A Secure Compiler for ML Modules – Extended Version

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. Most 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 the low-level interactions. We evaluate an implementation of the compiler on relevant test scenarios.

Keywords: Secure Compilation, Modules, Contextual Equivalence

1 Introduction

High-level functional programming languages such as ML and Haskell offer programmers numerous security features through abstractions such as type systems, module systems and encapsulation primitives. Motivated by speed, memory efficiency and portability these high-level functional programming languages are often compiled to low-level target languages such as C and assembly [4,14]. The security features of 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) [23]. PMA is a low-level memory isolation mechanism, that protects a certain memory area by restricting access to that area based on the location of the program counter. Our 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 target language [19], 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 of the compilation scheme intensely (Section 5), we consider formally verifying the implementation of the 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.* [19] 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 created through constructors 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 of different types that require different type checks and security constraints. In this work we in-

vestigate all of the security challenges introduced by functors and develop an efficient method of encoding the required checks.

Limitations Our result is limited in that we do not consider the effects of certain low-level technical challenges such as: integer overflows, stack overflows and out of memory errors.

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 and the module system is based on Leroy's variant of the SML module system that incorporates a strict distinction between abstract types and manifest types in the signatures [11].

The Core 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. A formal definition of the syntax is given below.

$$
\begin{array}{lll}
\text{Expressions:} & e & ::= v_i \quad \text{Value Identifier} \\
& | & p.v_i \quad \text{Value Identifier of a Structure p} \\
& | & v \mid x \mid (e_1\ e_2) \mid \langle e_1, e_2 \rangle \\
& | & \texttt{op}\ e_1\ e_2 \mid \texttt{fst}\ e \mid \texttt{snd}\ e \mid \texttt{if}\ e_1\ e_2\ e_3 \\
& | & \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid \texttt{!}e \mid \texttt{ref}\ e \mid \texttt{exit}\ e \\
& | & \texttt{letrec}\ x : \tau = e_1\ \texttt{in}\ e_2 \mid e_1; e_2 \mid \texttt{fix}\ e \\
\text{Operands:} & \texttt{op} & ::= + \mid - \mid * \mid < \mid > \mid == \\
\text{Values:} & v & ::= \texttt{unit} \mid l \mid \overline{n} \mid b \mid \texttt{fun}\ x : \tau = e \mid \langle v_1, v_2 \rangle \\
\text{Booleans:} & b & ::= \texttt{true} \mid \texttt{false} \\
\text{Expression Types:} & \tau & ::= T_i \quad \text{Type Identifier} \\
& | & p.T_i \quad \text{Type Component of a Structure p} \\
& | & \textbf{bool} \mid \textbf{int} \mid \textbf{unit} \mid \tau_1 \rightarrow \tau_2 \\
& | & \textbf{ref}\ \tau \mid \tau_1 \times \tau_2
\end{array}
$$

Note that v_i and t_i are identifiers of the module system. These module identifiers cannot be renamed since they form the paths of the module systems, as such the run-time language will extend them with stamps that enable alpha conversion. Note also that x are variables and $\overline{n}$ indicates the syntactic term representing the number n.

The locations l of the core language are an artefact of the static semantics that do not appear in the syntax used by programmers. This is in accordance to most commonly used ML variants where locations do not have an explicit representation [14]. The locations are tracked in a location store $\mu ::= \emptyset \mid \mu, l = v$. The `exit` operator terminates execution with a value v, this addition is needed to satisfy the requirements of the witness algorithm of Section 4. The `letrec` operator is implemented using the `fix` operator which is also an artefact of the static semantics that does no appear in the syntax for the programmers.

The typing rules of the core language are fully standard. A formal definition of the rules is listed below.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x \ : \ \tau} \qquad \frac{}{\Gamma \vdash b \ : \ \textbf{bool}} \qquad \frac{}{\Gamma \vdash \overline{n} : \textbf{int}}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{fst}\ e \ : \ \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{snd}\ e \ : \ \tau_2} \qquad \frac{}{\Gamma \vdash \texttt{unit} : \textbf{unit}}$$

$$\frac{\Gamma \vdash e_1 : \textbf{int} \quad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash (\texttt{op}\ e_1\ e_2) \ : \ \textbf{int}} \qquad \frac{\Gamma \vdash e_1 : \textbf{int} \quad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash (\texttt{cp}\ e_1\ e_2) \ : \ \textbf{bool}} \qquad \frac{\Gamma \vdash e_1 : \tau \to \tau}{\Gamma \vdash \texttt{fix}\ e_1 : \tau}$$

$$\frac{\Gamma(l) : \tau}{\Gamma \vdash l : \textbf{ref}\ \tau} \qquad \frac{\Gamma \vdash e_1 : \textbf{ref}\ \tau}{\Gamma \vdash !e_1 : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau}{\Gamma \vdash \texttt{ref}\ e_1 : \textbf{ref}\ \tau}$$

$$\frac{\Gamma \vdash e_1 : \textbf{unit} \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1;\ e_2 : \tau} \qquad \frac{\Gamma \vdash v_1 : \tau}{\Gamma \vdash \textbf{exit}\ v_1 : \tau} \qquad \frac{\Gamma \vdash e_1 : \textbf{ref}\ \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : \textbf{unit}}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\texttt{fun}\ x : \tau_1 = e) \ : \ \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \ : \ \tau_2}$$

$$\frac{\Gamma, x : \tau_1 \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{letrec}\ x : \tau_1 = e_1\ \texttt{in}\ e_2 \ : \ \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 \ : \ \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle \ : \ \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e_1 : \textbf{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{if}\ e_1\ e_2\ e_3 \ : \ \tau}$$

$$\frac{\Gamma = \Gamma';\ \textbf{val}\ v_i : \tau; \Gamma''}{\Gamma \vdash v_i \ : \ \tau} \qquad \frac{\Gamma \vdash p : (\textbf{sig}\ S_1;\ \textbf{val}\ v_i : \tau;\ S_2 \textbf{end})}{\Gamma \vdash p.v_i \ : \ \tau\{n_i \leftarrow p.n \ \mid \ n_i \in Dom(S_1)\}}$$

The reduction rules use reduction contexts E to lift the basic reduction steps of the λ-calculus to a standard left-to-right call-by-value semantics as per Felleisen-and-Hieb [6].

$$\begin{aligned} E ::= {} & [\cdot] \mid E\ e \mid v\ E \mid \langle E, e \rangle \mid \langle v, E \rangle \mid \texttt{fst}\ E \mid \texttt{snd}\ E \mid \texttt{op}\ E\ e \\ & \mid \texttt{op}\ v\ E \mid \texttt{if}\ E\ e_1\ e_2 \mid \texttt{let}\ x = E\ \texttt{in}\ e \mid \texttt{!}E \mid \texttt{ref}\ E \mid E \ \texttt{:=}\ e \mid E;\ e \\ & \mid v \ \texttt{:=}\ E \mid \texttt{fix}\ E \mid \texttt{exit}\ E \end{aligned}$$

