On Modular and Fully-Abstract Compilation

Marco Patrignani
MPI-SWS Saarbrücken, Germany
name.surname@mpi-sws.org

Dominique Devriese   Frank Piessens
iMinds-Distrinet, KU Leuven, Belgium
name.surname@cs.kuleuven.be

***Abstract*—Secure compilation studies compilers that generate target-level components that are as secure as their source-level counterparts. *Full abstraction* is the most widely-proven property when defining a secure compiler.**

A compiler is *modular* if it allows different components to be compiled independently and then to be linked together to form a whole program.

Unfortunately, many existing fully-abstract compilers to untyped machine code are not modular. So, while fully-abstractly compiled components are secure from malicious attackers, if they are linked against each other the resulting component may become vulnerable to attacks.

This paper studies how to devise modular, fully-abstract compilers. It first analyses the attacks arising when compiled programs are linked together, identifying security threats that are due to linking. Then, it defines a compiler from an object-based language with method calls and dynamic memory allocation to untyped assembly language extended with a memory isolation mechanism. The paper provides a proof sketch that the defined compiler is fully-abstract and modular, so its output can be linked together without introducing security violations.

This paper uses colours to distinguish elements of different languages; please print this in colour.

I. Introduction

A compiler is a tool that, among other things, translates programs written in a source language into programs written in a target language. A compiler is *secure* when it preserves all security properties of the components (i.e., partial programs) it inputs in the components it outputs. Secure compilation studies compilers that generate target-level components that are as secure as their source-level counterparts. Full abstraction is the most widely-adopted property to prove when defining a secure compiler [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11]. A fully-abstract compiler preserves and reflects observational equivalence between the source components it inputs and the target components it produces. Observational equivalence captures security properties such confidentiality, integrity, etc, and since a fully-abstract compiler preserves all of them, it is a secure compiler.

A compiler is *modular* when it operates on components and its output can be *linked* together into larger components (and possibly into whole programs).

With the advent of machine-code level security architectures (e.g., protected modules architectures (*PMA*) [12], [13], [14], capability machines [15], [16], micro-policies enforcing architectures [17], address space layout randomisation [2], etc.) researchers have investigated secure (fully-abstract) compilation to such architectures [2], [3], [4], [18].

For secure compilation to *PMA* (informally, a low-level memory isolation mechanism), only compilation of a single protected module has been considered. Generalizing this to multiple modules, supporting modular compilation and linking of protected modules at machine code level, would be useful for a number of reasons: (i) code is easier to develop and distribute in standalone components and (ii) merging modules in a single one is not always possible nor desirable. Concerning (i), standard arguments for separate compilation apply. Particularly, it is more efficient to compile only the modules that changed and link it against the previously compiled other code as opposed to recompiling the entire code base. Moreover, it is common in practice and useful to use libraries that are built from safe source code but only available in compiled form. When targeting *PMA* with attestation capabilities, a benefit of having each module in a different protected module is that this makes it possible to attest them independently. Concerning (ii), there can be several reasons why merging more modules in a single one is not desirable; even with a secure compiler, it can still be useful to protect securely compiled components from each other. Consider a source language that allows programmers to write both safe and unsafe code at the same time (e.g., Rust, C# unsafe blocks or the language considered by Juglaret *et al.* [19]). A desirable property for a compiler for such a language is that unsafe code does not affect the safe one. In this case, one can foresee splitting the safe and the unsafe program parts in two different modules and using the secure compiler presented in this paper for the safe part. In this way, both the safe and the unsafe code cannot be directly tampered with by an attacker and the unsafe code can not affect the safe one. Another setting where merging multiple modules is undesirable is when a programmer provides a hand-optimised securely-compiled component that performs better than the ones obtained by a secure compiler, but with the same security guarantees. In this case, merging such a component with other ones in a single module seems undesirable, as the hand-made assembly can be error-prone and maliciously-crafted. However, by using a modular secure compiler, other components can still interact with the hand-optimised one and no one's security can be tampered.

Extending the compiler to support modular compilation is, however, surprisingly complicated for a number of reasons:

1) with a single protected module, all run-time meta information about the execution that needs to be protected (e.g., the call stack, or dynamic type information) can

be stored within that single protected module. For a modular compiler, that state must be divided over the various protected modules, or – possibly – stored in a centralized trusted protected module.

2) with a single protected module, object references are either private to the module or public. With support for multiple protected modules, object references can also be shared between some modules and still be unknown to other ones. This requires a mechanism for protecting object references that is more elaborate than what the existing secure compiler to *PMA* uses.

There are several interesting approaches to address these challenges. This paper investigates how these issues can be addressed constructing a modular fully-abstract compiler for *PMA* in the style of Intel SGX [14] *without* additional hardware support. Even if this turns out to be intricate, we believe there is value in this approach, as this kind of hardware support is available on commercially available systems today, whereas more advanced hardware support is only available in research prototypes.

An alternative approach to modular, secure compilation would be to investigate what kind of novel hardware security architectures can help in addressing these challenges. This is the approach taken by Juglaret *et al.* [18] and it can likely lead to a secure compilation scheme. An important downside is that it may take a while before such hardware extensions are available in mainstream systems.

The main contribution of this paper is $\llbracket \cdot \rrbracket^{\mathsf{JEM}}_{\mathsf{AIL}}$, a modular and fully-abstract compiler to SGX-like *PMA*. The source language of $\llbracket \cdot \rrbracket^{\mathsf{JEM}}_{\mathsf{AIL}}$ is Jem, an object-based imperative language with method calls and dynamic memory allocation. The target language of $\llbracket \cdot \rrbracket^{\mathsf{JEM}}_{\mathsf{AIL}}$ is Ail, untyped assembly language with an explicit linking mechanism extended with *PMA*. Jem, Ail and *PMA* are formalised in Section II. Explicitly considering linking between assembly components generates a number of previously-unobserved problems that are presented in Section III. The main contribution of this paper is the formalisation of $\llbracket \cdot \rrbracket^{\mathsf{JEM}}_{\mathsf{AIL}}$, presented in Section IV. Supporting additional language features is discussed in Section V. The proof sketch of $\llbracket \cdot \rrbracket^{\mathsf{JEM}}_{\mathsf{AIL}}$ being fully-abstract and modular is presented in Section VI. Finally, Section VII discusses related work and Section VIII concludes. Complete language formalisations, proofs, sketches and technical formalities can be found in the companion technical report [20].

II. Languages Formalisation

This section describes Jem (Section II-A) and Ail (Section II-B), respectively the source and the target language of the secure compiler $\llbracket \cdot \rrbracket^{\mathsf{JEM}}_{\mathsf{AIL}}$.

A. The Source Language Jem

Jem is a strongly-typed, single-threaded, object-based imperative language that has **private** fields and **public** methods; it does not allow any undefined behaviour to arise. Jem is presented in a *green* font (Figure 1).

$$
\begin{array}{lrl}
\textit{component} & \mathcal{C} ::= & \overline{C}\\
\textit{classes} & C ::= & \mathtt{import}\ \overline{I};\overline{X};\ \mathtt{class}\ c\{K\ \overline{F_t}\ \overline{M}\};\overline{O}\\
\textit{objects} & O ::= & \mathtt{object}\ o : t\{\overline{F}\}\\
\textit{class declarations} & I ::= & \mathtt{class\text{-}decl}\ c\{\overline{m : M_t}\}\\
\textit{object declarations} & X ::= & \mathtt{obj\text{-}decl}\ o : t;\\
\textit{methods} & M ::= & \mathtt{public}\ m(\overline{x}) : M_t\ \{\mathtt{return}\ E;\}\\
\textit{signatures} & M_t ::= & t(\overline{t}) \to t\\
\textit{fields} & F ::= & \mathtt{private}\ f = v\\
\textit{field types} & F_t ::= & f : t\\
\textit{constructors} & K ::= & c(\overline{f} : \overline{t})\ \{\mathtt{this}.\overline{f'} = \overline{f}\}\\
\textit{types} & t ::= & \mathtt{Unit} \mid \mathtt{Bool} \mid \mathtt{Int} \mid c \mid \mathtt{Obj}\\
\textit{values} & v ::= & \mathtt{unit} \mid \mathtt{true} \mid \mathtt{false} \mid n \mid o\\
\textit{operations} & \mathtt{op} ::= & + \mid - \mid == \mid \cdots\\
\textit{expressions} & E ::= & v \mid x \mid E.f \mid E.f = E \mid E.m(\overline{E})\\
 & & \mid E\ \mathtt{op}\ E \mid \mathtt{new}\ t(\overline{E}) \mid E;E \mid \mathtt{this}\\
 & & \mid \mathtt{if}\ (E)\ \{E\}\ \mathtt{else}\ \{E\} \mid \mathtt{exit}\ E\\
 & & \mid \mathtt{instanceof}(E : c) \mid \mathtt{var}\ x : t = E
\end{array}
$$

Figure 1. Syntax of JEM; lists of elements $a_1 \cdots a_n$ are denoted as $\overline{a}$.

A class C declares (external) classes and objects it requires (these are called **import** requirements) then it defines its constructor, fields, methods and objects implementing that class. Objects of a class can only be allocated by methods of that class (so cross-component memory allocation happens via factory methods). Class declarations I define class signatures, i.e., the class name and the methods implemented by that class. Object declarations X are references to objects implementing a different class. A Jem component $\mathcal{C}$ is a collection of classes $\overline{C}$. If all **import** requirements of $\overline{C}$ are satisfied by some other class in $\overline{C}$, then $\overline{C}$ is a whole program. Two components $\mathcal{C}_1$ and $\mathcal{C}_2$ satisfy each other, denoted with $\mathcal{C}_1 \frown \mathcal{C}_2$, if all **import** requirements of $\mathcal{C}_1$ are classes and objects in $\mathcal{C}_2$ and vice-versa.

The top of the Jem class hierarchy is **Obj**, a class defining no methods. All classes implicitly extend **Obj**; Jem does not provide any other form of inheritance. Primitive types are **Unit**, inhabited by `unit`, **Bool**, inhabited by `true` and `false` and **Int**, inhabited by natural numbers n. Identifiers for classes c, objects o, methods m, fields f and variables x are taken from distinct denumerable sets.

The semantics of Jem is standard and unsurprising; it is omitted for space reasons.

The security mechanism of Jem is given by **private** fields, which can be used to define security properties such as confidentiality and integrity (as defined in Section III-A).

1) Contextual Equivalence for Jem*:* To reason about the behaviour of Jem components, contextual equivalence is used [21]. Contextual equivalence is the coarsest relation that tells when two components are behaviourally equivalent; its definition (Definition 1) is rather standard [2], [3], [4], [5], [6], [7], [11], [22], [23].

Contexts $\mathbb{C}$ are partial programs with a hole ($[\cdot]$), formally $\mathbb{C} ::= \overline{\mathcal{C}}[\cdot]$. The plugging of a component $\mathcal{C}$ in a context $\mathbb{C}$, denoted as $\mathbb{C}[\mathcal{C}]$, returns a whole program $\mathbb{C}; \mathcal{C}$. There are two (common) assumptions for the plugging to succeed. The first one is that $\mathcal{C} \frown \mathbb{C}$, the second one is that $\mathcal{C}$ and $\mathbb{C}$ are well-typed. If any assumption is not upheld, the plugging returns the empty program.

