{\mathbf{o}}^{\mathsf{S}}(\mathsf{M})$
2. Apply f to a. Store the result in r.
3. Return $\mathit{Marshall}_{\mathbf{c}}^{\mathsf{S}'}(r)$

4 Compiler Reflection

Denote the result of compiling the module M down to A+I as $\mathsf{M}^{\downarrow}$. Compiler reflection is formally expressed as.

$$\mathsf{M}_1 \simeq \mathsf{M}_2 \Rightarrow \mathsf{M}_1^{\downarrow} \simeq \mathsf{M}_2^{\downarrow}$$

It states that the equivalences of the modules M_1 and M_2 are preserved through the secure compilation scheme in the A+I context. To prove this statement we will prove the contra-positive: $\mathsf{M}_1^{\downarrow} \not\simeq \mathsf{M}_2^{\downarrow} \Rightarrow \mathsf{M}_1 \not\simeq \mathsf{M}_2$. This contra-positive can be stated as: whenever an A+I context can distinguish between two compiled modules, there exists a ModuleML context that can distinguish between the original modules. As detailed in Section 2.2 we do not directly reason about contextual equivalence for A+I programs but instead rely on trace equivalence. As such we can redefine compiler reflection as follows.

Theorem 1 (Module Differentation). *Any two ModuleML modules M_1 and M_2 whose compilation results produce two different low-level traces $\overline{\gamma_1}$ and $\overline{\gamma_2}$ are not contextually equivalent. Formally:* $\mathsf{Traces}(\mathsf{M}_1{}^{\downarrow}) \neq \mathsf{Traces}(\mathsf{M}_2{}^{\downarrow}) \Rightarrow \mathsf{M}_1 \not\simeq \mathsf{M}_2$.

To prove the theorem we adopt the established proof technique [8,16] of developing an algorithm that given two ModuleML modules M_1 and M_2 and their differing

A+I traces $\overline{\gamma_1}$ and $\overline{\gamma_2}$ can produce a "witness" ModuleML context C that can distinguish between $\mathtt{M}_1$ and $\mathtt{M}_2$.

We have implemented exactly such a witness building algorithm in Ocaml[3]. The algorithm analyses the labels of the low-level traces $\overline{\gamma_1}$ and $\overline{\gamma_2}$ that detail the interactions between an unknown A+I context (it's a black box) and the modules $\mathtt{M}_1$ and $\mathtt{M}_2$. For the algorithm to be correct, it must detect the first two labels γ in the traces that differ. Assuming the first differening labels are at position i, the algorithm produces an ModuleML module that will replicate the first $i - 1$ labels of the traces and at the i-th step will diverge for $\mathtt{M}_1$ and terminate for $\mathtt{M}_2$, distinguishing them as required. The resulting module must thus keep track of the number of interactions it has with the unknown A+I context, which is done through the use of the ModuleML locations. A full explanation of the inner workings of the algorithm is provided in the accompanying technical report [10].

5 Implementation and Experimental Results

We have developped a compiler[4]. that compiles ModuleML modules using either the secure compilation scheme detailed in this paper or through a naive and *insecure* compilation scheme that features none of the security checks. The compiler targets the Fides implementation of PMA [19]. 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 compiled module, the Fides hypervisor will be forced to switch between the two virtual machines for each call and callback between the context and the module.

The security checks described in this paper are only triggered when execution crosses the boundary between protected and unprotected memory. As such we benchmark five scenarios (included with the source code of the compiler) that involve boundary crossings. In the first scenario (*Value*) the A+I context retrieves a value binding by calling the appropriate entry point. In the second scenario (*Closure Application*) the A+I context applies a secure closure to another secure closure using the closure application entry point. In the third scenario (*Callback*) the atacker applies a secure closure to a function of the A+I context. In the next scenario (*Functor Application*) the A+I context applies a functor to a module of the A+I context using the functor application entry point. In the final scenario (*Dynamic Value*) the A+I context accesses the value binding of a structure that results from applying a functor at run-time. We have timed the performance of each of these five scenarios, as denoted in Table 1.

The tests were performed on a Dell Latitude with a 2.67 GHz Intel Core i5 and 4GB of DDR3 RAM. The difference between rows "Insecure" and "Insecure + Fides" shows the high overhead of the Fides architecture. It is especially notable in the call back and functor application scenarios which transition between the protected and unprotected memory twice. The security checks of the functor

[3] https://github.com/sylvarant/moduleml-witness-algorithm
[4] https://github.com/sylvarant/secure-ml-compiler

	Insecure	**Insecure + Fides**	**Secure + Fides**
Value Binding	0.18μs	17.59μs	17.86μs
Closure Application	0.32μs	17.68 μs	18.09μs
Callback	0.31μs	36.59μs	36.97μs
Functor Application	0.57μs	37.14μs	106.50μs
Dynamic Value Binding	0.26μs	17.73μs	18.41μs

Table 1: The average execution time for each test scenario.

application scenario have by far the biggest performance impact. This is due to the fact that this scenario involves both the dynamic type checking of the structure input by the A+I context as well as the creation of a new module, two computationally intensive operations. The additional performance impact of the security checks in the other scenarios is small, peaking at about 4% when securing the value binding of a dynamically obtained structure.