The reduction rules are largely standard except for the inclusion of Ω which stores the module environment as detailed in the following paragraph.

$$\frac{}{\Omega|\mu \mid E[\texttt{if}\ \#t\ e_2\ e_3] \longrightarrow_l \Omega|\mu \mid E[e_2]} \quad \frac{}{\Omega|\mu \mid E[\texttt{if}\ \#f\ e_2\ e_3] \longrightarrow_l \Omega|\mu \mid E[e_3]}$$

$$\frac{}{\Omega|\mu \mid E[((\texttt{fun}\ x : \tau = e)\ v)] \longrightarrow_l \Omega|\mu \mid E[e']}$$

$$\frac{}{\Omega|\mu \mid E[\texttt{let}\ x = v\ in\ e] \longrightarrow_l \Omega|\mu \mid E[e[v/x]]}$$

$$\frac{e' = \texttt{let}\ x = \texttt{fix}(\lambda x : \tau.e_1)\ \texttt{in}\ e_2}{\Omega|\mu \mid E[\texttt{letrec}\ x : \tau = e_1\ \texttt{in}\ e_2] \longrightarrow_l \Omega|\mu \mid E[e_2[v/x]]}$$

$$\frac{}{\Omega|\mu \mid E[\texttt{fst}\ \langle v_1, v_2\rangle] \longrightarrow_l \Omega|\mu \mid E[v_1]} \quad \frac{}{\Omega|\mu \mid E[\texttt{snd}\ \langle v_1, v_2\rangle] \longrightarrow_l \Omega|\mu \mid E[v_2]}$$

$$\frac{}{\Omega|\mu \mid E[\texttt{op}\ \overline{n_1}\ \overline{n_2}] \longrightarrow_l \Omega|\mu \mid E[\overline{max(rnd(n_1\ \texttt{op}\ n_2), 0)}]}$$

$$\frac{}{\Omega|\mu \mid E[\texttt{cp}\ \overline{n_1}\ \overline{n_2}] \longrightarrow_l \Omega|\mu \mid E[b]} \quad \frac{\mu' = \mu, [l \mapsto v]}{\Omega|\mu \mid E[\texttt{ref}\ v] \longrightarrow_l \Omega \mid \mu' \mid E[l]}$$

$$\frac{\mu(l) = v}{\Omega|\mu \mid E[!l] \longrightarrow_l \Omega|\mu \mid E[v]} \quad \frac{}{\Omega|\mu \mid E[\texttt{unit}; e_2] \longrightarrow_l \Omega|\mu \mid E[e_2]}$$

$$\frac{}{\Omega|\mu \mid E[\texttt{fix}(\lambda x : \tau.e)] \longrightarrow_l \Omega|\mu \mid E[e[(\lambda x : \tau.e)/x]]}$$

$$\frac{\Omega(v_i) = e}{\Omega|\mu \Vdash E[v_i] \longrightarrow_l \Omega|\mu \Vdash E[e]} \quad \frac{}{\Omega|\mu \Vdash E[\texttt{exit}\ v] \longrightarrow_l \Omega|\mu' \Vdash v}$$

$$\frac{\mu' = \mu, [l \mapsto v]}{\Omega|\mu \mid E[l := v] \longrightarrow_l \Omega \mid \mu' \mid E[\texttt{unit}]} \quad \frac{\Omega|\mu \Vdash p \longrightarrow_m^* \Omega'|\mu \quad \Omega'(v_i) = v}{\Omega|\mu \Vdash E[p.v_i] \longrightarrow_l \Omega|\mu \Vdash E[v]}$$

The Module System As mentioned previously the module system is an adaption of Leroy's variant of the SML module system that features manifest types [11]. It consists of structures, functors and signatures. Structures are a sequence of structure components c that are either value bindings, module bindings or type bindings. The components of structures are assumed to have distinct names. Functors are functions from modules to modules. Signatures are a sequence of signature components C that are either value declarations, abstract or manifest type declarations and module declarations. Our compiler also supports simplified module inclusion through an `open` X_i instruction.

To keep the formalisation as simple as possible the module system does not feature standalone signatures binding. Our compiler does, however, support signatures as simple type bindings but not as structure components. The full abstraction and typing consequences of the latter were not investigated. The syntax of the module system is listed below.

Paths:	p ::=	X_i	Module Identifier
	|	$p.X_i$	Module Id of a Structure
Module Expressions:	m ::=	p	Paths
	|	**struct** s **end**	Module Structure
	|	$(m : M)$	Type Ascription
	|	**functor**$(X_i : M)\ m$	Functor with body m
	|	$m_1(m_2)$	Module Application
Structure Body:	s ::=	ϵ | $c;\ s$	
Structure Components:	C ::=	**val** $v_i = e$	Value Binding
	|	**type** $T_i = \tau$	Type Binding
	|	**module** $X_i = m$	Module Binding
Programs:	P ::=	**struct** s **end** ;; e	
Module Types:	M ::=	**sig** S **end**	Signature Type
	|	**functor**$(X_i : M) \to M'$	Functor Type
Signature Body:	S ::=	ϵ | $C;\ S$	
Signature Components:	C ::=	**val** $v_i : \tau$	Value Declaration
	|	**type** T_i	Abstract Type
	|	**type** $T_i = \tau$	Manifest Type
	|	**module** $X_i : M$	Module Declaration

Note that a program P in ModuleML is defined as a sequence of structure definitions followed by a core language expression e that functions as the starting point of the program P. ModuleML thus does not support standalone core expressions.

The typing rules for the ModuleML module system are largely standard. In this work we use SML style generative functors which return "fresh" abstract types with each application, in contrast to Leroy's applicative functors [12] that return the same abstract type with each application. While applicative functors allow for higher modularity, generative functors provide better data encapsulation and are thus more in line with our overall goal of secure compilation [5]. A formalisation of the typing rules is given below.

$$\frac{(\Gamma = \Gamma';\ \textbf{module}\ X_i : M;\ \Gamma'')}{\Gamma \vdash X_i\ :\ M} \qquad \frac{\Gamma \vdash p : M}{\Gamma \vdash p : M/p}$$

$$\frac{\Gamma \vdash m : M' \quad \Gamma \vdash M' <: M}{\Gamma \vdash m\ :\ M} \qquad \frac{\Gamma \vdash s : S}{\Gamma \vdash \textbf{struct}\ s\ \textbf{end}\ :\ \textbf{sig}\ S\ \textbf{end}}$$

$$\frac{\Gamma \vdash M\ \mathsf{wf} \quad X_i \notin Dom(\Gamma) \quad (\Gamma; \textbf{module}\ X_i : M) \vdash m : M'}{\Gamma \vdash \textbf{functor}(X_i : M)\ m : \textbf{functor}(X_i : M) \to M'}$$

$$\frac{\Gamma \vdash m_1 : \textbf{functor}(X_i : M) \to M' \quad \Gamma \vdash m_2 : M}{\Gamma \vdash m_1(m_2)\ :\ M'\{X_i \mapsto m_2\}}$$