Definition 1 (Contextual equivalence for JEM). $\mathcal{C}_1 \simeq_{ctx} \mathcal{C}_2 \triangleq \forall \mathbb{C}, \mathbb{C}[\mathcal{C}_1]\Uparrow \iff \mathbb{C}[\mathcal{C}_2]\Uparrow$, where $\Uparrow$ means divergence, i.e., the execution of an unbounded number of reduction steps.

B. PMA and the Target Language AIL

AIL (acronym of Assembly plus Isolation and Linking) is a low-level language that models a von Neumann machine enhanced with *PMA* (Section II-B1), with an idealised form of cryptographic nonces (Section II-B2) and with an explicit linking mechanism (Section II-B3). AIL is the adaptation of an analogous language that had no linking mechanism, no idealised cryptographic nonces and a single *PMA* module [4], [24]. After presenting *PMA* and AIL, this section describes the semantics (Section II-B4) and contextual equivalence for AIL (Section II-B5).

1) PMA: PMA is an assembly-level isolation mechanism based on program counter-based memory access control. *PMA* can be implemented in software (e.g., via a hypervisor) [12], [25], [26] or in hardware [13], [14], [27]; Intel is bringing it to mainstream processors with the Intel SGX instruction set [14]. *PMA* logically divides the memory space into several protected and one unprotected section; a protected section is called *protected module*, each module has a unique *module id*. All protected sections are further divided into a *code* and a *data* section. Code sections contain a variable number of *entry points*: the only protected addresses to which instructions in unprotected memory or in other protected sections can jump. Entry points allow code from within the module to interoperate with external code. Data sections are only accessible from within the protected code section of the same module. The table below summarises the *PMA* access control model.

From \ To	Unprotected	Protected		
		Entry Point	Code	Data
Unprotected	r w x	x		
Protected	r w x	Same id		
		r x	r x	r w
		Different id		
		x		

2) Core AIL*:* AIL is presented in a *pink* font (Figure 2).

Words w are either natural numbers n (including instruction encodings and module ids), symbols σ (explained in Section II-B3) or symbolic nonces π. Addresses a are pairs of natural numbers and module ids, so a module with id id has an infinite memory starting from address $(id, 0)$. Unprotected memory has id 0. The registers file r contains an unbounded number of registers $\mathtt{r}$. The flags register f contains a sign (SF) and a zero flag (ZF) that are set by arithmetic and testing instructions. Memories m are lists of bindings from addresses to words. Module descriptors s are triples that define a module memory layout: id indicates the id of the module, n_c is the length (in number of addresses) of the code section and n is the number of entry points. Between each entry point there are a fixed amount of addresses that are not entry points, indicate this number of addresses with $\mathcal{N}_w$. The first entry point is located at address 0, the second one at address $\mathcal{N}_w$ and so on; the last one is located at address $n \cdot \mathcal{N}_w$ such that $n \cdot \mathcal{N}_w < n_c$. Symbolic nonces π are non-guessable, unforgeable tokens; they model what a program could create using cryptographic primitives or unguessable randomisation [28], [29]. For the sake of simplicity, they are considered to be 0 for arithmetic operators.[1] Nonce oracles h are used as suppliers of fresh nonces for the `new` instruction (see Rule Eval-new in Figure 4). Instructions i are elements of the set $\mathcal{I}$ (Table I) that define the programming language executed on the architecture.

numbers	$n ::= n \in \mathbb{N}$
words	$w ::= n \mid \sigma \mid \pi$
addresses	$a ::= (id, n)$
memories	$m ::= \overline{a \mapsto w}$
flags	$f ::= \mathsf{ZF} \mapsto 0 \mid 1; \mathsf{SF} \mapsto 0 \mid 1$
register files	$r ::= \overline{\mathtt{r} \mapsto w}$
module ids	$id \in \mathcal{ID} \subset \mathbb{N}$
symbolic nonces	$\pi \in \Pi$
module descriptors	$s ::= (id, n_c, n)$
nonce oracles	$h ::= \overline{\pi}$
symbols	$\sigma, \iota \in \mathcal{S}$
instructions	$i \in \mathcal{I} \subset \mathbb{N}$

Figure 2. Formalisation of AIL (part 1: language).

3) Modules, Programs and Linking (Figure 3): To deal with linking, AIL has symbols σ, ι. Symbols in a module are placeholders for words that will be filled when the module is linked against another one; symbols can be found in memory and in the symbol table. A symbol table t contains exported method and object bindings as well as required method and object bindings. An exported method binding EM (resp. object binding EO) maps a method name m and its type M_t (resp. an object name o and its class c) to the address a where the method (resp. class) is located. A required method binding RM (resp. object binding RO) maps a method name m and its type M_t (resp. an object name o and its class c) to the id symbol ι and the number symbol σ used for it. A method (resp. object) export binding fulfils a method (resp.

[1] Having arithmetic operations affect nonces would only let us model guessing attacks on them, but they are assumed to be resilient to these attacks so we chose this option for simplicity.

$\texttt{movl}\ \texttt{r}_\texttt{d}\ \texttt{r}_\texttt{s}\ \texttt{r}_\texttt{i}$	Load the word from the address in registers ($\texttt{r}_\texttt{s}$,$\texttt{r}_\texttt{i}$) into register $\texttt{r}_\texttt{d}$.
$\texttt{movs}\ \texttt{r}_\texttt{d}\ \texttt{r}_\texttt{s}\ \texttt{r}_\texttt{i}$	Store the contents of register $\texttt{r}_\texttt{s}$ at the address found in registers ($\texttt{r}_\texttt{d}$,$\texttt{r}_\texttt{i}$).
$\texttt{movi}\ \texttt{r}_\texttt{d}\ \texttt{k}$	Load the constant value $\texttt{k}$ into register $\texttt{r}_\texttt{d}$.
$\texttt{add}\ \texttt{r}_\texttt{d}\ \texttt{r}_\texttt{s}$	Write $\texttt{r}_\texttt{d} + \texttt{r}_\texttt{s}$ into register $\texttt{r}_\texttt{d}$ and set the ZF flag accordingly.
$\texttt{sub}\ \texttt{r}_\texttt{d}\ \texttt{r}_\texttt{s}$	Write $\texttt{r}_\texttt{d} - \texttt{r}_\texttt{s}$ into register $\texttt{r}_\texttt{d}$ and set both the ZF and the SF flags accordingly.
$\texttt{cmp}\ \texttt{r}_\texttt{s}\ \texttt{r}_\texttt{d}$	Compare $\texttt{r}_\texttt{s}$ and $\texttt{r}_\texttt{d}$ and set the flags accordingly.
$\texttt{jmp}\ \texttt{r}_\texttt{d}\ \texttt{r}_\texttt{i}$	Jump to the address located in register $\texttt{r}_\texttt{d}$ in the module with id $\texttt{r}_\texttt{i}$.
$\texttt{je}\ \texttt{f}_\texttt{i}\ \texttt{r}_\texttt{i}$	If flag $\texttt{f}_\texttt{i}$ is set, jump to the address in register $\texttt{r}_\texttt{i}$ in the current module.
$\texttt{zero}$	Set all registers to 0.
$\texttt{new}\ \texttt{r}_\texttt{d}$	Initialise register $\texttt{r}_\texttt{d}$ with a fresh symbolic nonce.
$\texttt{halt}$	Stop the execution.

Table I
INSTRUCTION SET $\mathcal{I}$.

$$\frac{p = (id, n) \qquad m(p) = \texttt{new}\ \texttt{r}_\texttt{d} \qquad r' = r[\texttt{r}_\texttt{d} \mapsto \pi] \qquad \pi \notin h}{(p, r, f, m, s, \pi; h) \xrightarrow{id}\!\!\!\twoheadrightarrow ((id, n+1), r', f, m, s, h)}\ \text{(Eval-new)}$$

$$\frac{(p, r, f, m, s, h) \xrightarrow{id}\!\!\!\twoheadrightarrow (p', r', f', m', s', h') \quad p \vdash \mathsf{currentModule}(\overline{s}, s) \qquad p = (id, n)}{(p, r, f, m, \overline{s}, h) \twoheadrightarrow (p', r', f', m', \overline{s}, h')}\ \text{(Eval-module)}$$

$$\frac{p = (id, n) \qquad m(p) = (\texttt{movs}\ \texttt{r}_\texttt{d}\ \texttt{r}_\texttt{s}\ \texttt{r}_\texttt{i}) \quad \overline{s} \vdash \mathsf{writeAllowed}(n, (r(\texttt{r}_\texttt{d}), r(\texttt{r}_\texttt{i}))) \quad m' = m[(r(\texttt{r}_\texttt{d}), r(\texttt{r}_\texttt{i})) \mapsto r(\texttt{r}_\texttt{s})]}{(p, r, f, m, \overline{s}, h) \twoheadrightarrow ((id, n+1), r, f, m', \overline{s}, h)}\ \text{(Eval-movs)}$$

$$\frac{p = (id, n) \qquad m(p) = (\texttt{jmp}\ \texttt{r}_\texttt{d}\ \texttt{r}_\texttt{i}) \qquad n' = r(\texttt{r}_\texttt{d}) \quad id' = r(\texttt{r}_\texttt{i}) \qquad r' = r'[\texttt{r}_0 \mapsto id] \qquad \overline{s} \vdash \mathsf{validJump}(p, (id', n'))}{(p, r, f, m, s, h) \twoheadrightarrow ((id', n'), r', f, m, s, h)}\ \text{(Eval-jmp)}$$

Figure 4. An excerpt of the dynamic semantics of AIL

object) requirement if both bind the same method name and type (resp. object name and class).

$$\begin{array}{llll}
\textit{modules} & M ::= (m; s; t) & \textit{programs} & P ::= (m; \overline{s}; t) \\
\textit{symbol tables} & & & t ::= \overline{EM}; \overline{EO}; \overline{RM}; \overline{RO} \\
\textit{exported methods} & EM ::= m : M_t \mapsto a & & \\
\textit{exported objects} & EO ::= o : c \mapsto n & & \\
\textit{required methods} & RM ::= m : M_t \mapsto \iota; \sigma & & \\
\textit{required objects} & RO ::= o : c \mapsto \sigma & &
\end{array}$$

Figure 3. Formalisation of AIL (part 2: linking).

A single AIL module M is a triple listing a memory m, its module descriptor s and a symbol table. An AIL program P is a collection of modules whose module descriptors are listed in $\overline{s}$. If the required bindings of a program are empty lists, then the program is whole. When a program is not whole it is partial, i.e., it is a collection of modules with a symbol table with unfulfilled requirements. Informally, two AIL programs P_1 and P_2 satisfy each other, denoted with $P_1 \frown P_2$, if all the required bindings of P_1 are fulfilled by a binding in P_2 and vice-versa. Modules and programs are well-formed if the only symbols in their memory are those captured in the symbol table. In the following, we only consider well-formed programs and modules.

Two memories (resp. two module descriptors) agree if their domains are disjoint (resp. if their ids are distinct). Two AIL modules P_1 and P_2 can be joined together, denoted with $P_1{+}P_2$, only if the two memories and if the memory descriptors agree. Joining of modules results in a program that is the concatenation of the memories, memory descriptors and symbol tables of P_1 and P_2. Joining of symbol tables results in a new symbol table. If a requirement (method or object) is fulfilled by a requirement in another table, that requirement is removed from the resulting table. Moreover, in the resulting memory, all symbols of fulfilled requirements are replaced with what is bound by the fulfiller requirement. Consider P_1 to require method $m : M_t \mapsto \iota; \sigma$ and P_2 to export method $m : M_t \mapsto (id, n)$. When merging P_1 and P_2, all occurrences of ι and σ in the memory of P_1 will be replaced with id and n, respectively. The resulting symbol table will no longer have the requirement for $m : M_t$ as that has been fulfilled.

