Secure Compilation to Protected Module Architectures

MARCO PATRIGNANI, PIETER AGTEN, RAOUL STRACKX, and BART JACOBS,
iMinds-DistriNet, Dept. Computer Science, KU Leuven, Belgium
DAVE CLARKE, Dept. Information Technology, Uppsala, Sweden and iMinds-DistriNet, Dept. Computer Science, KU Leuven, Belgium
FRANK PIESSENS, iMinds-DistriNet, Dept. Computer Science, KU Leuven, Belgium

A fully abstract compiler prevents security features of the source language from being bypassed by an attacker operating at the target language level. Unfortunately, developing fully abstract compilers is very complex, and it is even more so when the target language is an untyped assembly language. To provide a fully abstract compiler that targets untyped assembly, it has been suggested to extend the target language with a protected module architecture—an assembly-level isolation mechanism which can be found in next-generation processors. This article provides a fully abstract compilation scheme whose source language is an object-oriented, high-level language and whose target language is such an extended assembly language. The source language enjoys features such as dynamic memory allocation and exceptions. Secure compilation of first-order method references, cross-package inheritance, and inner classes is also presented. Moreover, this article contains the formal proof of full abstraction of the compilation scheme. Measurements of the overhead introduced by the compilation scheme indicate that it is negligible.

Categories and Subject Descriptors: D.2.3 [**Coding Tools and Techniques**]: Object-Oriented Programming; D.2.4 [**Software/Program Verification**]: Formal Methods; D.3.4 [**Processors**]: Compilers; D.4.6 [**Security and Protection**]: Verification

General Terms: Secure Compilation, Object-Oriented Programming, Untyped Machine Code, Security

Additional Key Words and Phrases: Fully abstract compilation, protected module architecture

ACM Reference Format:
Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. 2015. Secure compilation to protected module architectures. ACM Trans. Program. Lang. Syst. 37, 2, Article 6 (April 2015), 50 pages.
DOI: http://dx.doi.org/10.1145/2699503

1. INTRODUCTION

Modern high-level languages such as ML, Java, or Scala offer security features to programmers in the form of type systems, module systems, encapsulation primitives, and so forth. In order for these high-level programs to be executed, they are input

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). Marco Patrignani and Pieter Agten hold a Ph.D. fellowship from the Research Foundation Flanders (FWO). Raoul Strackx is a Ph.D. fellow of the agency for Innovation by Science and Technology (IWT).
Authors' addresses: M. Patrignani, P. Agten, R. Strackx, B. Jacobs, and F. Piessens, Computer Science Department, KU Leuven, Celestijnenlaan 200-A, 3000 Leuven, Belgium; email: {first.last}@cs.kuleuven.be;. D. Clarke (Current address) Department of Information Technology, Polacksbacken (Lägerhyddsvägen 2), Uppsala, Sweden; email: {first.last}@it.uu.se.

DOI: http://dx.doi.org/10.1145/2699503

to other programs: compilers (in certain cases, they are given to interpreters which are eventually compiled). Compilers take programs written in a language, called the *source language*, and output other programs written (often) in a different language, called the *target language*. The target language is (often) machine code, and it can be executed directly by the processor. Unfortunately, most target languages do not offer the same security features as high-level source languages. Consider assembly as the target language of a compiler. An attacker acting at the assembly level can inspect and modify the whole memory space, breaking security properties such as confidentiality and integrity of compiled code. One way to withstand such an attacker is by means of secure compilation, which makes a compiler produce target-level output as secure as its source-level input.

A way to achieve secure compilation is through a *fully abstract* compiler. A fully abstract compiler preserves (and reflects) all abstractions of the source language in the target language, including security abstractions. So, when the target of a fully abstract compiler is assembly, the power of a low-level attacker is reduced to that of a source-level attacker.