$$\frac{\Gamma \vdash p : (\textbf{sig}\ S_1;\ \textbf{module}\ X_i : M;\ S_2\ \textbf{end})}{\Gamma \vdash p.X\ \ :\ \ M\{n_i \mapsto p.n \mid n_i \in Dom(S_1)\}}$$

$$\frac{\Gamma \vdash e : \tau \quad v_i \notin Dom(\Gamma) \quad (\Gamma;\ \textbf{val}\ v_i : \tau) \vdash s : S}{\Gamma \vdash (\textbf{val}\ v_i = e;\ s)\ \ :\ \ (\textbf{val}\ v_i : \tau;\ S)}$$

$$\frac{\Gamma \vdash \tau \quad T_i \notin Dom(\Gamma) \quad (\Gamma;\ \textbf{type}\ T_i = \tau) \vdash s : S}{\Gamma \vdash (\textbf{type}\ T_i = \tau;\ s)\ \ :\ \ (\textbf{type}\ T_i = \tau;\ S)}$$

$$\frac{\begin{array}{c}\Gamma \vdash m : M \quad X_1 \notin Dom(\Gamma) \\ (\Gamma;\ \textbf{module}\ X_1 : M) \vdash s : S\end{array}}{\Gamma \vdash (\textbf{module}\ X_1 : M = m;\ s) : (\textbf{module}\ X_1 : M;\ S)}$$

$$\frac{\Gamma \vdash M \mathsf{wf} \quad \Gamma \vdash m : M}{\Gamma \vdash (m : M) : M} \qquad \frac{\Gamma \vdash s : S \quad \Gamma;\ \textbf{sig}\ S\ \textbf{end} \vdash e : \tau}{\Gamma \vdash \textbf{struct}\ s\ \textbf{end}\ ;;\ e\ \mathsf{ok}} \qquad \frac{}{\Gamma \vdash \epsilon : \epsilon}$$

The functor application rule states that the type of the argument must be well-formed. The well-formedness rules are standard and are thus omitted. The path typing rule relies on a type strengthening rule: (M/p), that enriches the module type M to reflect that its abstract type components come from the path p. The strengthened type M/p is a subtype of M. Hence, it is always safe to apply the strengthening rule before checking type inclusion. Type strengthening is defined as follows:

$$\begin{aligned}
(\textbf{sig}\ S\ \textbf{end})/p &= \textbf{sig}\ S/p\ \textbf{end} \\
(\textbf{functor}(X_i : M) \rightarrow M')/p &= \textbf{functor}(X_i : M) \rightarrow M' \\
\epsilon/p &= \epsilon \\
(\textbf{val}\ v_i : \tau;\ S)/p &= \textbf{val}\ v_i : \tau;\ S/p \\
(\textbf{type}\ T_i;\ S)/p &= \textbf{type}\ T_i = p.T;\ S/p \\
(\textbf{type}\ T_i = \tau;\ S)/p &= \textbf{type}\ T_i = p.T;\ S/p \\
(\textbf{module}\ X_i : M;\ S)/p &= \textbf{module}\ X_i : M/p.X_i;\ S/p
\end{aligned}$$

The following sub-typing rules, are as expected.

$$\frac{\begin{array}{c}\Gamma \vdash M_2 <: M_1 \\ (\Gamma; \textbf{module}\ X_i : M_2) \vdash M_1' <: M_2'\end{array}}{\Gamma \vdash \textbf{functor}(X_i : M_1) \rightarrow M_1' <: \textbf{functor}(X_i : M_2) \rightarrow M_2'}$$

$$\frac{\begin{array}{c}f : \{1, ..., l\} \mapsto \{1, ..., k\} \\ \forall i \in \{1, ..., l\}.\Gamma; C_1; ...; C_k \vdash C_{f(i)} <: C_i'\end{array}}{\Gamma \vdash \textbf{sig}\ C_1; ...; C_k\ \textbf{end} <: \textbf{sig}\ C_1'; ...; C_l'\ \textbf{end}}$$

$$\frac{\Gamma \vdash \tau <: \tau'}{\Gamma \vdash (\textbf{val}\ v_i : \tau) <: (\textbf{val}\ v_i : \tau')} \qquad \frac{}{\Gamma \vdash (\textbf{type}\ T_i = \tau) <: (\textbf{type}\ T_i)}$$

$$\frac{}{\Gamma \vdash (\textbf{type}\ T_i) <: (\textbf{type}\ T_i)} \qquad \frac{\Gamma \vdash T_i \approx \tau}{\Gamma \vdash (\textbf{type}\ T_i) <: (\textbf{type}\ T_i = \tau)}$$

$$\frac{\Gamma \vdash \tau \approx \tau'}{\Gamma \vdash (\textbf{type}\ T_i = \tau) <: (\textbf{type}\ T_i = \tau')} \qquad \frac{\Gamma \vdash M <: M'}{\Gamma \vdash (\textbf{module}\ X_i : M) <: (\textbf{module}\ X_i : M')}$$

$$\frac{\Gamma = \Gamma';\ \textbf{type}\ T_i = \tau;\ \Gamma''}{\Gamma \vdash T_i \approx \tau} \qquad \frac{\Gamma \vdash p : (\textbf{sig}\ S_1;\ \textbf{type}\ T_i = \tau;\ S_2\ \textbf{end})}{\Gamma \vdash p.T \approx \tau\{p.n \mapsto n_i \ \mid\ n_i \in Dom(S_1)\}}$$

We provide a dynamic semantics for ModuleML that will serve as a guideline for the implementation of the secure compiler. The dynamics semantics are based on Leroy's calculus of Applicative functors [12], excluding the applicative functor part. The dynamic semantics are defined through big step reduction rules that convert a module m into an environment Ω and a location store μ, formally defined as $\mu ::= \ \emptyset \mid \mu, l = v$. The environment Ω stores the module and value bindings defined in the module. Formally Ω is defined as:

$$\Omega ::= \varepsilon \ \mid\ v_i \mapsto v;\ \Omega \ \mid\ X_i \mapsto \Omega';\ \Omega \ \mid\ (\lambda X_i.m);\ \Omega$$

The reduction rules for the modules of ModuleML, denoted as $\Omega|\mu \Vdash m \longrightarrow_m \Omega'|\mu' \Vdash \Omega''$, are illustrated below.