4) Dynamic Semantics of AIL*:* The dynamic semantics of whole programs ($\twoheadrightarrow$) relates program states (p, r, f, m, s, h) where p is the program counter (i.e., an address). This semantics relies on another one for modules ($\xrightarrow{id}\!\!\!\twoheadrightarrow$); the latter tells when a module with id id performs a reduction (Rule Eval-module). Figure 4 shows an excerpt of the reduction rules. Memory access is indicated as: $m(a) = w$; memory update is indicated as $m[a \mapsto w]$. The same notation is adopted for register files and flags register access and update. Assumptions of the form $\overline{s} \vdash \mathsf{name}(a, a')$ model the enforcement of the *PMA* access control policy; their names are self-explanatory. For example, $s \vdash \mathsf{writeAllowed}(n, n')$ tells if an address n' can be written from another address n and $\overline{s} \vdash \mathsf{validJump}(a, a')$ tells if address a' is executable (x) from address a. $p \vdash \mathsf{currentModule}(\overline{s}, s)$ tells whether the program counter p is in a specific module whose descriptor is s where s is within $\overline{s}$. Instruction $\texttt{jmp}$ is the only instruction that can perform cross-module jumps (Rule Eval-jmp). When a cross-module jump is performed, the *PMA* architecture inserts the id of the caller in register $\texttt{r}_0$. This functionality is called caller-callee authentication; some *PMA* implementations provide it [12], [13] and it is achievable on Intel SGX by means of the EREPORT instruction [30]. Whenever the execution of an instruction violates the *PMA* access control policy (e.g., jumping to a module address that is not an entry point), the execution is terminated.

5) Contextual Equivalence for AIL*:* Analogously to JEM, to reason about AIL components we define when they are contextually equivalent. Contexts $\mathbb{P}$ are partial programs with a hole ($[\cdot]$) that can be filled with a component in order to create a whole program, formally $\mathbb{P} ::= P[\cdot]$. Plugging a component P in a context $\mathbb{P}$, denoted as $\mathbb{P}[P]$, returns a whole program $\mathbb{P}{+}P$, given that $P \frown \mathbb{P}$.

Contextual equivalence for languages with oracles relies on a notion of contextual preorder [31] which is needed to ensure obvious equivalences are satisfied (Example 1).

Example 1 (Need for preorders)**.** *Consider a module P_{n2} that generates two nonces and returns only one and another one P_{n1} that generates and returns one nonce only. These programs are intuitively equivalent, but a contextual equivalence definition analogous to that for* Jem *simply does not yield this fact because we cannot quantify over all oracles and use the same oracle for both programs. Given that P_{n1} runs with an oracle h_1, P_{n2} has the same behaviour if it runs with an oracle h_2 that has all elements of h_1 interleaved with fresh elements. Dually, given that P_{n2} runs with an oracle h_2, P_{n2} has the same behaviour if it runs with an oracle h_1 that has only the even-numbered elements of h_2.* ⊡

Definition 2 (Contextual preorder for Ail)**.** $P_1 \underset{\sim}{\sqsubset} P_2 \triangleq \forall \mathbb{P}, h_1, \exists h_2 . \mathbb{P}[P_1], h_1 \Uparrow \Rightarrow \mathbb{P}[P_2], h_2 \Uparrow$, where $\mathbb{P}[P], h$ indicates the initial state of $\mathbb{P}{+}P$ with oracle h.

Definition 3 (Contextual equivalence for Ail)**.** $P_1 \simeq_{ctx} P_2 \triangleq$ $P_1 \underset{\sim}{\sqsubset} P_2$ and $P_2 \underset{\sim}{\sqsubset} P_1$.

III. Security Pitfalls

This section describes the threat model considered in this paper (Section III-A) and the security pitfalls that arise when linking is explicitly considered (Section III-B).

A. The Threat Model

The goal of this section is to familiarise the reader with the security aspects of secure compilation that are relevant for this paper. We do not provide an in-depth analysis of these aspects as they are not the focus of this paper, though we believe these insights can be helpful for many readers.

Secure compilation papers generally consider a threat model for an attacker with target-level code injection capabilities that operates, for example, by exploiting a bug in the system and injecting or loading arbitrary target programs. His goal is to violate the security properties of compiled programs found in the system. Specifically, in this paper we consider an attacker that can load arbitrary malicious code.

In this paper, the system is the von Neumann machine formalised by the Ail language, where multiple protected modules are found. The compiled programs that the attacker wants to violate span *some* of the protected modules, as explained in Section I; their security properties must not be violated. The attacker is modelled as code (and data) that spans unprotected memory as well as *other* protected modules; thus the attacker is modelled as an Ail context. This attacker can interact with any of the compiled programs whose security is of interest in any way that the assembly language (and the *PMA* access control policy) allow him. Thus, the attacker must respect the processor-enforced *PMA* access control policy and, most importantly, he cannot tamper with it.

We are interested in program security properties that can be expressed by means of contextual equivalence; some examples include confidentiality and integrity. A value is confidential if it cannot be discerned by other components than the one declaring it. In other words, a value v in a component $\mathcal{C}$ is confidential if $\mathcal{C}$ is contextually-equivalent to $\mathcal{C}'$ which is $\mathcal{C}$ with a different value for v. Integrity of a value means that it cannot be modified by other components than the one declaring it. In other words, a value v in a component $\mathcal{C}$ has integrity if $\mathcal{C}$ is contextually-equivalent to $\mathcal{C}'$ which is $\mathcal{C}$ where every interaction with other component is followed by a check that the value of v is the same as before the interaction.

The goal of the paper is to provide a compiler $[\![\cdot]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$ that produces secure Ail modules. Thus, $[\![\cdot]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$ takes Jem components that possibly has some security properties and outputs Ail modules that has the same security properties. If this holds, then the attacker's goal is nullified.

As compiler full-abstraction is preservation and reflection of contextual equivalence, proving $[\![\cdot]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$ to be fully-abstract implies that $[\![\cdot]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$ is secure.

B. Security Problems Related to Linking

Problems 1 to 4 are full-abstraction violations that arise in the presence of linking. Any full-abstraction violation can be lead back to confidentiality or integrity violations.

Problem 1 (Object id guessing)**.** *Consider an object allocated at address a in module M_1 and that is only shared between two modules M_1 and M_2. As object ids are just addresses in memory, nothing prevents another module M_3 to guess address a and call methods on it. Note however, that this violates the intended confidentiality of a, that is supposed to be visible only to M_1 and M_2.*

A naïve solution to this problem is tracking which module will receive a certain object id in order to detect and stop guesses from modules that have not received that object id. However, this does not scale to the case where M_2 forwards a to M_3. Since the forwarding is done outside of M_1, M_1 has no way of adding M_3 to the modules that are allowed to access a. As this may not be known statically, a different solution is needed.

To address this concern, target-level object ids must be unguessable. ■

Problem 2 (Call stack shortcutting)**.** *As all assembly languages,* Ail *calls and returns are jumps to addresses in memory. Consider the following sequence of cross-module function calls: $M_2 \rightarrow M \rightarrow M_1 \rightarrow M'$, where primes are used to denote subsequent calls to the same module.*

M' can return to M_2, as it learnt the address to return there when M_2 called it. However, that instruction bypasses the rest of the call stack and particularly M_1 in a way that is not possible in the source language, where control flow follows a well-bracketed sequence of calls/returns.

To address this concern, securely compiled code must ensure a well-bracketed traversal of the stack. ■

Problem 3 (Types of objects in other modules)**.** *Consider an object of type t' allocated at address a in module M_1 and a method m compiled in a different module M_2 that takes a parameter of type t. t and t' are different. A third module M_3 could pass a as an argument to the compilation of m, violating the* Jem *assumption that only well-typed programs are executed. A dynamic check on the type of a should be made, but the code of m resides in M_2 and it has no way to access a, which resides in module M_1.*

To address this concern, dynamic typechecks must be made on all objects that are passed and received via methods and the encoding of the class of an object must be accessible outside of the module containing the object. ■

Problem 4 (Existence of objects in other modules)**.** *Analogously, to Problem 3, in place of an object of the wrong type, M_3 code can call the compilation of m by passing a non-existent object id. A non-existent object id is a non-`null` object id that does not point to an object, so that calling methods on it will fail.*

To address this concern, when securely compiled code receives an object from another module, it must assess whether that object exists or not. ■

IV. A Secure Compiler from Jem to Ail

$\llbracket \cdot \rrbracket^{\mathsf{JEM}}_{\mathsf{AIL}}$ is a modular, two-step compiler that inputs a Jem class C and returns two Ail modules. Formally:

$$\llbracket C \rrbracket^{\mathsf{JEM}}_{\mathsf{AIL}} = \mathtt{protect}(\llparenthesis C \rrparenthesis) + \llbracket Sys \rrbracket^{\mathsf{JEM}}_{\mathsf{AIL}}.$$

$\llbracket \cdot \rrbracket^{\mathsf{JEM}}_{\mathsf{AIL}}$ consists of a call to the $\llparenthesis \cdot \rrparenthesis$ function (Section IV-A), followed by a call to the $\mathtt{protect}(\cdot)$ one (Section IV-C). The result is joined with an additional module $\llbracket Sys \rrbracket^{\mathsf{JEM}}_{\mathsf{AIL}}$ (Section IV-B).

Intuitively, $\llparenthesis \cdot \rrparenthesis$ is a compiler responsible for correctly translating classes, methods and objects to assembly. $\mathtt{protect}(\cdot)$ is a wrapper around the code generated by $\llparenthesis \cdot \rrparenthesis$ that prevents direct access to that code and ensures that any interaction with that code is regulated to behave as valid Jem code. This wrapping ensures that the output of $\llbracket \cdot \rrbracket^{\mathsf{JEM}}_{\mathsf{AIL}}$ is secure. $\llbracket Sys \rrbracket^{\mathsf{JEM}}_{\mathsf{AIL}}$ is a central, trusted module that operates as a monitor, regulating the structure of function calls and keeping information of allocated objects.

$\llbracket \cdot \rrbracket^{\mathsf{JEM}}_{\mathsf{AIL}}$ is modular, i.e., it can be applied to Jem components $\mathcal{C}$. In this case $\llbracket \cdot \rrbracket^{\mathsf{JEM}}_{\mathsf{AIL}}$ recursively calls itself on all classes composing the Jem component. The secure linker $\mathtt{link}(\cdot)$ combines all compiled components to a low-level component (Section IV-D). Formally:

$$\llbracket C_1; \cdots; C_n \rrbracket^{\mathsf{JEM}}_{\mathsf{AIL}} = \mathtt{link}(\llbracket C_1 \rrbracket^{\mathsf{JEM}}_{\mathsf{AIL}}, \cdots, \llbracket C_n \rrbracket^{\mathsf{JEM}}_{\mathsf{AIL}})$$

The overall structure of the compiler proposed in this paper is similar to the one for a single-module version of *PMA* [4], but two important extensions are needed to handle the issues (Items 1 and 2) mentioned in Section I:

1) we introduce a small centralized trusted module $\llbracket Sys \rrbracket^{\mathsf{JEM}}_{\mathsf{AIL}}$ that keeps information about the global control flow of the program, and about types of objects shared between multiple modules. A key challenge is to design this module such that the information it exposes does not break full abstraction (Section IV-B).
2) we introduce unforgeable object references. Since we do not want to assume hardware support for this (as is done for instance in capability machines), we solve this by assuming the existence of a secure random number generator that can create unguessable random numbers (nonces) that are sufficiently large to make brute-force attacks infeasible (as achievable via the `rdrand` instruction of intel processors [32]). We model these nonces symbolically (i.e., π) and use them to represent references to objects outside of the module where the object is defined (Section IV-C1).