Informally, a compiler is fully abstract when it translates indistinguishable source-level programs into indistinguishable target-level programs. Formally, a fully abstract compiler preserves and reflects *contextual equivalence* between source- and target-level programs. Two programs are contextually equivalent if they cannot be distinguished by a third one. For example, consider two instances P_1 and P_2 of the same program that contain two different values in the same variable v. Denote their fully abstract-compiled counterparts by $P_1^{\downarrow}$ and $P_2^{\downarrow}$. If P_1 and P_2 are contextually equivalent, then $P_1^{\downarrow}$ and $P_2^{\downarrow}$ must also be. So, if the content of v is *confidential* (e.g., the value stored in v could be `private` and never communicated), no program interacting with $P_1^{\downarrow}$ or $P_2^{\downarrow}$ can discern it. Notice that a fully abstract compiler does not eliminate source-level security flaws. It is, in a sense, conservative, as it introduces no more vulnerabilities at the target-level than the ones already exploitable at the source level.

The notion of fully abstract compilation also has consequences for code reasoning, as it allows *source-level reasoning*. When writing code, programmers need to consider both the source-level setting where the code is written and the target-level setting where the code runs. Source-level reasoning means that a programmer needs only to reason about the behaviour of source-level code. The target-level code is guaranteed to behave as its source-level counterpart even in the presence of target-level attacks.

With these premises, it is clear that fully abstract compilation is hard to achieve. Compilation of Java to JVM or of C# to the .NET framework are some of the examples where compilation is not fully abstract, as Kennedy [2006] presented. Nevertheless, techniques to achieve fully abstract compilation exist. They rely on address space layout randomisation [Abadi and Plotkin 2012; Jagadeesan et al. 2011], type-based invariants [Ahmed and Blume 2011; Fournet et al. 2013], and enhancing the target language with a protected module architecture [Agten et al. 2012; Patrignani et al. 2013].

This article describes a fully abstract compilation scheme from a secure high-level language to untyped assembly language enhanced with a protected module architecture. Protected module architectures offer a fine-grained, program counter-based memory access control mechanism that provides isolation for specific memory areas. This isolation mechanism is adopted to preserve source-level security properties in the output of the compiler. The target language is similar to a modern processor architecture, so the compilation scheme handles subtleties such as flags and registers. The source language is a component-based, Java-like object-oriented programming language that borrows extensively from the Java Jr. language of Jeffrey and Rathke [2005b], extending it with exceptions.

The article provides the following contributions:

—the presentation of a fully abstract compilation scheme for a strongly typed, component-based, object-oriented programming language to untyped assembly language;
—the highlighting of mistakes that make a naïve compilation scheme not fully abstract and their correction;
—the formalisation of both the source and the target languages and the proof of full abstraction of the compilation scheme; and
—the benchmarking of the overhead introduced by the fully abstract compiler.

The prototype of the compiler described in this article was implemented by means of additional LLVM passes [Lattner and Adve 2004].

The results of this article are expressed with respect to a specific target language adopting a specific protected module architecture. The lessons learned can be applied to target languages adopting different architectures up to minor technical changes. In fact, the article highlights what are the features that the protected module architectures needs to implement to provide the isolation mechanism necessary for the compilation scheme to be fully abstract.

The article first presents an informal overview of background notions such as the source and target languages (Section 2), then it describes the secure compilation scheme (Section 3). The article then formalises the source and target languages (Section 4) and provides the formal proof of full abstraction of the compilation scheme (Section 5). Then, the article presents the benchmarking of the overhead introduced by the secure compilation scheme (Section 6), followed by a discussion on its extensions and limitations (Section 7). Finally, the article discusses related work (Section 8), future work, and concludes (Section 9).

2. INFORMAL OVERVIEW

This section first presents a comparison between different target-level protection mechanisms (Section 2.1), then it informally introduces both the target and the source languages of the fully abstract compiler (Sections 2.2 and 2.3). This section then gives an overview of how we will carry out the proof of full abstraction of the compilation scheme of Section 3 (Section 2.4). Contextual equivalence is then presented as a good candidate for security policies enforcement through compilation (Section 2.5). Finally, this section present the threat model for this article (Section 2.6).