$$\frac{\Omega(X_i) = \Omega'}{\Omega|\mu \Vdash X_i \longrightarrow_m \Omega|\mu \Vdash \Omega'} \qquad \frac{\Omega|\mu \Vdash m \longrightarrow_m \Omega'|\mu' \Vdash \Omega''}{\Omega|\mu \Vdash (m : M) \longrightarrow_m \Omega'|\mu' \Vdash \Omega''}$$

$$\frac{\Omega|\mu \Vdash p \longrightarrow_m \Omega|\mu \Vdash \Omega' \quad \Omega'(X_i) = \Omega''}{\Omega|\mu \Vdash p.X_i \longrightarrow_m \Omega|\mu \Vdash \Omega''} \qquad \frac{\Omega|\mu \Vdash s \longrightarrow_s \Omega'|\mu' \Vdash \Omega'}{\Omega|\mu \Vdash \textbf{struct}\ s\ \textbf{end} \longrightarrow_m \Omega'|\mu' \Vdash \Omega'}$$

$$\frac{}{\Omega|\mu \Vdash (\textbf{functor}(X_i : M)m) \longrightarrow_m \Omega|\mu \Vdash (\lambda X_i.m);\ \Omega}$$

$$\frac{\begin{array}{c}\Omega|\mu \Vdash m_1 \longrightarrow_m \Omega|\mu' \Vdash (\lambda X_i.m_f); \Omega_f \\ \Omega|\mu' \Vdash m_2 \longrightarrow_m \Omega^m|\mu'' \Vdash \Omega^m\end{array}}{\Omega|\mu \Vdash m_1(m_2) \longrightarrow_m \Omega_f; X_i \mapsto \Omega^m|\mu''| \Vdash m_f}$$

These reduction rules rely on two other reduction relations. A reduction relation $\Omega|\mu|t \longrightarrow_l \Omega'|\mu'|t'$: the reduction rules for the core language and a reduction relation $\Omega|\mu|s \longrightarrow_s \mu'|\Omega$ that converts a structure body s to an Ω binding, as listed below.

$$\frac{\begin{array}{c}\Omega|\mu \Vdash m \longrightarrow_m \Omega|\mu' \Vdash \Omega^m \\ \Omega; X_i \mapsto \Omega^m|\mu' \Vdash s \longrightarrow_s \Omega'|\mu'' \Vdash \Omega^s\end{array}}{\Omega|\mu \Vdash \textbf{module}\ X_i = m;\ s \longrightarrow_s \Omega|\mu'' \Vdash (X_i \to \Omega^m;\ \Omega^s)}$$

$$\frac{\begin{array}{c}\Omega|\mu \Vdash e \longrightarrow_l^* \Omega|\mu' \Vdash v \\ \Omega; v_i \mapsto v|\mu' \Vdash s \longrightarrow_s \Omega'|\mu''| \Vdash \Omega^s\end{array}}{\Omega|\mu \Vdash \textbf{val}\ v_i = e;\ s \longrightarrow_s \Omega|\mu''| \Vdash (v_i \mapsto v;\ \Omega^s)}$$

$$\frac{\Omega|\mu \Vdash s \longrightarrow_s \Omega'|\mu' \Vdash \Omega_s}{\Omega|\mu \Vdash \textbf{type}\ t = \tau; s \longrightarrow_s \Omega|\mu' \Vdash \Omega_s} \quad \frac{}{\Omega|\mu \Vdash \epsilon \longrightarrow_s \Omega|\mu \Vdash \varepsilon}$$

Full program reduction, denoted as $\varepsilon|\emptyset|P \longrightarrow_p \Omega|\mu|v$, can now be defined as:

$$\frac{\varepsilon|\emptyset \Vdash s \longrightarrow_m \varepsilon|\mu| \Vdash \Omega|\mu \quad \Omega|\mu \Vdash e \longrightarrow_l \Omega|\mu'|v}{\varepsilon|\emptyset \Vdash \textbf{struct}\ s\ \textbf{end};; e \longrightarrow_p \Omega|\mu' \mid v}$$

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

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

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 includes either the module $\mathtt{M}_1$ or $\mathtt{M}_2$ into C, implementing the hole of the context.

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 consisting of a program counter p, a register file r, a flags register f and a memory space m that maps addresses to words and contains all code and data. The supported instructions are the standard assembly instructions for integer arithmetic, value comparison, address jumping, stack pushing and popping, register loading and memory storing. For a full formalisation of these instructions and their operational semantics we refer to Patrignani's and Clarke's formalisation [20].

In order to support fully abstract compilation, without resorting to static checks on the assembly code, some memory protection mechanism is needed [2,19]. In this paper we use one such low-level memory protection mechanism referred to as Protected Module Architecture (PMA). PMA is a fine-grained, program counter-based, memory access control mechanism that divides memory

into protected and unprotected memory. The protected memory is further split into two sections: a protected code section accessible only through a fixed of collection of designated entry points, and a protected data section that can only be accessed by the code section. As such the unprotected memory is limited to executing the code at entry points, neither the code nor the data of the protected memory can be executed, written or read by the unprotected memory. The code section can only be executed by the entrypoints and the data can only accessed by the code section. An overview of the access control mechanism between the protected and unprotected memory 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 [18,23], Intel is developing a new instruction set, referred to as SGX, that enable the usage of PMA in commercial processors [17].

Trace Equivalence Our secure compiler relates contextually equivalent ModuleML modules to contextually equivalent low-level components. Reasoning about contexts is, however, notoriously complex [24]. 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 [20].

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 $(\mathbf{unknown}, m, s)$, a state modelling that code is executing in unprotected memory. The trace semantics denotes the observations of the low-level A+I contexts that interact with the protected memory through labels L . The labels are formalised 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 include a tick $\surd$ indicating that the evaluation has terminated observable actions Additionally, observable actions 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.

Formally the trace semantics of a low-level A+I program L, denoted as $\mathsf{Traces}(L)$, are computed as follows: $\mathsf{Traces}(L) = \{\overline{\alpha} | \exists \Lambda. \Lambda_0(L) \xRightarrow{\overline{\alpha}} \Lambda\}$. Where Λ_0 is the initial state and the relation $\Lambda \xRightarrow{\overline{\alpha}} \Lambda'$ describes the traces generated by transitions between states. Two A+I programs L_1 and L_2 are trace-equivalent, denoted as $L_1 \simeq_t L_2$, if their traces are the same.

Definition 2 (Trace Equivalence).

$$P_1 \simeq_t P_2 \ if \ \mathsf{Traces}(P_1) = \mathsf{Traces}(P_2).$$

An important property of this trace equivalence is that it is as precise as contextual equivalence in the target language:

Proposition 1 (Full Abstraction of the Traces [20]).

$$P_1 \simeq_t P_2 \iff P_1 \simeq_l P_2$$

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 where injected code operates with kernel-level privileges and thus bypasses all existing software-based security mechanisms disclosing confidential data, disrupting running applications and so forth. For the sake of simplicity, no differentiation between kernel and user code is defined in A+I: all code is already operating at the kernel level. Thus, by modelling the attacker as injecting A+I code, we are modelling kernel-level code injection.

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 [19] and full abstraction results [15].

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.