A. The First Step: $\llparenthesis \cdot \rrparenthesis$

$\llparenthesis \cdot \rrparenthesis$ is a compiler that translates a single Jem class to Ail code, data and a symbol table. Instead of giving a specific instance of $\llparenthesis \cdot \rrparenthesis$, we define what assumptions (Assumption 1, 2 and 4) such a compiler must uphold for the full-abstraction result to hold. The full-abstraction result is achieved parametrically for any $\llparenthesis \cdot \rrparenthesis$ that upholds those assumptions.

Assumption 1 (Output of $\llparenthesis \cdot \rrparenthesis$)**.** *The compilation of a* Jem *class returns a memory m_c providing code implementing methods $\overline{M}$, a memory m_d providing data implementing objects $\overline{O}$ and a symbol table t. t contains exported and required methods and objects bindings sorted lexicographically by class and then method or object name [4].*

The produced code and data implement a class-local stack that is used to perform calls and returns between methods.

*$\llparenthesis \cdot \rrparenthesis$ is multi-entry i.e., it exports an address for each method that can be called. $\llparenthesis \cdot \rrparenthesis$ is single-exit i.e., all jumps outside the code are performed by a common piece of code located in a known part of the produced code: m_{exit}. $\llparenthesis \cdot \rrparenthesis$ expects the address where **`instanceof`** is implemented to be supplied later, so a call to **`instanceof`** is compiled to symbols $(\iota_{inst}, \sigma_{inst})$ which capture both parts of the address where **`instanceof`** is. This is captured by the required bindings with a binding of the form $instanceof : \mathtt{Obj}(\mathtt{Obj}, \mathtt{Obj})\mathtt{Bool} \mapsto (\iota_{inst}, \sigma_{inst})$ which must be in t.*

Jem *object ids o are compiled to numbers n that point to a memory region with type information and fields.*

Formally: $\llparenthesis \mathtt{import}\ \overline{I}; \overline{X};\ \mathtt{class}\ c\{K\ \overline{F_t}\ \overline{M}\}; \overline{O} \rrparenthesis = (m_c + m_{exit} + m_{inst}; m_d; t)$.

We overload the $\llparenthesis \cdot \rrparenthesis$ notation and use it to indicate values as are compiled by $\llparenthesis \cdot \rrparenthesis$. Indicate the compilation of `unit` as $\llparenthesis \mathtt{unit} \rrparenthesis$ and the compilation of `true` as $\llparenthesis \mathtt{true} \rrparenthesis$. Analogously, indicate the encoding of Jem types t in Ail as $\llparenthesis t \rrparenthesis$. For example, indicate the encoding of **`Unit`** as $\llparenthesis \mathtt{Unit} \rrparenthesis$ and the encoding of a class type c as a natural number $\llparenthesis c \rrparenthesis = n$.

Assumption 2 (Calling convention of $\llparenthesis \cdot \rrparenthesis$)**.** *Registers are used according to the following calling convention. $\mathtt{r}_0$ is used by caller-callee authentication to store the caller module id. $\mathtt{r}_1$ to $\mathtt{r}_4$ are used as general working registers, so they do not* *contain relevant information.* $\mathtt{r}_5$ *identifies the return address in a* jmp *that models a method call; if* $\mathtt{r}_5$ *is* 1*, then that* jmp *is interpreted as a return.* $\mathtt{r}_6$ *identifies the current object (*`this`*) in a target-level method call or the returned value in a target-level return.* $\mathtt{r}_7$ *onwards are used to communicate method parameters.*

Assumption 3 (Restrictions of $(\!|\cdot|\!)$)**.** *Compiled components do not use the* **exit** E *expression. Addiitonally, they do not read nor write to unprotected memory.*

Assumption 4 (Correctness of $(\!|\cdot|\!)$)**.** $(\!|\cdot|\!)$ *must be correct and adequate, i.e., it translates any* Jem *expressions and values into* Ail *code that behaves in the same way [33], [34], [35].*

B. The System Module: $[\![Sys]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$

The system module $[\![Sys]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$ is a central, trusted module that provides functionality used by all securely-compiled components. $[\![Sys]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$ contains: a global store $\mathcal{G}$ (Section IV-B1), a global call stack $\mathcal{S}$ (Section IV-B2) and functionality to interact with $\mathcal{G}$ and $\mathcal{S}$ (Section IV-B3); its definition is provided last (Section IV-B4).

In the following, when code *aborts* we mean that all registers and flags are reset, then halt is executed.

1) The Global Store $\mathcal{G}$*:* $\mathcal{G}$ tracks globally-known object ids (i.e., ids that are not just local to a module), their type and the module id where the object resides. Formally $\mathcal{G} = \overline{w \mapsto w', id}$, where w is an object id and w' is a class encoding. Retrieval from $\mathcal{G}$ is denoted with $\mathcal{G}(w)$ and addition to it is denoted with $\mathcal{G} + (w \mapsto w', id)$.

$[\![Sys]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$ provides two entry points where the following procedures are implemented: `testObj`(w, w') and `registerObj`(w, w'). The former tells if an object w exists and implements a certain class w'. The latter adds a new binding to $\mathcal{G}$, the new binding is added for the module whose id is in $r(\mathtt{r}_0)$, i.e., for the module that calls `registerObj`$(\cdot)$. Parameters w and w' for these procedures are expected respectively in registers $\mathtt{r}_7$ and $\mathtt{r}_8$.

```
testObj$(w, w')$
if $w \notin$ dom$(\mathcal{G})$ then abort
if $\mathcal{G}(w) \equiv (w', \_)$ then
    return $0$
else return $1$
```

```
registerObj$(w, w')$
if $w \in$ dom$(\mathcal{G})$ then abort
$\mathcal{G} + w \mapsto w', r(\mathtt{r}_0)$
return $0$
```

2) The Global Call Stack $\mathcal{S}$*:* $\mathcal{S}$ tracks all Ail-level function calls. Formally, $\mathcal{S} = \overline{(a)}$.

$[\![Sys]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$ provides the following procedures implemented at entry points: `forwardCall`$()$ and `forwardReturn`$()$. Both procedures use the following helper functions: `resetFlags`$(\cdot)$ sets flags to 0, `resetRegisters`$(\cdot)$ inputs the registers that need to be reset to 0 and `resetRegistersExcept`$(\cdot)$ inputs the registers that need *not* be reset to 0.

`forwardCall`$()$ reads an address from $\mathtt{r}_3$ and $\mathtt{r}_4$ and forwards the call there. Before forwarding, it stores in $\mathcal{S}$ the addess where to return as passed via registers $\mathtt{r}_0$ and $\mathtt{r}_5$. In order to ensure a correct return, the procedure stores the address of the entry point for `forwardReturn`$()$ (i.e., $3 * \mathcal{N}_w$) in $\mathtt{r}_5$. Since the caller id is placed in $\mathtt{r}_0$ by the caller-callee authentication mechanism of *PMA*, it cannot be tampered with. This procedure aborts if the module jumping here is the one that jumped here last (if any) or if the address where the call is forwarded is inside $[\![Sys]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$.

```
forwardCall$()$
If $\mathcal{S} \neq \emptyset$
    Let $\mathcal{S} = (id, \_); \mathcal{S}'$
    if $r(\mathtt{r}_0) == id$ then abort
if $r(\mathtt{r}_3) == 1$ then abort
Push $(r(\mathtt{r}_0), r(\mathtt{r}_5))$ on $\mathcal{S}$
Set $\mathtt{r}_5$ to $3 * \mathcal{N}_w$
resetFlags$()$
resetRegisters$(\mathtt{r}_0, \mathtt{r}_1, \mathtt{r}_2)$
jmp $\mathtt{r}_3$ $\mathtt{r}_4$
```

`forwardReturn`$()$ pops the head of the stack and returns there, setting $\mathtt{r}_5$ to 1 as expected in the case for returns. If the call stack is empty, it aborts to prevent returning when no code was called. If the id of the caller module stored on $\mathcal{S}$ is different from the id of the module jumping to this procedure, it aborts, as it detects a non-well-bracketed execution flow.

```
forwardReturn$()$
If $\mathcal{S} = \emptyset$ then abort
Pop $(id, n)$ from $\mathcal{S}$ into $\mathtt{r}_2$ and $\mathtt{r}_1$
if $id \neq r(\mathtt{r}_0)$ then abort
Set $\mathtt{r}_5$ to 1;
resetFlags$()$
resetRegistersExcept$(\mathtt{r}_1, \mathtt{r}_2, \mathtt{r}_5, \mathtt{r}_6)$
jmp $\mathtt{r}_1$ $\mathtt{r}_2$        // i.e., jump to address $(id, n)$
```

The calling convention is updated as follows: calls to different modules must go via $[\![Sys]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$, with registers $\mathtt{r}_3$ and $\mathtt{r}_4$ set to the method that needed to be called. Both these procedures do not use registers $\mathtt{r}_6$ onwards, so they do not alter the calling convention regarding parameters and returned values.

3) Functionality of $[\![Sys]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$*:* The addition of $[\![Sys]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$ may seem to violate full abstraction as it provides target-level functionality that are not available in Jem [1]. Let us now see why this is not the case.

Procedure `testObj`$(\cdot)$ is analogous to the **`instanceof`** expression in Jem. This procedure aborts when the object id parameter is not registered in $\mathcal{G}$ i.e., an ill-formed execution of **`instanceof`**. Aborting in this case lets $[\![Sys]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$ prevent object id guessing (Problem 1).

Procedure `registerObj`$(\cdot)$ is analogous to object creation. This procedure aborts when the object id is already registered; aborting in this case ensures that object ids are globally unique.

Procedure `forwardCall`$(\cdot)$ is analogous to performing a function call. This procedure aborts when the same module performs two calls in a row without there being a return in between or if it performs a call to $[\![Sys]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$ as those behaviours are not available in Jem.

Procedure `forwardReturn`$(\cdot)$ is analogous to returning from a function call. This procedure aborts when a return is not made by the module who was called last. This enforces well-bracketed control flow, preventing Problem 2 as well as external code returning when no method was called.

All the procedures implemented in $[\![Sys]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$ add functionality in AIL that is already available in JEM, so adding $[\![Sys]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$ when linking AIL components does not violate full-abstraction.

4) $[\![Sys]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$ Definition: Assume the functions provided by $[\![Sys]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$ span memory m, whose size is n addresses. Let the data needed by these functions plus the initialisation of $\mathcal{G}$ and $\mathcal{S}$ span memory m'. $[\![Sys]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$ is always compiled to a module with id 1 (since 0 is the id of unprotected code) and it is defined as follows:

$$(m+m'; (1,n,4); EM_{to}, EM_{ro}, EM_{rc}, EM_{tr}; \emptyset; \emptyset; \emptyset)$$

$[\![Sys]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$ exports a method binding for each procedure it defines; it has no exported object bindings and no required bindings.