2.1. Low-Level Protection Mechanism

In order to safeguard low-level assembly programs from malicious attackers, a number of alternatives arise. In the following we review the three most interesting ones: address space layout randomisation, protected module architectures, and processes; a review of additional ones is left for Section 8. We do not give a thorough review of all existing low-level protection mechanism, the interested reader is referred to the PhD. thesis of Younan [2008] for an in-depth analysis of the subject. Alternatives to using low-level protection mechanism include using massive-scale, diversified software [Homescu et al. 2013; Larsen et al. 2014].

Address Space Layout Randomisation: ASLR. ASLR is a technique that randomises the memory layout of key data areas of a program such as the base of the stack, of the heap, of libraries, and so forth. The executable is divided in segments whose order is randomised by the dynamic linker (i.e., just before running the executable). This technique is used to hinder an attacker from mounting "return to libc" attacks, or from using previously acquired knowledge of the location of certain data to access that data.

ASLR has been used to achieve (probabilistic) fully abstract compilation of the simply typed λ-calculus [Abadi and Plotkin 2012] and of the same language extended with dynamic memory allocation, first and higher-order references and call/cc [Jagadeesan et al. 2011]. It requires a small trusted computing base (TCB) consisting of the randomisation function used to generate the different memory layouts.

The major drawback of ASLR (and of related techniques such as stirring [Wartell et al. 2012]) for secure compilation is that it relies on a good randomisation function, so it only provides probabilistic guarantees. Moreover, there have been successful derandomisation attacks to existing ASLR implementations [Shacham et al. 2004], which could be used to violate the security of a compiler relying on ASLR. Finally, certain attacks, such as the ones presented in Problem 5 in Section 3, focus on corrupting data instead of the control flow of execution. These are not simply solved with ASLR and require additional care, as discussed in Section 3.

Protected Module Architectures: PMA. PMA provides assembly code with the ability to create protected modules. Like their high-level counterpart, such as ML modules, these modules offer an interface mechanism to allow interoperation with code that is not in the module. Additionally, these modules *isolate* what is placed within the module boundaries.

The most common way to implement PMA is through program counter-based memory access control mechanisms [McCune et al. 2010, 2008; Noorman et al. 2013; Singaravelu et al. 2006; Strackx and Piessens 2012; McKeen et al. 2013]. We review these mechanisms from the work of Patrignani et al. [2013] and Strackx and Piessens [2012]. This mechanism logically divides the memory into a protected and an unprotected section. The protected section is further divided into a code and a data section. The code section contains a variable number of *entry points*: the only protected addresses to which instructions in unprotected memory can jump and execute. The data section is accessible only from the protected section. The size and location of each memory section are specified in a memory descriptor. The following table summarises the access control model enforced by the protection mechanism.

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

This protection mechanism provides a secure environment for code that needs to be protected from a potentially malicious surrounding environment. It is appealing in the context of embedded systems, where kernel-level protection mechanisms are often lacking. Very recently, Intel publicly announced Software Guard Extensions (Intel SGX), a hardware implementation of a Protected Module Architecture. Hence PMA support will be broadly available in mainstream processors within a few years, and any processor with PMA support can benefit from the secure compilation techniques proposed in this article. Moreover, secure compilation and PMA aid the verification of properties of C code placed inside a protected module [Agten et al. 2015]. Finally, the TCB of PMA can be very small [McCune et al. 2008; Singaravelu et al. 2006; Strackx and Piessens 2012; Strackx et al. 2010] or even zero-software [Eldefrawy et al. 2012; Noorman et al. 2013; McKeen et al. 2013]. The interested reader is referred to the work of Strackx et al. [2013] for further inquiries on this line of security research.

Processes. The same access control policy of PMA can be implemented by using operating system-level processes. In this setting, each process gets a different address space, which provides a protected memory where other processes have no access rights.