6 Related Work

Secure (fully abstract) compilation was first introduced by Abadi [1] as a criticism of the way in which Java was translated into the Java bytecode language. Secure compilation schemes have since been introduced for many different source language and target languages. Closely related to this work is the secure compilation scheme for ML to JavaScript by Fournet *et al.* [5]. Their definition of ML, however, does not feature a module system. Their Javascript attacker model is also more high-level than our untyped assembly contexts with low-level code execution privileges. Another related compilation scheme is the secure compilation scheme for the $\lambda_\mu$hashref-calculus to a machine model with adress space layout randomisation by Jagadeesan *et al.* [7]. Like the ModuleML used in this work the $\lambda_\mu$hashref-calculus features dynamic memory allocation. In contrast to ModuleML, locations in $\lambda_\mu$hashref are observable through a hash operation. The attacker model differs as well. Whereas the attacker in this work is unable to read the memory of the securely compiled program, due to the PMA mechanism, the attacker considered by Jagadeesan *et al.* can probe the memory.

Verified compilation, is a broad research topic that aims to provide compilers that are proven to be correct [2,12]. The resulting compilers thus come with proofs for the preservation property that we have assumed (Section 1). Many established verified compilation results hold only for closed world assumptions, but recently, verified compilers have appeared for partial programs as well. Related to this work is a verified compositional compiler for an ML language, that features references and recursive types, to assembly by Hur and Dreyer [6]. Their compiler preserves the equivalences of ML programs for well-behaved assembly contexts, but does not consider the threats posed by possibly malicious contexts.

Throughout the secure compilation scheme we make use of our previously developed interaction counting masking system [9] to securely share the values of security relevant abstractions. Alternatively, we could have applied the sealing mechanism of Matthews et al. [14], to achieve the same result.

7 Conclusions

This paper presented a secure compiler for ModuleML: a light-weight ML language with higher-order functions, references and a module system. This secure compilation scheme compiles ModuleML to untyped assembly code enhanced with a memory isolation mechanism, known as the Protected Module Architecture, in a way that reflects the equivalences of ModuleML. This security property is proven through the implementation of a witness building algorithm.

References

1. M. Abadi. Protection in programming-language translations. In *Secure Internet programming*, pages 19–34. Springer-Verlag, London, UK, 1999.
2. A. Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *PLDI '07*, pages 54–65, New York, NY, USA, 2007. ACM.
3. P. Codognet and D. Diaz. wamcc: Compiling Prolog to C. In *ICLP*, pages 317–331. MIT PRess, 1995.
4. D. Dreyer. *Understanding and Evolving the ML Module System.* PhD thesis, Carnegie Mellon, May 2005.
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. C.-K. Hur and D. Dreyer. A kripke logical relation between ml and assembly. In *POPL '11*, pages 133–146. ACM, 2011.
7. R. Jagadeesan, C. Pitcher, J. Rathke, and J. Riely. Local memory via layout randomization. In *CSF '11*, pages 161–174. IEEE, 2011.
8. A. Jeffrey and J. Rathke. A fully abstract may testing semantics for concurrent objects. *Theor. Comput. Sci.*, 338(1-3):17–63, 2005.
9. A. Larmuseau and D. Clarke. Formalizing a secure foreign function interface. In *SEFM 2015*, LNCS. To appear, currently at: https://db.tt/y87tcQ0V.
10. A. Larmuseau, M. Patrignani, and D. Clarke. A secure compiler for ml modules - extended version. Technical Report 2015-028, Uppsala University, September 2015.
11. X. Leroy. Manifest types, modules, and separate compilation. In *POPL '94*, pages 109–122, New York, NY, USA, 1994. ACM.
12. X. Leroy. Formal verification of a realistic compiler. *CACM*, 52(7):107–115, 2009.
13. X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vôuillon. The Objective Caml system, release 4.02. Technical report, INRIA, August 2014.
14. J. Matthews and A. Ahmed. Parametric polymorphism through run-time sealing or, theorems for low, low prices! In *ESOP 2008*, pages 16–31, 2008.
15. 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.
16. M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens. Secure compilation to protected module architectures. *TOPLAS*, 37(2):6:1–6:50, 2015.
17. M. Patrignani and D. Clarke. Fully Abstract Trace Semantics of Low-level Isolation Mechanisms. In *SAC '14*, pages 1562–1569. ACM, 2014.
18. C. Queinnec. *Lisp in small pieces.* Cambridge University Press, 2003.
19. R. Strackx and F. Piessens. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *CCS*, pages 2–13, 2012.