- $EM_{to} = testObj : \texttt{Obj}(\texttt{Obj},\texttt{Obj})\texttt{Bool} \mapsto (1,0)$
- $EM_{ro} = registerObj : \texttt{Obj}(\texttt{Obj},\texttt{Obj})\texttt{Unit} \mapsto (1,\mathcal{N}_w)$
- $EM_{rc} = forwardCall : \texttt{Obj}(\cdot)\texttt{Unit} \mapsto (1,2*\mathcal{N}_w)$
- $EM_{tr} = forwardReturn : \texttt{Obj}(\cdot)\texttt{Unit} \mapsto (1,3*\mathcal{N}_w)$

$[\![\cdot]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$ knows the offset of all procedures defined in $[\![Sys]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$, so it can perform calls to them. For example a call to registerObj(·) is a $\mathtt{jmp}\ \mathtt{r_d}\ \mathtt{r_i}$ where $r(\mathtt{r_d}) \mapsto \mathcal{N}_w$ and $r(\mathtt{r_i}) \mapsto 1$.

C. The Second Step: protect(·)

The protect(·) function is a wrapper that takes the memory generated by $(\!|\cdot|\!)$ and adds checks to it, making it secure. This section first presents the required helper functions to compile JEM features: dynamic memory allocation (Section IV-C1), function calls (Section IV-C2) and outcalls (Section IV-C3). The definition of protect(·) is provided last (Section IV-C4).

1) Dynamic memory allocation: Dynamic memory allocation is the creation of objects at runtime via the **new** expression in JEM. The representation of an object id within a compiled component does not change. To ensure security of object ids when communicated between AIL modules, object ids are no longer just an address in memory, they are symbolic nonces. The latter are called *cross-module object ids* and they are denoted with $[\![o]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$. Formally: $[\![o]\!]^{\mathsf{JEM}}_{\mathsf{AIL}} = \pi$. Symbolic nonces cannot be forged nor guessed, so the format of cross-module object ids addresses Problem 1. Cross-module object ids being a symbolic nonce instead of an address does not disrupt the functionality of compiled JEM code, which functions in the same way as before.

To relate cross-module and internal object ids within a module, masking tables are used [4]. A masking table, indicated with $\mathcal{T}$, is a bidirectional hash map between internal object id representations and cross-module ones; each module has its own masking table. Formally $\mathcal{T} ::= \overline{w \mapsto \pi}$. Before an (internal) object id w is first passed to external code, it is placed in the table. This process is called *masking*, thus the passed id π is called the *mask*. Denote the retrieval of a mask π with $\mathcal{T}(\pi)$ and the retrieval of an internal object id w with $\mathcal{T}(w)$. Any retrieval causes abortion if the element to be retrieved is not in $\mathcal{T}$. Adding an internal object id w to a table $\mathcal{T}$ is denoted with $\mathcal{T}+w$; the binding $w \mapsto \pi$, where π is fresh, is added to $\mathcal{T}$.

Next are the functions used to manage masking tables.

A compiled component must store the encoding of the class it contains for helper functions to rely on. Function classOf(n) returns the encoding of the class implemented by the object compiled at address n in the current module. Function isInternal(w) takes a class type encoding and returns true if the current module implements that class.

Function updateMaskingTable(w,n) inputs a pair of an internal object id and a class type encoding. This function is invoked before releasing an object to external code to add the id of freshly-allocated object to the masking table. This function retrieves the class of the object and then checks if that object id is already in the masking table; if not, it adds the id to the table and registers its cross-module id globally. Retrieving the class object may be necessary, as some arguments may have formal type **Obj**. A crucial part of this function is the call to registerObj$(\mathcal{T}(w), n')$, which makes the information that object w, with cross-module id $\mathcal{T}(w)$, implements class n' globally available via $[\![Sys]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$.

```
updateMaskingTable($w, n$)
if $n$ is a class type encoding then
  let $n'$ = classOf($w$)
  if isInternal($n'$) and if $w \notin$ dom($\mathcal{T}$) then
    $\mathcal{T}$+$w$ ; registerObj($\mathcal{T}(w), n'$)
```

Function loadObjects(·) loads the internal object ids of the masking indexes that are passed as input into the related register.

$$\frac{\texttt{loadObjects}(\pi_1, \cdots, \pi_n)}{\forall i \in 1..k \qquad r(\mathtt{r_i}) \mapsto \mathcal{T}(\pi_i)}$$

Function maskingTable(EO) creates the masking table for all exported objects bindings listed in EO.

$$\frac{\texttt{maskingTable}(o_1 : c_1 \mapsto n_1, \cdots, o_k : c_k \mapsto n_k) = \mathcal{T}}{\forall i \in 1..k \quad \text{let } w_i = \texttt{new}\ \mathtt{r_0} \quad \mathcal{T} = n_1 \mapsto w_1, \cdots, n_k \mapsto w_k}$$

2) Function calls: Function calls are calls from outside to within a compiled module. The *PMA* access control policy ensures that these calls can only be calls to entry points. Therefore, an entry point is created for all methods [4]. Function methodEP(M_t, n) creates the code to be placed at a method entry point for a method with signature M_t whose implementation is located at address (id, n) in the current module with id id. Following is the pseudo-code of method entry points; notation $n \mapsto$ code means that n is the address where code is located.

Upon jumping to an entry point, a check is made that the jump comes from the $[\![Sys]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$ module and that the return address is the forwardReturn(·) address (i.e., $3*\mathcal{N}_w$ in a module whose id is 1). If this is not the case, execution is

```
              methodEP($t(t_1, \cdots, t_k) \to t', n$) =
              If $r(\mathtt{r}_0) \neq 1$ or $r(\mathtt{r}_5) \neq 3 * \mathcal{N}_w$ then abort
              loadObjects($r(\mathtt{r}_6), r(\mathtt{r}_7), \cdots, r(\mathtt{r}_{6+k})$)
              dynamicTypechecks($r(\mathtt{r}_6), \llparenthesis t \rrparenthesis$)
              for i = 1 to k
                dynamicTypechecks($r(\mathtt{r}_{6+i}), \llparenthesis t_i \rrparenthesis$)
$n' \mapsto$      Set $\mathtt{r}_5$ to $n' + 1$ and jump to module-local address $n$
              updateMaskingTable($r(\mathtt{r}_6), \llparenthesis t' \rrparenthesis$)
              maskObjectId(6)
              Set $\mathtt{r}_5$ to $1$ and jmp to the address ($1$,3*$\mathcal{N}_w$)
```

aborted as some code is trying to bypass $[\![Sys]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$. All data needed for this check is known statically Then, masked object ids are loaded via function loadObjects(·). Only parameters of object type are loaded; if a parameter could not be loaded, the execution is aborted. Once the objects are loaded, dynamic typechecks are made via function dynamicTypechecks(·) (explained below). These checks are located inside the module where class t is compiled, so checking that the current object ($\mathtt{r}_6$) is of type $\llparenthesis t \rrparenthesis$ is equivalent to checking that the current object implements the current method. Then the code jumps to the method body located at address n setting $\mathtt{r}_5$ to the address where that code must return. There, current arguments are placed on the module-local stack alongside other information required by function activation records. When the method body returns (to address $n' + 1$), the masking table is (possibly) updated with the value to be returned (in $\mathtt{r}_6$) via function updateMaskingTable(·) and internal object ids in registers are masked via function maskObjectId(·).

Let us now provide details about the auxiliary functions.

dynamicTypechecks(·) ensures that each parameter inhabits its type. **Unit**-typed values are checked to be $\llparenthesis$unit$\rrparenthesis$ and **Bool**-typed ones are checked to be either $\llparenthesis$true$\rrparenthesis$ or $\llparenthesis$false$\rrparenthesis$ [4], [7]. Objects are dynamically typechecked by means of the testObj(·) function (as discussed in Section IV-B). If any check fails, the execution is aborted. By checking the existence and types of all parameters, securely compiled code is resilient to Problems 3 and 4.

```
dynamicTypechecks($w, n$)
if $n \equiv \llparenthesis$Unit$\rrparenthesis$ then  if $w \neq \llparenthesis$unit$\rrparenthesis$ then abort
if $n \equiv \llparenthesis$Bool$\rrparenthesis$ then  if $w \neq \llparenthesis$true$\rrparenthesis$ and $w \neq \llparenthesis$false$\rrparenthesis$ then abort
if $n$ is a class type and $w \neq \llparenthesis$null$\rrparenthesis$ then
  if testObj($w, n$) == $1$ then abort
```

Function maskObjectId(·) inputs the index of the register to mask and loads a cross-module object id there.

$$\frac{\texttt{maskObjectId}(n)}{r(\mathtt{r}_n) \mapsto \mathcal{T}(r(\mathtt{r}_n))}$$

3) Outcalls: Outcalls are the dual of function calls, i.e., they are calls from within to outside a module. To allow returning from outcalls, a specific entry point must be created: the *return entry point* [4].

Function preamble(·) returns what all code must execute before making an outcall, function returnEP(·) returns the code m to be placed at the return entry point.

```
preamble() =
Let $a$ be the address where the external method is; $a \equiv (id, w)$
signatureOf($a$) = $\llparenthesis t \rrparenthesis(\llparenthesis t_1 \rrparenthesis, \cdots, \llparenthesis t_n \rrparenthesis) \to \llparenthesis t' \rrparenthesis$
storeData($r(\mathtt{r}_6), \llparenthesis t' \rrparenthesis, r(\mathtt{r}_5)$)
for i = 1 to n
  updateMaskingTable($r(\mathtt{r}_{6+i}), \llparenthesis t_i \rrparenthesis$)
  maskObjectId($6 + i$)
resetRegistersExcept($\mathtt{r}_0, \cdots, \mathtt{r}_{6+n}$)
Set $r(\mathtt{r}_3)$ to $id$ and $r(\mathtt{r}_4)$ to $w$
Call forwardCall( · )
```

The *preamble* code is executed before jumping to an external method located at address a, assume this address is communicated via registers $\mathtt{r}_0$ and $\mathtt{r}_1$. Compiled code will execute the preamble(·) before jumping outside to ensure the right checks are made. signatureOf(·) is used to determine the signature of the compiled method related to a. All method signatures are known statically so their signature encoding can be stored in a table mapping addresses to signature encodings; the table is placed in the data section of the module. This data is communicated to protect(·) via the required method bindings RM returned by $\llparenthesis \cdot \rrparenthesis$. preamble() then calls to storeData(·), which stores the current object $\mathtt{r}_6$, the expected return type $\llparenthesis t' \rrparenthesis$, and where to resume the execution after the outcall, an address that is assumed to be passed in $\mathtt{r}_5$. For any parameter, the masking table is updated with any possible newly created object via function updateMaskingTable(·) and their ids are masked with function maskObjectId(·). Then, registers that are not used to convey parameters nor the address where to jump are reset to 0 by function resetRegistersExcept(·) in order not to leak information ($[\![Sys]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$ will erase unused registers with index less than 6). This ensures that $\mathtt{r}_5$ contains 0, so $[\![Sys]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$ will return to the return entry point located at address 0 when forwarding the return after this call. Finally the code sets $\mathtt{r}_3$ and $\mathtt{r}_4$ as expected by $[\![Sys]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$, then it jumps to the proxy function for method calls: forwardCall().

```
          returnEP() =
$0 \mapsto$     If $r(\mathtt{r}_0) \neq 1$ then abort
          loadData()            // access $w_o$, $\llparenthesis t' \rrparenthesis$, $n'$
          loadObjects($r(\mathtt{r}_6)$)
          dynamicTypechecks($r(\mathtt{r}_6), \llparenthesis t' \rrparenthesis$)
          Resume execution from address $n'$ with current object $w_o$
```

When a return is made, if the module returning is not $[\![Sys]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$ (i.e., a module with id 1), then the code aborts, as some code is trying to bypass $[\![Sys]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$. Then, the current object w_o, the expected return type $\llparenthesis t \rrparenthesis$ and the address where to resume execution n' are loaded from the module-local stack via function loadData(·) into registers $\mathtt{r}_7$ onwards (since these registers are unused). Finally, the returned value $\mathtt{r}_6$ is checked to be of the expected type. Execution aborts if any check fails, otherwise it resumes within the module with the loaded current object.