`movl` $\mathtt{r_d}$ $\mathtt{r_s}$	Load the word from the memory address in register $\mathtt{r_s}$ into register $\mathtt{r_d}$.
`movs` $\mathtt{r_d}$ $\mathtt{r_s}$	Store the contents of register $\mathtt{r_s}$ at the address found in register $\mathtt{r_d}$.
`movi` $\mathtt{r_d}$ `k`	Load the constant value `k` into register $\mathtt{r_d}$. Note that $\mathtt{k} < 2^\ell$.
`add` $\mathtt{r_d}$ $\mathtt{r_s}$	Write $\mathtt{r_d} + \mathtt{r_s} \bmod 2^\ell$ into register $\mathtt{r_d}$ and set the ZF flag accordingly.
`sub` $\mathtt{r_d}$ $\mathtt{r_s}$	Write $\mathtt{r_d} - \mathtt{r_s} \bmod 2^\ell$ into register $\mathtt{r_d}$ and set both the ZF and the SF flags accordingly.
`cmp` $\mathtt{r_1}$ $\mathtt{r_2}$	Calculate $\mathtt{r_1} - \mathtt{r_2}$ and set both the ZF and the SF flags accordingly.
`jmp` $\mathtt{r_i}$	Jump to the address located in register $\mathtt{r_i}$.
`je` $\mathtt{r_i}$	If the ZF flag is set, jump to the address in register $\mathtt{r_i}$.
`jl` $\mathtt{r_i}$	If the SF flag is set, jump to the address in register $\mathtt{r_i}$.
`call` $\mathtt{r_i}$	Push the value of the program counter +1 onto the call stack and jump to the address in register $\mathtt{r_i}$.
`ret`	Pop a value from the call stack and jump to the popped location.
`halt`	Stop the execution. We consider the value of register $\mathtt{r_0}$ the result of the execution.

Fig. 1. Instruction set $\mathcal{I}$ of language $\mathcal{A}$+I.

Different processes can communicate via message passing, effectively calling functions implemented in each other's address space. Additionally, shared memory can be used for larger data management.

There are two drawbacks for this approach. First, processes and interprocess communication are relatively heavyweight constructs to use. Crossing the boundary between one process and another is more costly than switching between protected and unprotected code (and vice-versa) in current PMA implementations. Additionally, the TCB of this protection mechanism includes the whole operating system, which is much larger than current PMA implementations.

The reduced TCB, the security benefits and the novelty in choosing it as a protection mechanism motivate the choice of PMA to protect the target language of the compilation scheme.

2.2. The Target Language $\mathcal{A}$+I, Informally

To model a realistic compilation scheme, the target language should be close to what is used by modern processors. For this reason, this article adopts $\mathcal{A}$+I (acronym of $\mathcal{A}$ssembly plus Isolation), a low-level language that models a standard Von Neumann machine consisting of a program counter, a registers file, a flags register and memory space [Agten et al. 2012; Patrignani et al. 2013; Patrignani and Clarke 2014], enhanced with a protected module architecture. The registers file contains 12 general-purpose registers $\mathtt{r_0}$-$\mathtt{r_{11}}$ and a stack pointer register SP, which points to the top of the *call stack*. The flags register contains two flags: a zero flag ZF and a sign flag SF, which are set when arithmetic operations output a zero or a negative number, respectively. The call stack is located in unprotected memory. Instructions executed by the language are listed in Figure 1. For the sake of simplicity, assume the architecture targeted by the language works with ℓ bit-long words, where ℓ is a power of 2. This allows the formalisation presented to scale to architectures with words of different sizes. A complete formalisation of $\mathcal{A}$+I is delayed until Section 4.1. The protection mechanism affects the semantics of the language, preventing the execution of certain instructions, in accordance with the PMA access control policy presented in Section 2.1. Following are some code snippets that exemplify the semantics of $\mathcal{A}$+I. In all examples concerning $\mathcal{A}$+I code, assume the presence of a protected memory section spanning from address 100 to 200, with a *single* entry point at address 100. Let P_s denote the code located in the protected section and P_u denote the code located in the unprotected one. Every instruction is preceded by the address where it is located; execution starts at address 0.