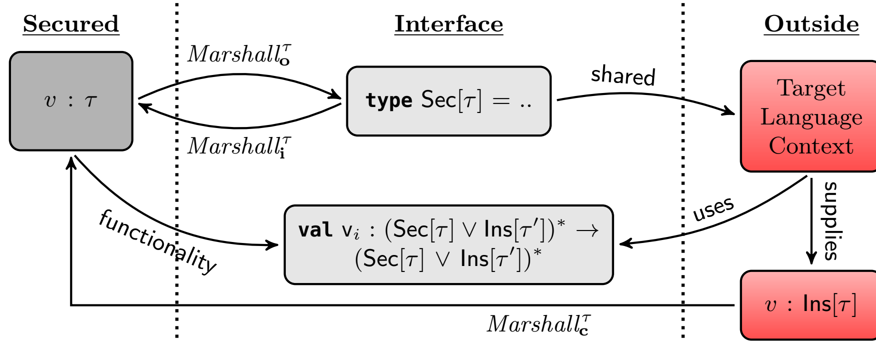

Fig. 1. The Secure ADT pattern isolates an abstraction of type τ through an ADT-like interface that shares secured instances of τ ($\mathsf{Sec}[\tau]$) and accepts outside input ($\mathsf{Ins}[\tau']$). Marshalling functions handle the transitions between the representations.

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 the source language. These 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 $Marshall^{\tau}_{\mathbf{o}} : \tau \rightarrow \mathsf{Sec}[\tau]$ converts values into their secured instances. The function $Marshall^{\tau}_{\mathbf{i}} : \mathsf{Sec}[\tau] \rightarrow \tau$ 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: $Marshall^{\tau}_{\mathbf{c}} : \mathsf{Ins}[\tau] \rightarrow \tau$.

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: $\mathit{Marshall}^{\tau'}_{\mathsf{c}} : \mathsf{Ins}[\tau'] \rightarrow \tau'$.

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 it's signature to a protected module (Figure 2) separating it from the low-level A+I context in the unprotected memory. This protected module is always of a fixed size, ensuring that the attacker cannot observe the size of the source program.

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 that can only be accessed by the securely compiled program. This ensures that the run-time memory of the compiled program is also inaccessible to the low-level A+I context.

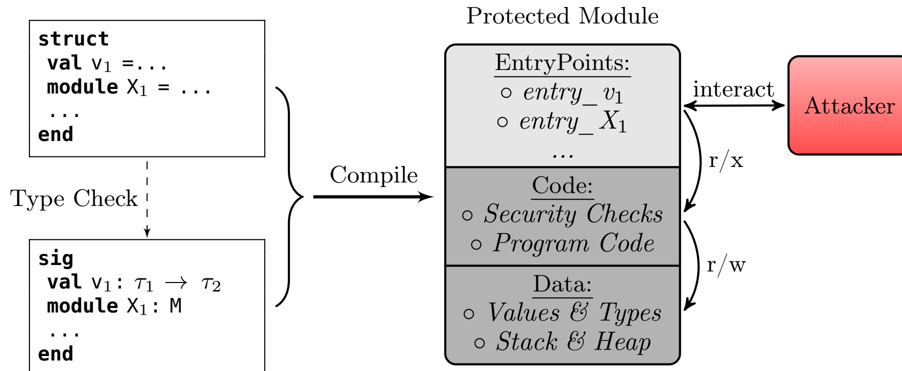

Fig. 2. Our scheme compiles the module and its type into the protected memory. The entry points provides the ADT-like interface, the data section isolates the abstractions.

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 sequences are not compiled using the Secure ADT pattern, but must still be marshalled when interacting with the A+I context (Section 3.1).

3.1 Booleans, Integers and Sequences

The securely compiled module shares and inputs not only abstractions such as functions, but also basic ModuleML values: booleans, integers and sequences. 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 $\mathit{Marshall}_{\mathsf{c}}^{\mathtt{int}} : \mathsf{Ins}[\mathtt{int}] \to \mathtt{int}$, which converts A+I integers into ModuleML integers, and $\mathit{Marshall}_{\mathsf{o}}^{\mathtt{int}} : \mathtt{int} \to \mathsf{Ins}[\mathtt{int}]$, which converts ModuleML integers to A+I integers. The marshalling functions for booleans are analogous.

Marshalling sequences is different. When marshalling, for example, a pair $\langle v_1, v_2 \rangle$ the marshalling functions for sequences marshall each value with the Marshalling *out* the pair $\langle 1, 2 \rangle$, for example, will thus produce a value of type $\langle \mathsf{Ins}[\mathtt{int}], \mathsf{Ins}[\mathtt{int}] \rangle$, while marshalling out the $\langle (\lambda x : \tau.t), (\lambda x : \tau.t') \rangle$ 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 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 A+I context, we share a secured instance of type Sec[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 A+I context has made to abstract types. Note that each member of a sequence (Section 3.1) counts as a separate request. The marshalling functions $\mathit{Marshall}_{\mathsf{o}}^{\mathsf{T}}$ and $\mathit{Marshall}_{\mathsf{i}}^{\mathsf{T}}$ are thus implemented as extending the table $\mathcal{A}$ and looking up an index in $\mathcal{A}$ respectively, as illustrated in Figure 3.

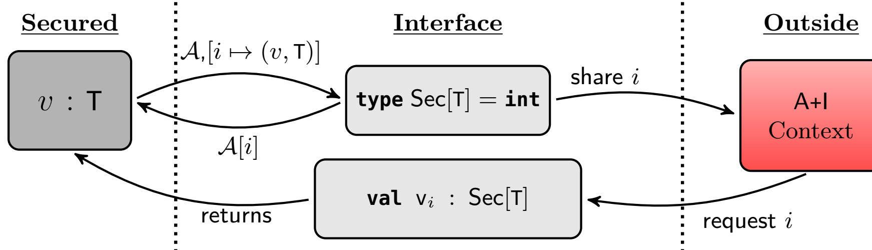

Fig. 3. We use request counting to obscure the value of an abstract type.

We have formally proven in prior work [10], by means of a full abstraction proof, that these request counting indices do not reveal any information to the A+I context other than the number of times the 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 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 [19], 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 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 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 that the flag L_f is set. If not abort.
2. For each and every value binding v_i in the compiled module:
   (a) Evaluate the expression e.
   (b) 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 that the flag L_f is set. If not abort.
2. Fetch the value v and its type τ from the data section.
3. Return $Marshall^{\tau}_{\mathbf{o}}(v)$

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 that the flag L_f is set. If not abort.
2. Return $Marshall^{\mathsf{S}}_{\mathbf{o}}(s)$

$Marshall^{\mathsf{S}}_{\mathbf{o}}$ and $Marshall^{\mathsf{S}}_{\mathbf{i}}$ As dictated by the Secure ADT pattern, structures are not shared directly but instead marshalled out using a type directed function $Marshall^{\mathsf{S}}_{\mathbf{o}} : \mathsf{S} \to \mathsf{Sec}[\mathsf{S}]$. 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 $Marshall^{\mathsf{S}}_{\mathbf{i}} : \mathsf{Sec}[\mathsf{S}] \to \mathsf{S}$ to retrieve the original structure and its signature from $\mathcal{M}$. Note that this marshalling function $Marshall^{\mathsf{S}}_{\mathbf{i}} : \mathsf{Sec}[\mathsf{S}] \to \mathsf{S}$ performs an implicit type check as the function fails whenever the retrieved signature is not a *subtype* of S.