4) The protect(·) *function:* protect(·) is formalised in Rule protect(·) definition.

To ensure that the compiler is correct, protect(·) needs to provide an implementation of **instanceof**, addressing Problem 5 below.

Problem 5 (Using **instanceof**)**.** *Consider a module M that contains object o implementing class c. Another module M_1 executes the following code:* ***instanceof***$(o : c)$*. In order to tell if the test succeeds or not, the code of M_1 must know the class of o. However, with the strong encapsulation provided by PMA, that information resides in the memory of M, which is not accessible by M_1.* ■

To correctly implement cross-module **instanceof**, the class of an object needs to be publicly known, which is a functionality provided by $\llbracket \mathit{Sys} \rrbracket^{\mathsf{JEM}}_{\mathsf{AIL}}$. When calling **instanceof** on an object whose type is implemented in another module, it suffices to call `testObj( · )` to know if the test succeeds or not. To ensure this happens, the required method binding for $\mathit{instanceof}$ is replaced with a method binding for the same symbols to `testObj( · )`. Linking to $\llbracket \mathit{Sys} \rrbracket^{\mathsf{JEM}}_{\mathsf{AIL}}$ will ensure that those symbols are replaced with the address of `testObj( · )`.

$$\text{(}\texttt{protect}(\,\cdot\,)\text{ definition)}$$

$$\frac{\begin{array}{c}
(\!| C |\!) = (m_c + m_{exit}; m_d; \mathit{EM}; \mathit{EO}; \mathit{RM} + \mathit{RM}_i; \mathit{RO}) \\
\mathit{EM} = m_1 : M_{t1} \mapsto (id, n_1), \cdots, m_k : M_{tk} \mapsto (id, n_k) \\
\mathit{EM}' = \; m_1 : M_{t1} \mapsto a_1, \cdots, m_k : M_{tk} \mapsto a_k \\
\mathit{EO} = o_1 : c_1 \mapsto n''_1, \cdots, o_j : c_j \mapsto n''_j \\
\mathit{EO}' = o_1 : c_1 \mapsto \mathcal{T}(n''_1), \cdots, o_j : c_j \mapsto \mathcal{T}(n''_j) \\
\mathit{RM}_i = \mathit{instanceof} : \mathbf{Obj}(\mathbf{Obj}, \mathbf{Obj})\mathbf{Bool} \mapsto (\iota_{inst}, \sigma_{inst}) \\
\mathit{RM}'_i = \mathit{testObject} : \mathbf{Obj}(\mathbf{Obj}, \mathbf{Obj})\mathbf{Bool} \mapsto (\iota_{inst}, \sigma_{inst}) \\
s = (id, n', k+1) \qquad \texttt{maskingTable}(\mathit{EO}) = \mathcal{T} \\
m_m = \texttt{implementationOf}(\mathcal{T}) \qquad m_r = \; \texttt{returnEP}() \\
m_{ce} = \; \texttt{extraCode}() \qquad m_{de} = \; \texttt{extraData}() \\
\forall i \in 1..k \quad a_i = (id, i \cdot \mathcal{N}_w) \quad m_i = \; \texttt{methodEP}(M_{ti}, n_i) \\
m'_{exit} = \texttt{append}(m_{exit}, \texttt{preamble}(\,\cdot\,)) \\
m_{code} = m_1 + \cdots + m_k + m_r + m_c + m'_{exit} + m_{ce} \\
n' = |m_{code}| \qquad m = m_{code} + m_d + m_{de} + m_m
\end{array}}{\texttt{protect}((\!| C |\!)) = \; m; s; \mathit{EM}'; \mathit{EO}'; \mathit{RM} + \mathit{RM}'_i; \mathit{RO}}$$

`protect( · )` calculates the memory layout s based on the functionality listed in EM. The memory created by `protect( · )` contains the following functionality. Each method exported to external code is given an entry point with code created with the `methodEP( · )` function. Function `returnEP( · )` yields memory m_r for the return entry point. The code section of m is completed with the `extraCode( · )`, containing the implementation of all the helper functions described above (e.g., `dynamicTypechecks( · )`, `resetFlagsAndRegs( · )` etc.) and with m_c, the code generated by $(\!| \cdot |\!)$. Additionally, the code section appends the `preamble( · )` procedure at the exit-point code provided by $(\!| \cdot |\!)$ to ensure that it is always executed before exiting the module. After the code section, m contains `extraData( · )`, i.e., all the data needed by the helper functions (e.g., the encodings of method signatures and of types) and by m_d, the data generated by $(\!|\cdot|\!)$. Finally the data section contains the masking table m_m, which is obtained with function `maskingTable( · )`; as memories are infinite, the data section of a module has no boundaries.

What `protect( · )` returns exports the same methods and objects as $(\!| \cdot |\!)$. The former are bound to entry points and the latter have their ids masked.

D. The Secure Linker: `link( · )`

This section presents function `link( · )`, which inputs and returns AIL modules; it is formalised as follows:

$$\begin{aligned}
\texttt{link}(P, P') &= M_1 \uplus \cdots \uplus M_n \uplus M'_1 \uplus \cdots \uplus M'_m \\
\text{where} \quad & P \equiv M_1 \uplus \cdots \uplus M_n \\
\text{and} \quad & P' \equiv M'_1 \uplus \cdots \uplus M'_m
\end{aligned}$$

Modules M_i are obtained through calls to $\llbracket \cdot \rrbracket^{\mathsf{JEM}}_{\mathsf{AIL}}$, they all are a pair of a compiled JEM class and an instance of $\llbracket \mathit{Sys} \rrbracket^{\mathsf{JEM}}_{\mathsf{AIL}}$. Informally, the $\uplus$ operator does the following:

- it performs AIL-level joining of modules ($+$, presented in Section II-B2) ensuring that only one occurrence of $\llbracket \mathit{Sys} \rrbracket^{\mathsf{JEM}}_{\mathsf{AIL}}$ is present in the resulting component;
- it initialises the resulting $\llbracket \mathit{Sys} \rrbracket^{\mathsf{JEM}}_{\mathsf{AIL}}$ table $\mathcal{G}$ with all exported object bindings for all M_i, so that static objects are registered in $\mathcal{G}$.

V. Discussion

This section presents how to extend $\llbracket \cdot \rrbracket^{\mathsf{JEM}}_{\mathsf{AIL}}$ to a source language supporting object-orientation (Section V-A) and how to support multi-register data (Section V-B).

A. Supporting Object-Orientation

JEM can be made object-oriented by adding support for interfaces, inheritance and dynamic dispatch. We suggest doing this by adding interfaces to JEM, making class types private to a component and only using interface types in the types of cross-component method calls and returns [4], [36]. Since different JEM components can implement the same interface, a number of concerns arise. Nevertheless, we believe the concerns can be adequately addressed and this section also describes a possible change to our compiler which we believe is a way to solve the concerns.

Problem 6 (Module id at the target level)**.** *Consider two modules M_1 and M_2 containing the compilation of two classes that implements the same interface i. Code in another module M could input objects of that interface. However, M cannot be sure of where the object id is located unless it indicates its module id. In fact, all that M knowns when receiving an argument is the type i, but both M_1 and M_2 implement it.* ■

To address Problem 6, cross-module object ids need to state the module id where they reside; formally $\llbracket o \rrbracket^{\mathsf{JEM}}_{\mathsf{AIL}} = \pi, id$. Unless AIL is extended to merge this information in nonces, object ids span multiple words (Section V-B describes how to securely compile them).

Often, object-oriented languages implements dynamic dispatch as vtables that are located at the address where an object is compiled. With *PMA*, M_1 has no access to the vtable of o if o is allocated in M_2. The single-module version of AIL [4] used entry points as a vtable but this does not hold when multiple modules are considered.

Problem 7 (Dynamic dispatch)**.** *Consider two modules M_1 and M_2 containing the compilation of two classes implementing the same interface Bank. Module M_1 also implements interface $\mathit{Account}$ while module M_2 also implements interface $\mathit{Currency}$. Since methods need to be sorted alphabetically based on their namespaces (which include interface names), methods for Bank in M_1 will have different entry points than the same methods in M_2.*

A module M interacting with M_1 and M_2 however should not need to know the full specification of which component contains which interfaces, as this would defeat the purpose of object orientation. All that M knows is that somewhere outside its memory, interface Bank is implemented. When receiving an object from M_1 or from M_2 that implements the Bank interface, M needs to be able to calculate where to jump in order to call methods on it. ■

To address Problem 7, a single method entry point is created at address $\mathcal{N}_w$ instead of an entry point per method implemented in a component. That entry point serves as a dynamic dispatch entry point. All cross-module method calls update the calling convention to use $\mathtt{r}_6$ as a container for the encoding of the method to be called. Formally, indicate a method encoding as follows $(\!| m |\!) = n$.

When a module receives an object id of the form π, id, it can call method m on it by jumping to address $(id, 0)$ and setting $\mathtt{r}_6$ to $(\!| m |\!)$. Parameters and the current object are then passed via registers $\mathtt{r}_7$ onwards.

The dynamic dispatch entry point must perform a new check: $\mathtt{r}_6$ must be a valid method encoding. All methods implemented in interfaces are known statically, so a module can save this information in its memory to encode this check. If this check succeeds, the execution continues by dispatching to the checks for method entry points of Section IV-C.

B. Supporting Multi-Register Data

In some cases, securely-compiled code needs to communicate data that spans multiple registers. For example, the source language could be extended to support computation on complex numbers, or the format of object ids could vary (as discussed in Section V-A). Communicating data that spans multiple registers can be done by extending AIL with cryptographic functions (either in the form of instructions, as presented here, or as module-internal procedures).

First, all modules must have a public and a private key, all modules know each other's public keys but the private one is confidential to the module [13]. Second, cryptograhic functions for signing and verifying signatures are needed.

$\mathtt{sign}\ \mathtt{r_s}\ \mathtt{r_e}\ \mathtt{r_k}\ \mathtt{r_d}$	Sign all registers from $\mathtt{r_s}$ to $\mathtt{r_e}$ with the (private) key found in register $\mathtt{r_k}$ and place the result in register $\mathtt{r_d}$.
$\mathtt{verify}\ \mathtt{r_s}\ \mathtt{r_e}\ \mathtt{r_k}\ \mathtt{r_d}$	Set the ZF according to whether the signature found in register $\mathtt{r_d}$ has been created for all registers from $\mathtt{r_s}$ to $\mathtt{r_e}$ with the dual key of that found in register $\mathtt{r_k}$.

Consider the case of multi-register object ids of Section V-A; cross-module object ids need to be changed into a triplet, so they span three registers. The triplet consists of: a masking index (as in Section IV-C), a type encoding and a signature of the two; the signature prevents Problem 8 below; Formally: $\llbracket o \rrbracket^{\mathsf{JEM}}_{\mathsf{AIL}} = \pi, n, w$.