Example 2.1 (*No execution of code in the protected memory partition*). P_u initialises register $\mathtt{r}_0$ to 101 (line 1) and then jumps to that address (line 2).

```
1  0   movi r0 101   // unprotected code
2  1   jmp r0
3  ...
4  100   add r0 r1   // protected code
5  101   ret
```

Since address 101 is not an entry point of the protected memory section, the jump of P_u does not succeed and execution is `halted` (this can happen in a variety of ways, for example, in our prototype detailed in Section 6.1, the execution is suspended and trapped by the hypervisor [Strackx and Piessens 2012]).

Example 2.2 (*No reading/writing the protected code section*). P_u initialises register $\mathtt{r}_0$ to 101 (line 1) and register $\mathtt{r}_1$ to 20 (line 2), then it writes the content of $\mathtt{r}_1$ at the address in $\mathtt{r}_0$ (line 3).

```
1  0   movi r0 101   // unprotected code
2  1   movi r1 20
3  2   movs r0 r1
4  ...
5  100   add r0 r1    // protected code
6  101   ret
```

Since address 101 is protected, P_u cannot write there, so execution is `halted`, as in Example 2.1. Analogously, if the instruction of line 2 were replaced with `movl` $\mathtt{r}_0$ $\mathtt{r}_1$, the execution fails. In that case, P_u would be attempting to read the protected memory section, while it does not have that privilege.

Example 2.3 (*Interoperation between protected and unprotected code*). P_u initialises register $\mathtt{r}_0$ to 12 (line 1), register $\mathtt{r}_1$ to 10 (line 2), register $\mathtt{r}_5$ to 100 (line 3) and then calls to the protected function located at address 100 (line 4), storing address 4 on the call stack. P_s subtracts registers $\mathtt{r}_0$ and $\mathtt{r}_1$ (line 6) and, if the result is greater than or equal to zero, it returns that result (line 9). Otherwise, if the result is less than zero, P_s jumps to address 104 (lines 7 and 8), and returns 0 (lines 10 and 11). Execution then continues in unprotected memory at address 4 (line 5, omitted), which is the address popped from the call stack.

```
1   0   movi r0 12    // unprotected code
2   1   movi r1 10
3   2   movi r5 100
4   3   call r5
5   ...
6   100   sub r0 r1    // protected code
7   101   movi r3 104
8   102   jl r3
9   103   ret
10  104   movi r0 0
11  105   ret
```

2.3. The High-level Language $\mathcal{J}$+E, Informally

The high-level language adopted by the compilation scheme is $\mathcal{J}$+E (acronym of $\mathcal{J}$ava plus Encapsulation): a strongly typed, single-threaded, component-based, object-oriented language that enforces `private` fields and `public` methods. $\mathcal{J}$+E extends the Java Jr. language of Jeffrey and Rathke [2005b] with local variables and exceptions. It was chosen because it provides a clear notion of encapsulation for a high-level component, which makes for simpler reasoning about the secure compilation scheme.