$Marshall^{\mathsf{S}}_{\mathbf{c}}$ 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 $Marshall^{\mathsf{S}}_{\mathbf{c}} : \mathsf{Ins}[\mathsf{S}] \to \mathsf{S}$, 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: $\mathsf{Ins}[\mathsf{S}] <: \mathsf{S}$ as it only verifies the bindings defined by S. The marshalling is thus an immediate

process, as marshalling in a lazy, on demand, manner would expose to the A+I context how and when its structure is used, something a ModuleML context cannot observe.

3.4 Higher-Order Functions

To compile the λ-terms of ModuleML the compiler uses closure conversion [22] 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 $\mathsf{Sec}[\tau_1 \to \tau_2]$: 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 $\mathit{Marshall}_{\mathbf{o}}^{\tau_1 \to \tau_2}$ and $\mathit{Marshall}_{\mathbf{i}}^{\tau_1 \to \tau_2}$ 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 the shared closured through an entry point of type: $\mathtt{appl} : \mathsf{Sec}[\tau_1 \to \tau_2] \to (\mathsf{Ins}[\tau_1] \vee \mathsf{Sec}[\tau_1]) \to (\mathsf{Ins}[\tau_2] \vee \mathsf{Sec}[\tau_2])$, 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 that the flag L_f is set. If not abort.
2. $c = \mathit{Marshall}_{\mathbf{i}}^{\tau_1 \to \tau_2}(i)$
3. Depending on the representation of v:
   (a) If $\mathsf{Ins}[\tau_1]$: $r = \mathit{Marshall}_{\mathbf{c}}^{\tau_1}(v)$
   (b) If $\mathsf{Sec}[\tau_1]$: $r = \mathit{Marshall}_{\mathbf{i}}^{\tau_1}(v)$
4. Apply c to v, store the result in r'.
5. Return $\mathit{Marshall}_{\mathbf{o}}^{\tau_2}(r')$

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.

$\mathit{Marshall}_{\mathbf{c}}^{\tau_1 \to \tau_2}$ 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: $\mathsf{Ins}[\tau_1 \to \tau_2]$. 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. Note that the type marshalling rule on line 3 ensures that the result conforms to the typing rules of ModuleML.

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

An example Consider, for example, the following two contextually equivalent implementations of the ModuleML value bindings $\mathtt{v}_1$ that applies two lambda's to its argument `g`, a function that takes two other functions as arguments, where $\mathtt{f}_1$ and $\mathtt{f}_2$ are two contextually equivalent lambdas.

```
val v₁ g:((int→ int)→(int→ int))→int = (g f₁) f₂
```

```
val v₁ g:((int→ int)→(int→ int))→int = (g f₁) f₁
```

The threat to compiler security in this example is that the low-level context could supply a function `g` that could distinguish between both implementations of $\mathtt{v}_1$, by observing that the right implementation applies the argument to $\mathtt{f}_1$ twice in contrast to the left implementation. The secure ADT pattern prevents this by enforcing that every time a closure is exposed to the low-level context it is shared through a fresh mask. The attacker will thus observe two fresh masks in its interactions with both implementations.

3.5 Locations

As is the case in most commonly used ML variants [14], 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 $\mathit{Marshall}_{\mathsf{o}}^{\mathbf{ref}\ \tau}$ and $\mathit{Marshall}_{\mathsf{i}}^{\mathbf{ref}\ \tau}$ 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: $\mathtt{write} : \mathsf{Sec}[\mathbf{ref}\ \tau] \rightarrow (\mathsf{Ins}[\tau] \vee \mathsf{Sec}[\tau]) \rightarrow \mathbf{unit}$, and a *read location* entry point of type $\mathtt{read} : \mathsf{Sec}[\mathbf{ref}\ \tau] \rightarrow (\mathsf{Ins}[\tau] \vee \mathsf{Sec}[\tau])$. 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 that the flag L_f is set. If not abort.
2. $l = Marshall_{\mathbf{i}}^{\mathbf{ref}\ \tau}(i)$.
3. Depending on the representation of v:
   (a) If $\mathsf{Ins}[\tau]$: $r = Marshall_{\mathbf{c}}^{\tau}(v)$
   (b) If $\mathsf{Sec}[\tau]$: $r = Marshall_{\mathbf{i}}^{\tau}(v)$
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.

$Marshall_{\mathbf{c}}^{\mathbf{ref}\ \tau}$ 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 $Marshall_{\mathbf{c}}^{\mathbf{ref}\ \tau} : \mathsf{Ins}[\mathbf{ref}\ \tau] \rightarrow \mathbf{ref}\ \tau$, 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
 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 and argument `y`, and a inner functor F_1 and an inner structure M_1 that copy 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 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 $Marshall_{\mathbf{o}}^{\mathbf{functor}(X_i:S)\to S'}$ and $Marshall_{\mathbf{i}}^{\mathbf{functor}(X_i:S)\to S'}$, where $\mathbf{functor}(X_i : S) \to S'$ 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 $Marshall_{\mathbf{c}}^{\mathbf{functor}\ (X_i:S)\to S'}$ 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, that defines a unique stamp Σ_i for each non-leaf node (Figure 4). These stamps Σ_i are used by the entry points for these bindings to authenticate the modules and inner modules that result from the A+I context applying the compiled functor.

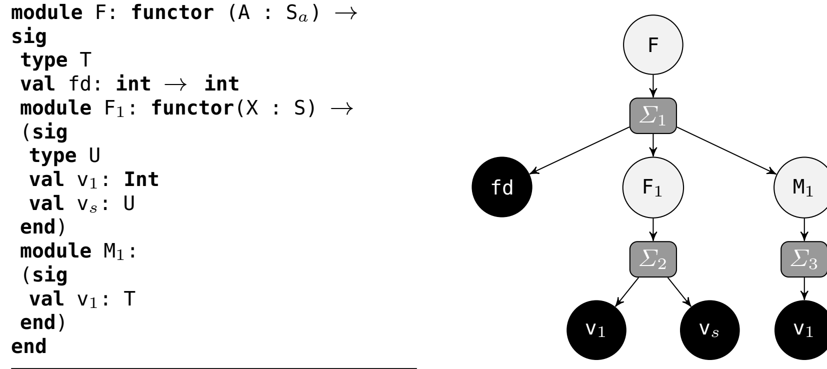

Fig. 4. The secure compiler compiles the signature of F into a tree of unique stamps Σ_i, that enable the functor entry points to identify their arguments.