Problem 8 (Object-id shuffling)**.** *Consider two objects o_1 and o_2 residing in two modules M_1 and M_2 implementing two different classes. Without signatures, those objects would have the following cross-module object ids: $\llbracket o_1 \rrbracket^{\mathsf{JEM}}_{\mathsf{AIL}} = \pi_1, n_1$ and $\llbracket o_2 \rrbracket^{\mathsf{JEM}}_{\mathsf{AIL}} = \pi_2, n_2$. An attacker can forge new object ids as follows: π_1, n_2 and π_2, n_1. Another module M_3 receiving the forged objects from the attacker has no way to tell whether they are forged by inspecting them.* ■

With the signature in place, the object ids of o_1 and o_2 change as follows: $\llbracket o_1 \rrbracket^{\mathsf{JEM}}_{\mathsf{AIL}} = \pi_1, n_1, w_1$ and $\llbracket o_2 \rrbracket^{\mathsf{JEM}}_{\mathsf{AIL}} = \pi_2, n_2, w_2$. An attacker can still forge object ids by creating π_1, n_2, w_2, but M_3 can verify the signature part of the triplet, therefore finding out when forgery has taken place.

With this approach, some auxiliary functions need to be changed. Function `loadObjects`$(\cdot)$ needs to verify that the object ids of all externally-located modules have been signed with the proper key and that the key corresponds to the module that implements the class n mentioned in the object id. As a module implements a single class, the type encoding tells a module which public key to use for the verification. Function `updateMaskingTable`$(\cdot)$ needs to ensure that communicated cross-module object id are signed.

Concerning $\llbracket \mathit{Sys} \rrbracket^{\mathsf{JEM}}_{\mathsf{AIL}}$, `registerObj`$(\cdot)$ is not needed and the pseudo code inserted by `protect`$(\cdot)$ does not call it. Procedure `testObj`$(\cdot)$ can be implemented locally in a module, since due to the new cross-module object ids the type of an object is part of its id.

VI. Full-Abstraction and Modularity of $\llbracket \cdot \rrbracket^{\mathsf{JEM}}_{\mathsf{AIL}}$

As stated in Section I, a compiler is fully-abstract if it preserves and reflects contextual equivalence of source and target components. This section briefly discusses how contextual equivalence is reflected (Section VI-A) and preserved (Section VI-B) for $\llbracket \cdot \rrbracket^{\mathsf{JEM}}_{\mathsf{AIL}}$. Then it concludes by presenting the proof sketch of full-abstraction and modular full-abstraction of $\llbracket \cdot \rrbracket^{\mathsf{JEM}}_{\mathsf{AIL}}$ (Section VI-C).

A. $[\![\cdot]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$ *Reflects Contextual Equivalence*

Proving that $[\![\cdot]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$ reflects contextual equivalence (Theorem 1) is analogous to proving that it is correct and adequate.

Theorem 1 ($[\![\cdot]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$ preserves behaviour)**.** $\forall \mathcal{C}_1, \mathcal{C}_2. [\![\mathcal{C}_1]\!]^{\mathsf{JEM}}_{\mathsf{AIL}} \simeq_{ctx} [\![\mathcal{C}_2]\!]^{\mathsf{JEM}}_{\mathsf{AIL}} \Rightarrow \mathcal{C}_1 \simeq_{ctx} \mathcal{C}_2$.

Intuitively, given Assumption 4, this holds because neither `protect( · )` nor the addition of $[\![Sys]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$ change the semantics of compiled programs.

B. $[\![\cdot]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$ *Preserves Contextual Equivalence*

Proving that $[\![\cdot]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$ preserves contextual equivalence (Theorem 2) is analogous to proving that $[\![\cdot]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$ is secure. Theorem 2 intuitively states that JEM abstractions are preserved in the AIL output $[\![\cdot]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$ produces.

Theorem 2 ($[\![\cdot]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$ is secure)**.** $\forall \mathcal{C}_1, \mathcal{C}_2. \mathcal{C}_1 \simeq_{ctx} \mathcal{C}_2 \Rightarrow [\![\mathcal{C}_1]\!]^{\mathsf{JEM}}_{\mathsf{AIL}} \simeq_{ctx} [\![\mathcal{C}_2]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$.

For Theorem 2 we proceed as follows. First, we devise a notion of trace equivalence $\overset{\mathrm{T}}{=}$ for securely-compiled components; this is a slight adaptation of a similar trace semantics for the single-module version of AIL [24]. Intuitively the trace semantics describes the behaviour of a set of modules with sets of traces, i.e., concatenation of actions such as call and return. Most importantly, trace semantics lets us disregard contexts when reasoning about modules behaviour. We assume that $\overset{\mathrm{T}}{=}$ is equivalent to $\simeq_{ctx}$, so the two notions can be exchanged.

Assumption 5 (Trace semantics coincides with contextual equivalence for securely compiled JEM components)**.** $\forall \mathcal{C}_1, \mathcal{C}_2. [\![\mathcal{C}_1]\!]^{\mathsf{JEM}}_{\mathsf{AIL}} \simeq_{ctx} [\![\mathcal{C}_2]\!]^{\mathsf{JEM}}_{\mathsf{AIL}} \iff [\![\mathcal{C}_1]\!]^{\mathsf{JEM}}_{\mathsf{AIL}} \overset{\mathrm{T}}{=} [\![\mathcal{C}_2]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$.

Then we re-state Theorem 2 in contrapositive form:

$$\forall \mathcal{C}_1, \mathcal{C}_2. [\![\mathcal{C}_1]\!]^{\mathsf{JEM}}_{\mathsf{AIL}} \overset{\mathrm{T}}{\neq} [\![\mathcal{C}_2]\!]^{\mathsf{JEM}}_{\mathsf{AIL}} \Rightarrow \mathcal{C}_1 \not\simeq_{ctx} \mathcal{C}_2$$

To achieve $\mathcal{C}_1 \not\simeq_{ctx} \mathcal{C}_2$ we need to show (by negating Definition 1) that there exist a context that, wlog, terminates with $\mathcal{C}_1$ and diverges with $\mathcal{C}_2$. This context is said to *differentiate* between $\mathcal{C}_1$ and $\mathcal{C}_2$. For this, we devise an algorithm $\langle\!\langle \cdot \rangle\!\rangle$ that can always generate such a differentiating context given two JEM components whose compiled counterparts are trace-inequivalent [4], [24], [37]. Since $\langle\!\langle \cdot \rangle\!\rangle$ is sketched to be correct (Theorem 3), we can use it to witness that two JEM components are contextually-inequivalent if their compiled counterparts are trace-inequivalent.

Theorem 3 (Algorithm correctness)**.** $\forall \mathcal{C}_1, \mathcal{C}_2$, $\overline{\alpha}\alpha_1 \in \mathsf{Traces}_{\mathsf{AIL}}([\![\mathcal{C}_1]\!]^{\mathsf{JEM}}_{\mathsf{AIL}})$, $\overline{\alpha}\alpha_2 \in \mathsf{Traces}_{\mathsf{AIL}}([\![\mathcal{C}_2]\!]^{\mathsf{JEM}}_{\mathsf{AIL}})$, $\alpha_1 \neq \alpha_2$, $\langle\!\langle \mathcal{C}_1, \mathcal{C}_2, \overline{\alpha}\alpha_1, \overline{\alpha}\alpha_2 \rangle\!\rangle = \mathbb{C}$ *such that* $\mathbb{C}[\mathcal{C}_1]\Uparrow \not\Leftrightarrow \mathbb{C}[\mathcal{C}_2]\Uparrow$.

C. *Full-Abstraction and Modularity*

By the theorems of Sections VI-A and VI-B, $[\![\cdot]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$ is fully-abstract (Theorem 4).

Theorem 4 ($[\![\cdot]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$ is fully-abstract)**.** $\forall \mathcal{C}_1, \mathcal{C}_2$. $\mathcal{C}_1 \simeq_{ctx} \mathcal{C}_2 \iff [\![\mathcal{C}_1]\!]^{\mathsf{JEM}}_{\mathsf{AIL}} \simeq_{ctx} [\![\mathcal{C}_2]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$.

The novel result we are after for $[\![\cdot]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$ is *modular full-abstraction*, i.e. components can be compiled separately and linked together afterwards without compromising security. Indicate the linking of two AIL components P_1 and P_2 as $\mathtt{link}(P_1, P_2)$. This definition is derived from Ahmed's definition of horizontal compiler compositionality [38]; it states that we can create source-level components by joining an arbitrary number of them and by compiling them individually and then linking the result. Additionally, we prove this result for arbitrary target-level components P and P' that are equivalent to securely-compiled ones. P and P' can be seen as hand-optimised versions of $\mathcal{C}_2$ and $\mathcal{C}_4$ that respect the behaviour imposed by $[\![\cdot]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$. Formally, we have that Corollary 1 is a corollary of Theorem 4.

Corollary 1 (Modular full-abstraction)**.** $\forall \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4$. $\forall P. [\![\mathcal{C}_2]\!]^{\mathsf{JEM}}_{\mathsf{AIL}} \simeq_{ctx} P$, $\forall P'. [\![\mathcal{C}_4]\!]^{\mathsf{JEM}}_{\mathsf{AIL}} \simeq_{ctx} P'$, $\mathcal{C}_1; \mathcal{C}_2 \simeq_{ctx} \mathcal{C}_3; \mathcal{C}_4 \iff \mathtt{link}([\![\mathcal{C}_1]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}, P) \simeq_{ctx} \mathtt{link}([\![\mathcal{C}_3]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}, P')$.

The companion technical report [20] contains proofs sketches for Theorem 1 based on Assumption 2, for Theorem 2 based on Assumption 5, Theorems 3 and 4 as well as proofs for Corollary 1 (and for related helper lemmas). The assumption that $\overset{\mathrm{T}}{=}$ is equivalent to $\simeq_{ctx}$ is left to prove for future work.

VII. Related Work

Secure compilation through full-abstraction was pioneered by Abadi [1] and successfully applied to many different settings [2], [3], [4], [5], [6], [7], [10], [11]. Parrow proved which conditions must hold in source and target languages to provide a fully-abstract compiler between the two [23]. Gorla and Nestmann concluded that full-abstraction is meaningful when it entails properties like security [22].

A large body of research provided secure compilers for a variety of languages with different language features. Three works are closely related to the present one. The first one is the fully-abstract compilation scheme of single-module code to single-module *PMA* of Patrignani *et al.* [4], where linking is not explicitly considered. The second one is the secure compiler targeting an extension of the PUMP machine (a tag-based assembly-level architecture that enforces micro-policies with each instruction) [17] by Juglaret and Hritcu [18]. This work considers a very similar source language, so it incurs in very similar problems to what discussed here. Intuitively, Juglaret and Hritcu use tags to create *PMA*-like modules (with their local stacks) where jumps can only be done at specific addresses. By relying on a very different architecture, their solution differs significantly from ours. For example, to address Problem 2, they rely on linear tags for return addresses. Their tags also capture type information, so that dynamic typechecks (as needed for Problem 3) are made based on tags. As stated in Section I, the main difference between the two works is in the architecture they target: while SGX-like *PMA* is readily available, it may be a while before PUMP-like machine hit the market. The third closest secure compilation result is the (probabilistic) fully-abstract compiler to ASLR-enhanced target languages, which also does not explicitly consider linking [2], [3]. ASLR prevents some linking problems (e.g., object guessing) but not all of them (e.g., call stack shortcutting). We expect that ASLR-based secure compilation can be made resilient to all linking related attacks by adopting analogous countermeasures to those described in this paper.