$$
\begin{array}{lrl}
\textit{components} & C & ::= \overline{P} \\
\textit{packages} & P & ::= \{\textbf{package}\ p; \overline{D_i}\} \mid \{\textbf{package}\ p; \overline{D_e}\} \\
\textit{import declarations} & D_i & ::= \textbf{interface}\ i\ \textbf{extends}\ \overline{t}\ \{\overline{M_t}\} \\
 & & \mid \textbf{extern}\ o : t; \\
\textit{export declarations} & D_e & ::= \textbf{class}\ c\ \textbf{extends}\ t\ \textbf{implements}\ \overline{t}\ \{K\ \overline{F_t}\ \overline{M}\} \\
 & & \mid \textbf{object}\ o : t\ \textbf{implements}\ \overline{t}\ \{\overline{F}\} \\
\textit{constructors} & K & ::= c(\overline{f} : \overline{t}, \overline{f'} : \overline{t'})\ \{\textbf{super}(\overline{f}); \textbf{this}.\overline{f''} = \overline{f'}\} \\
\textit{fields} & F & ::= \textbf{private}\ f = v \\
\textit{field types} & F_t & ::= \textbf{private}\ f : t \\
\textit{methods} & M & ::= \textbf{public}\ m(\overline{x} : \overline{t}) : t\ \ [\textbf{throws}\ t]\ \ \{\textbf{return}\ E;\} \\
\textit{method types} & M_t & ::= \textbf{public}\ m(\overline{x} : \overline{t}) : t\ \ [\textbf{throws}\ t] \\
\textit{expressions} & E & ::= v \mid x \mid E.f \mid E.f = E \mid E.m(\overline{E}) \mid E\ \mathsf{op}\ E \mid \textbf{exit}\ E \\
 & & \mid E; E \mid E\ \mathsf{in}\ p \mid \textbf{var}\ x : t = E \mid \textbf{if}\ (E)\ \{E\}\ \textbf{else}\ \{E\} \\
 & & \mid \textbf{new}\ t(\overline{E}) \mid \textbf{try}\ \{E\}\ \textbf{catch}\ (x : t)\ \{E\} \mid \textbf{throw}\ E \\
\textit{types} & t & ::= p.c \mid p.i \mid p.c\ \mathsf{in}\ p \mid p.c\ \mathsf{in}\ * \mid \textbf{Obj} \mid \textbf{Unit} \mid \textbf{Bool} \mid \textbf{Int} \\
\textit{operations} & \mathsf{op} & ::= + \mid - \mid \cdot \mid / \mid \wedge \mid \vee \mid \cdots \\
\textit{values} & v & ::= p.o \mid \mathsf{unit} \mid \mathsf{true} \mid \mathsf{false} \mid n \mid \textbf{throw}\ v
\end{array}
$$

Fig. 2. Syntax of $\mathcal{J}$+E. Denote a sequence of elements $E_1, \ldots, E_n$ with $\overline{E}$.

$\mathcal{J}$+E supports many of the basic constructs one expects from a programming language (Figure 2). A program in $\mathcal{J}$+E is called a *component*, it is a collection of sealed packages that communicate via interfaces and public objects. $\mathcal{J}$+E partitions packages into *import* and *export* ones. Import packages are analogous to the `.h` header file of a C program.[1] They define *interfaces*, which are named collections of method signatures, and *externs*, which are references to externally defined objects. Export packages provide an implementation of an import package. They define *classes*, which are named collections of method implementations and fields (also known as instance variables), and *objects*, which implement classes and provide bindings from fields to values. Methods in $\mathcal{J}$+E can (optionally) throw exceptions. The top of the class hierarchy is Obj, the only primitive types in $\mathcal{J}$+E are `Unit`, inhabited by `unit`; `Bool`, inhabited by `true` and `false`; and `Int`, inhabited by word-sized integers. The expression E in p is borrowed from Java Jr. It is a type coercion that allows the following: if the expression E is well typed to run in package p with return type t, then the expression E in p is well typed to run in any package q with return type t, as long as t is a visible type in q.

Listing 1 illustrates the package system of $\mathcal{J}$+E. There, and in future code examples, we will massage the syntax of the presented examples for the sake of readability. It contains two package declarations: `P-Import` is an import package and `P-Export` is an export package implementing `P-Import`. `P-Export` provides class `AccountClass` that implements interface `Account` defined in `P-Import`. Object `extAccount` allocated in `P-Export` provides an implementation for the `extern` with the same name defined in `P-Import`.