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, such as our example functor F which seal $M_1.v_1$, 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 an ModuleML context can, we introduce a functor application entry point into the protected memory that has type: $\texttt{fappl} : \mathsf{Sec}[\textbf{functor}(\mathsf{X}_i : \mathsf{S}) \to \mathsf{S}'] \to (\mathsf{Ins}[\mathsf{S}] \vee \mathsf{Sec}[S]) \to \mathsf{Sec}[\mathsf{S}']$. 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 that the flag L_f is set. If not abort.
2. $f = Marshall_{\mathbf{i}}^{\textbf{functor}(\mathsf{X}_i:\mathsf{S})\to\mathsf{S}'}(i)$.
3. Depending on the representation of m:
   (a) If $\mathsf{Ins}[\mathsf{S}]$: $a = Marshall_{\mathbf{c}}^{\mathsf{S}}(a)$
   (b) If $\mathsf{Sec}[\mathsf{S}]$: $a = Marshall_{\mathbf{i}}^{\mathsf{S}}(a)$
4. Apply f to a. Store the result in r.
5. Stamp r with Σ_f.
6. Return $Marshall_{\mathbf{o}}^{S'}(r)$.

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 Besides enabling the A+I context to apply a functor, the secure compilation scheme also 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 $Marshall_{\mathbf{o}}^{S}$, 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. $d = Marshall_{\mathbf{i}}^{\mathsf{S}}(i)$.
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 $\texttt{F.fd}$, $\texttt{F.F}_1$ and $\texttt{F.M}_1$, 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
 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 $F_b.F_1$ as the implementation of $F_b.F_1$ exposes the value binding $A.v_s$, as highlighted in red in the listing for F_b.

The entry points for $F.F_1$ and $F.M_1$ 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.

$Marshall_{\mathbf{c}}^{\textbf{functor}(X_i:S)\to S'}$ 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 $Marshall_{\mathbf{c}}^{\textbf{functor}(X_i:S)\to S'} : \mathsf{Ins}[\textbf{functor}(X_i : S) \to S'] \to (\textbf{functor}(X_i : S) \to S')$, 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. $a = Marshall_{\mathbf{o}}^{S}(M)$
2. Apply f to a. Store the result in r.
3. Return $Marshall_{\mathbf{c}}^{S'}(r)$

4 Compiler Reflection

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

$$M_1 \simeq M_2 \Rightarrow M_1^{\downarrow} \simeq M_2^{\downarrow}$$

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: $M_1^{\downarrow} \not\simeq M_2^{\downarrow} \Rightarrow M_1 \not\simeq M_2$. 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:

Theorem 1 (Module Differentation). *Any two ModuleML modules m_1 and m_2 whose compilation results produce two different low-level traces $\overline{\alpha_1}$ and $\overline{\alpha_2}$ are not contextually equivalent. Formally:* $\mathsf{Traces}(m_1^{\downarrow}) \neq \mathsf{Traces}(m_2^{\downarrow}) \Rightarrow m_1 \not\simeq m_2$.

To prove the theorem we adopt the established proof technique [19,9] of developing an algorithm that given two modules m_1 and m_2 and their differing low level traces $\overline{\gamma_1}$ and $\overline{\gamma_2}$ can produce a "witness" context C that can distinguish between m_1 and 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 m_1 and m_2.

For the algorithm to be correct, it must detect the first two labels γ in the traces that differ. From Section 2.2, those labels can be: call, return, callback, returnback, write, read and termination. Because we have assumed that the execution starts outside of the compiled module (Section 3), the actions that appear at even-numbered positions in a trace will be calls or returnbacks from the A+I context, whereas actions that appear at odd-numbered positions will be returns or callbacks, generated by the ModuleML modules m_1 or m_2. 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 m_1 and terminate for m_2, distinguishing them as required.

Both in the implementation and in the examples that follow the low-level traces are adapted to be more understandable. For example, if the entry point of a value binding v_1 is located at `0x234`, the low-level label *call* `0x234` will be written as *call* v_1. This abstraction is safe, as it does not introduce additional information, it merely makes the traces more readable.

The algorithm first type checks the input modules. If the types differ it will produce a witness as detailed previously. If they are the same the algorithm proceeds. The algorithm starts with the following witness context:

```
struct
 open M
 val vc = ref 0
 val vd = (fun x : bool =
  letrec div : bool → bool =
  (fun x : bool = (div x)) in (div x))
 val vm = unit
end ;; vm
```

The *open M* statement includes the modules m_1 and m_2 as a module M, thus implementing the hole of the witness. The value binding v_m is of type **`Unit`** and serves as the starting point of the witness. The witness needs to keep track of

[3] http://bit.ly/1I3MJvy

the index that it replicates its, this is done by updating the value binding V_c, which holds a reference to the count. When the witness is able to differentiate two actions of the input modules it must diverge in one case and terminate in the other. Divergence is implemented through a diverging boolean function in the value binding v_d.

As the algorithm iterates through the traces it expands the witness as needed, using the signature obtained from the type check as its knowledge base for the input modules. In what follows we provide two examples. The first example illustrates how the algorithm reproduces the actions of traces that apply functions of the assembly context to closures. The second example illustrates how the algorithm reproduces the actions of traces that apply structures of the assembly context to secure functors. For more examples, such as of traces that are of different length or traces that return different locations we defer to the implementation.

Different CallBack Argument Consider the trace $\overline{\gamma_1}$:

$$call\ v_1? \cdot ret\ 1! \cdot call\ \texttt{apply}(1, \texttt{0x12})? \cdot call\ \texttt{0x12}(4)! \cdot ret\ 6? \cdot call\ \texttt{0x12}(4)!$$

where `apply` is the closure application entry point and `0x12` is an address of the assembly context. The second trace $\overline{\gamma_2}$ is identical except for the last action which is *call* `0x12`(5)!. The input modules are type as:

```
sig v_1 : int → int → int end
```

The algorithm reproduces action 0 of the trace by having the witness call v_1 from the starting point v_m and store the result in a variable x. internally the algorithm also rebuilds the table $\mathcal{N}$ associating the index 1 with the variable x and the type of v_1. To reproduce action 2 of the trace the algorithm must infer what the address `0x12` points to. By looking up the index 1 in $\mathcal{N}$ it the algorithms deduces that argument must be function within the witness of type **int** → **int**. Given no other information the algorithm will start by constructing a value binding v_{0x12} for a λ-term that returns 0 (the default value for **int**) and reproduces action 2 by applying x to v_{0x12}. When it reads action 3 the algorithm switches from implementing the value binding v_m to implementing the value binding v_{0x12} and simply has it return 5 when the witness step count is 2 to reproduce action 4. The final action differs in that the traces call v_{0x12} with different arguments. In this case the witness simply diverges if the argument is 4 otherwise it exits with value 0. The resulting witness is listed below:

```
struct ...
 val v_0x12 = (fun y:int =
  (if (== !v_c 2)then incr_v_c; 5
   else (if (== !v_c 3) then (if(== y 4) (v_d true)
   else (exit 0)) else 0))
 val v_m = incr_v_c;
  (let x = M.v_1 in incr_v_c; (x v_0x12))
end ;; v_m
```

Where $incr_v_c$ is short for $v_c := (+\ !v_c\ 1)$ and M is the module that fills the hole. Note that within main we do not check the current step count as the secure compiler does not produce code that calls the main function of the context.