Most secure compilation works adopt the fully-abstract compilation notion of Abadi [1]. These works achieve security by relying on type systems for the target language [5], [6], [7], cryptographic primitives [8], [9], [39] and the already mentioned ASLR [2], [3] and *PMA* [4]. Additionally, certain works provide secure compilers by means of type-preserving compilers [40], [41], [42], though they require the target language to be well-typed. Of all these works, only Abadi *et al.* [8] consider multiple modules, but in a distributed setting rather than on a single machine, so that presents different vulnerabilities than the ones considered in this paper.

VIII. Conclusion and Future Work

This paper presented a secure compilation scheme from Jem, an object-based imperative language to Ail, an untyped assembly language enhanced with *PMA*. Because Ail explicitly deals with linking, the secure compiler developed in this paper faces a number of threats that no previous work considers. This paper formalised the compiler $[\![\cdot]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$ and explaind how it withstands these new threats. Finally, it presented how $[\![\cdot]\!]^{\mathsf{JEM}}_{\mathsf{AIL}}$ is fully-abstract and modular, so that the additional threats arising from linking cannot be exploited by a malicious attacker.

The authors foresee a number of future research trajectories for this work: integrating a garbage collector with securely compiled programs, supporting secure compilation for concurrent programs and developing secure compilation schemes for emerging security architectures such as capability machines [15], [16].

Acknowledgements. The authors would like to thank Cătălin Hriţcu for sheperding this paper into acceptance as well as Amal Ahmed, Yannis Juglaret, Raoul Strackx and the anonymous reviewers for useful comments on an earlier version of this paper. Dominique Devriese holds a Postdoctoral mandate from the Research Foundation Flanders (FWO). This work has been supported in part by the Intel Labs University Research Office. This research is also partially funded by the Research Fund KU Leuven, and by the EU FP7 project NESSoS. With the financial support from the Prevention of and Fight against Crime Programme of the European Union (B-CCENTRE).

References

[1] M. Abadi, "Protection in programming-language translations," in *Secure Internet programming*. London, UK: Springer-Verlag, 1999, pp. 19–34. [Online]. Available: http://dl.acm.org/citation.cfm?id=380171.380174

[2] M. Abadi and G. D. Plotkin, "On protection by layout randomization," *ACM TISSEC*, vol. 15, no. 2, pp. 8:1–8:29, Jul. 2012. [Online]. Available: http://doi.acm.org/10.1145/2240276.2240279

[3] R. Jagadeesan, C. Pitcher, J. Rathke, and J. Riely, "Local memory via layout randomization," in *CSF '11*. IEEE, 2011, pp. 161–174. [Online]. Available: http://dx.doi.org/10.1109/CSF.2011.18

[4] M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens, "Secure Compilation to Protected Module Architectures," *ACM Trans. Program. Lang. Syst.*, vol. 37, no. 2, pp. 6:1–6:50, Apr. 2015. [Online]. Available: http://doi.acm.org/10.1145/2699503

[5] A. Ahmed and M. Blume, "Typed closure conversion preserves observational equivalence," *SIGPLAN Not.*, vol. 43, no. 9, pp. 157–168, Sep. 2008. [Online]. Available: http://doi.acm.org/10.1145/1411203.1411227

[6] ——, "An equivalence-preserving CPS translation via multi-language semantics," *SIGPLAN Not.*, vol. 46, no. 9, pp. 431–444, 2011. [Online]. Available: http://doi.acm.org/10.1145/2034574.2034830

[7] C. Fournet, N. Swamy, J. Chen, P.-E. Dagand, P.-Y. Strub, and B. Livshits, "Fully abstract compilation to JavaScript," *SIGPLAN Not.*, vol. 48, no. 1, pp. 371–384, Jan. 2013. [Online]. Available: http://doi.acm.org/10.1145/2480359.2429114

[8] M. Abadi, C. Fournet, and G. Gonthier, "Secure implementation of channel abstractions," *Information and Computation*, vol. 174, no. 1, pp. 37–83. [Online]. Available: http://dx.doi.org/10.1006/inco.2002.3086

[9] M. Bugliesi and M. Giunti, "Secure implementations of typed channel abstractions," in *POPL '07*, 2007, pp. 251–262. [Online]. Available: http://doi.acm.org/10.1145/1190216.1190253

[10] D. Devriese, M. Patrignani, and F. Piessens, "Fully-abstract compilation by approximate back-translation," in *POPL 2016*, 2016, pp. 164–177. [Online]. Available: http://doi.acm.org/10.1145/2837614.2837618

[11] W. J. Bowman and A. Ahmed, "Noninterference for free," in *ICFP '15*. New York, NY, USA: ACM, 2015.

[12] R. Strackx and F. Piessens, "Fides: Selectively hardening software application components against kernel-level or process-level malware," in *CCS 2012*. [Online]. Available: https://lirias.kuleuven.be/handle/123456789/354603

[13] J. Noorman et. al., "Sancus: Low-cost trustworthy extensible networked devices with a zero-software Trusted Computing Base," in *USENIX'13'*.

[14] F. McKeen et. al., "Innovative instructions and software model for isolated execution," in *HASP '13*, 2013. [Online]. Available: http://doi.acm.org/10.1145/2487726.2488368

[15] J. Woodruff et. al., "The CHERI capability model: Revisiting RISC in an age of risk," in *ISCA '14*, 2014, pp. 457–468. [Online]. Available: http://dl.acm.org/citation.cfm?id=2665671.2665740

[16] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway, "Capsicum: Practical capabilities for UNIX." in *USENIX*, 2010, pp. 29–46. [Online]. Available: http://dblp.uni-trier.de/db/conf/uss/uss2010.html#WatsonALK10

[17] A. Azevedo de Amorim, N. Collins, A. DeHon, D. Demange, C. Hriţcu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach, "A verified information-flow architecture," *SIGPLAN Not.*, vol. 49, no. 1, pp. 165–178. [Online]. Available: http://doi.acm.org/10.1145/2578855.2535839

[18] Y. Juglaret and C. Hritcu, "Secure compilation using micro-policies (Extended Abstract)," in *FCS 2015*, 2015.

[19] Y. Juglaret, C. Hriţcu, A. Azevedo de Amorim, and B. C. Pierce, "Beyond good and evil: Formalizing the security guarantees of compartmentalizing compilation," in *29th IEEE Symposium on Computer Security Foundations (CSF)*. IEEE Computer Society Press, Jul. 2016, to appear. [Online]. Available: http://arxiv.org/abs/1602.04503

[20] M. Patrignani, D. Devriese, and F. Piessens, "On Modular and Fully-Abstract Compilation – Technical Appendix," April 2016. [Online]. Available: https://arxiv.org/abs/1604.05044

[21] G. D. Plotkin, "LCF considered as a programming language," *Theoretical Computer Science*, vol. 5, pp. 223–255, 1977. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0304397577900445

[22] D. Gorla and U. Nestman, "Full abstraction for expressiveness: History, myths and facts," *Math Struct Comp Science*, 2014.

[23] J. Parrow, "General conditions for full abstraction," *Math. Struc. Comp. Sci.*

[24] M. Patrignani and D. Clarke, "Fully abstract trace semantics for protected module architectures," *Elsevier COMLAN*, vol. 42, no. 0, pp. 22 – 45, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1477842415000081

[25] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "Trustvisor: Efficient TCB reduction and attestation," in *SP '10*. IEEE, 2010, pp. 143–158. [Online]. Available: http://dx.doi.org/10.1109/SP.2010.17

[26] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth, "Reducing TCB complexity for security-sensitive applications: three case studies,"

SIGOPS Oper. Syst. Rev., vol. 40, no. 4, pp. 161–174, Apr. 2006. [Online]. Available: http://doi.acm.org/10.1145/1218063.1217951

[27] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative Technology for CPU Based Attestation and Sealing," in *HASP'13*, vol. 13, 2013.

[28] D. Dolev and A. C. Yao, "On the security of public key protocols," in *SFCS*. Washington, DC, USA: IEEE Computer Society, 1981, pp. 350–357. [Online]. Available: http://portal.acm.org/citation.cfm?id=1382435.1382728

[29] E. Sumii and B. C. Pierce, "A bisimulation for dynamic sealing," in *Principles of Programming Languages*. ACM, 2004, pp. 161–172.

[30] V. Costan and S. Devadas, "Intel SGX Explained." [Online]. Available: https://eprint.iacr.org/2016/086.pdf

[31] M. Abadi, J. Planul, and G. Plotkin, "Layout randomization and nondeterminism," *Electr. Notes Theo. Comp. Sci.*, vol. 298, pp. 29–50, 2013. [Online]. Available: http://dx.doi.org/10.1016/j.entcs.2013.09.006

[32] M. Hamburg, P. Kocher, and M. Marson, "Analysis of Intel's Ivy Bridge digital random number generator," *Cryptography Research, Inc.*, 2012.

[33] C.-K. Hur and D. Dreyer, "A Kripke logical relation between ML and assembly," *SIGPLAN Not.*, vol. 46, no. 1, pp. 133–146, Jan. 2011. [Online]. Available: http://doi.acm.org/10.1145/1925844.1926402

[34] N. Benton and C.-K. Hur, "Biorthogonality, step-indexing and compiler correctness," *SIGPLAN Not.*, vol. 44, no. 9, pp. 97–108, Aug. 2009. [Online]. Available: http://doi.acm.org/10.1145/1631687.1596567

[35] X. Leroy, "A formally verified compiler back-end," *J. Autom. Reason.*, vol. 43, no. 4, pp. 363–446, Dec. 2009. [Online]. Available: http://dx.doi.org/10.1007/s10817-009-9155-4

[36] A. Jeffrey and J. Rathke, "Java Jr.: fully abstract trace semantics for a core Java language," in *ESOP'05*, 2005, pp. 423–438. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-31987-0_29

[37] F. S. de Boer, M. M. Bonsangue, M. Steffen, and E. Ábrahám, "A fully abstract semantics for UML components," in *FMCO'04*. [Online]. Available: http://dx.doi.org/10.1007/11561163_3

[38] A. Ahmed, "Verified Compilers for a Multi-Language World," in *SNAPL 2015*, ser. (LIPIcs), vol. 32, 2015, pp. 15–31.

[39] R. Corin, P.-M. Deniélou, C. Fournet, K. Bhargavan, and J. Leifer, "A secure compiler for session abstractions," *Journal of Computer Security*, vol. 16, no. 5, pp. 573–636, 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1454415.1454419

[40] G. Barthe, T. Rezk, A. Russo, and A. Sabelfeld, "Security of multithreaded programs by compilation," *ACM TISSEC*, 2010. [Online]. Available: http://doi.acm.org/10.1145/1805974.1805977

[41] I. G. Baltopoulos and A. D. Gordon, "Secure compilation of a multi-tier web language," in *TLDI '09*. ACM, 2009, pp. 27–38. [Online]. Available: http://doi.acm.org/10.1145/1481861.1481866

[42] N. Swamy, C. Fournet, A. Rastogi, K. Bhargavan, J. Chen, P.-Y. Strub, and G. Bierman, "Gradual typing embedded securely in javascript," *SIGPLAN Not.*, vol. 49, no. 1, pp. 425–437, Jan. 2014. [Online]. Available: http://doi.acm.org/10.1145/2578855.2535889