Different Return From A Dynamic Module Consider the following trace $\overline{\gamma_1}$:

$$call\ X_1? \cdot ret\ 1! \cdot call\ \texttt{applyf}(1, \texttt{0xa})? \cdot read(\texttt{0x8a}, R_a)\ .write(\texttt{0xa}, R_r).ret\ 1! \\ \cdot call\ V_1(1)? \cdot ret\ false!$$

where *applyf* is the functor application entry point, and R_a is a record of a structure of the assembly context and R_r is a record returned by the modules as discussed in Section 3.6. The second trace $\overline{\gamma_2}$ is identical except for the last action which is *ret* **true**!. The input modules are typed as:

```
sig module X₁ : functor(Xᵢ : sig val v₁: int end) → (sig v₁: bool) end
```

The algorithm deduces from the signature that X_1 is a functor. The algorithm rebuilds the table $\mathcal{F}$ internally by associating X_1 with the index 1 and has the witness increase the step count by one. Note that it follows from Section 3.6 that action 1 will always return the same index for modules of the same signature, if that were not the case we would not be able to replicate a low-level call to a functor. To reproduce action 2 the algorithm deduces by looking up index 1 in $\mathcal{F}$ that it must build a module X_{0xa} that corresponds to the signature of X_i and apply it to X_1. Like functions modules are first constructed with default values based on the types and then expanded upon later. The algorithm reproduces action 2 by applying X_{0xa} to X_1 and taking another empty step. To reproduce action 4 the algorithm calls the value binding V_1 on the result of the previous application. The final action differs between $\overline{\gamma_1}$ and $\overline{\gamma_2}$ in that they return different booleans. The witness thus simply diverges as it did in the previous example. The resulting witness is listed below:

```
struct ...
 module Xₐ = struct val v₁ = 0 end
 module Xᵣ = M.X₁(X₁)
 val vₘ = incr_v_c;incr_v_c;
  let x = Xᵣ.v₁ in incr_v_c; (if (== x false)
   then (v_d true) else (exit unit))
end ;; vₘ
```

Proof Sketch. The proof analyses all possible differences between $\mathsf{Traces}(m_1^{\downarrow})$ and $\mathsf{Traces}(m_2^{\downarrow})$ and proves that the output of the algorithm differentiates between the ModuleML modules m_1 and m_2 through examples. □

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* compilation scheme that features none of the security checks. The compiler targets the Fides implementation of PMA [23]. Fides implements PMA through use of a hypervisor that runs two virtual machines: one that handles the secure memory module and one handles the outside memory. One consequence of this architecture is that, as the low-level context interacts with the 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 the table below.

	Unprotected	**Fides**	**Fides & Secured**
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

This table details the average execution time without protection, with Fides and with Fides as well as the security checks introduced by the compilation scheme . 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 "UnProtected" and "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 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.

[4] http://bit.ly/1I3MJvy

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.* [7]. Their definition of ML, however, does not feature a module system. Their Javascript attacker model is also more high-level then then 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.* [8]. 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 [3,13]. The resulting compilers thus come with proofs for the preservation property that we have assumed (Section 1). Many established verified compilation results, however, hold only under a closed world assumption. While our secure compilation scheme considers arbitrary contexts in the target language A+I, in verified compilation works the target language contexts are often assumed to be obtained through compilation, they thus do not misbehave. An exception to that limitation is the work by Perconti *et. al.* on verifying a compiler that does not compile whole programs [21]. That work, however, targets a typed assembly language in contrast to the untyped A+I.

Throughout the secure compilation scheme we make use of our previously developed interaction counting masking system [10] to securely share the values of security relevant abstractions. Alternatively, we could have applied the encrypting mechanism of Matthews et al. [15], to achieve the same result. The masking mechanism of Patrignani *et al.* [19], however, cannot be used as its identification of equal objects would break the equivalences of ModuleML.

A different line of research focusses on developing security architectures with access control mechanisms comparable to PMA: Flicker [16], Fides [23], Sancus [18] and the Intel SGX [17]. The existence of industry prototypes alongside research ones underlines the feasibility of bringing efficient and secure low-level memory access control to commodity hardware. Besides the secure compilation works of Patrignani and Agten *et al.* [19] for Fides-like PMA architectures, no results comparable to ours have been proven for these systems.

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. P. Agten, R. Strackx, B. Jacobs, and F. Piessens. Secure compilation to modern processors. In *2012 IEEE 25th Computer Security Foundations Symposium*, CSF 2012. IEEE, 2012.
3. 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.
4. P. Codognet and D. Diaz. wamcc: Compiling Prolog to C. In *ICLP*, pages 317–331. MIT PRess, 1995.
5. D. Dreyer. *Understanding and Evolving the ML Module System.* PhD thesis, Carnegie Mellon, May 2005.
6. M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2):235–271, Sept. 1992.
7. 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.
8. R. Jagadeesan, C. Pitcher, J. Rathke, and J. Riely. Local memory via layout randomization. In *CSF '11*, pages 161–174. IEEE, 2011.
9. A. Jeffrey and J. Rathke. A fully abstract may testing semantics for concurrent objects. *Theor. Comput. Sci.*, 338(1-3):17–63, 2005.
10. A. Larmuseau and D. Clarke. Formalizing a secure foreign function interface. In *SEFM*, LNCS, 2015. To appear, currently at: https://db.tt/y87tcQ0V.
11. X. Leroy. Manifest types, modules, and separate compilation. In *POPL '94*, pages 109–122, New York, NY, USA, 1994. ACM.
12. X. Leroy. Applicative functors and fully transparent higher-order modules. In *POPL '95*, pages 142–153, New York, NY, USA, 1995. ACM.
13. X. Leroy. Formal verification of a realistic compiler. *CACM*, 52(7):107–115, 2009.
14. X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vôuillon. The Objective Caml system, release 4.02. Technical report, INRIA, August 2014.
15. J. Matthews and A. Ahmed. Parametric polymorphism through run-time sealing or, theorems for low, low prices! In *ESOP 2008*, pages 16–31, 2008.
16. J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: an execution infrastructure for TCB minimization. *SIGOPS Oper. Syst. Rev.*, 42(4):315–328, 2008.
17. 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.
18. J. Noorman et al. Sancus: Low-cost trustworthy extensible networked devices with a zero-software Trusted Computing Base. In *USENIX*, pages 479–498, 2013.
19. 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.
20. M. Patrignani and D. Clarke. Fully Abstract Trace Semantics of Low-level Isolation Mechanisms. In *SAC '14*, pages 1562–1569. ACM, 2014.

21. J. T. Perconti and A. Ahmed. Verifying an open compiler using multi-language semantics. In *ESOP*, pages 128–148, 2014.
22. C. Queinnec. *Lisp in small pieces.* Cambridge University Press, 2003.
23. R. Strackx and F. Piessens. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *CCS*, pages 2–13, 2012.
24. E. Sumii and B. C. Pierce. A bisimulation for dynamic sealing. In *POPL '04*, pages 161–172, New York, NY, USA, 2004. ACM.