[1]The kind of C programs one writes when learning C: devoid of preprocessor instructions, macros, and so forth.

```
1  package P-Import;
2   interface Account {
3     public createAccount() : Account;
4     public getBalance() : Int;
5   }
6   extern extAccount : Account;
7
8  package P-Export;
9   class AccountClass implements P-Import.Account {
10
11    AccountClass() {
12      this.counter = 0;
13    }
14    private counter : Int;
15
16    public createAccount() : P-Import.Account {
17      return new P-Export.AccountClass();
18    }
19    public getBalance() : Int {
20      return this.counter;
21    }
22    public resetAccount() : Unit {
23      this.counter = 0;
24    }
25  }
26  object extAccount : AccountClass { private counter = 0 }
```

Listing 1. Example of the package system of $\mathcal{J}$+E.

One of the security mechanisms of $\mathcal{J}$+E is given by `private` fields, which can be used to define security properties such as confidentiality and integrity, as they are not accessible from outside the class declaring them. In $\mathcal{J}$+E, classes are private to the package that contains their declarations. Objects are allocated in the same package as the class they instantiate. Due to this package system, for a package to be compiled, it only needs the import packages of any package it depends on. As a result, formal parameters in methods have interface types because classes that implement those interfaces are unknown. This discipline is called *programming to an interface*. Additionally, because constructors are not exposed in interfaces, cross-package object allocation happens through factory methods. For example, the name of class `AccountClass` from Listing 1 is not visible from outside package `P-Export`; thus, expressions of the form `new P-Export.AccountClass()` cannot be written outside `P-Export`.

2.4. The Proof of Full Abstraction of the Compilation Scheme, Informally

A fully abstract compilation scheme preserves and reflects contextual equivalence of source- and target-level programs. Informally speaking, two programs C_1 and C_2 are contextually equivalent if they behave the same for *all* possible programs they interact with. The programs that C_1 and C_2 can interact with are called *contexts*, they are (partial) programs with a hole. Once the hole is filled with either C_1 or C_2, the program is whole. For any context, if the behaviour of the whole program does not change when the hole is filled with either C_1 or C_2, then C_1 and C_2 are contextually equivalent [Plotkin 1977]. This notion is denoted as $C_1 \simeq C_2$.

Given a source-level program C_1, denote the target-level program that is produced by compiling it as $C_1^{\downarrow}$. Fully abstract compilation is formally expressed as $C_1 \simeq C_2 \iff C_1^{\downarrow} \simeq C_2^{\downarrow}$. To prove this statement, the equivalence is split in two cases:

—The direction $C_1^{\downarrow} \simeq C_2^{\downarrow} \Rightarrow C_1 \simeq C_2$ states that the compiler outputs target-level programs that behave as the corresponding source programs. This is what most compilers achieve, at times even certifying the result [Chlipala 2007; Leroy 2009]; we are not interested in this direction. This is thus assumed; the implications of this assumption are made explicit in Section 5.2.

—The direction $C_1 \simeq C_2 \Rightarrow C_1^{\downarrow} \simeq C_2^{\downarrow}$ states that source-level abstractions are preserved through compilation to the target level. Proving this direction requires reasoning about contexts, which is notoriously difficult [Ahmed and Blume 2011]. This is even more so in this setting, where contexts are low-level memories lacking any inductive structure. To avoid working with contexts, we replace the notion of contextual equivalence ($\simeq$) at the target level, with that of trace equivalence ($\simeq_{\mathrm{T}}$), which also provides an inductive principle to use in the proof. The replacement is possible because the two equivalences have been proved to coincide in $\mathcal{A}$+I [Patrignani and Clarke 2014]. This direction is thus restated as $C_1 \simeq C_2 \Rightarrow C_1^{\downarrow} \simeq_{\mathrm{T}} C_2^{\downarrow}$.

Trace equivalence is based on trace semantics, which describes the behaviour of a program in terms of a set of traces: sequences of actions the program can undertake [Jeffrey and Rathke 2005b]. Two programs are thus trace equivalent if their trace semantics coincide, that is, if their sets of traces are the same. This notion (or rather, its negation) is used to prove the contrapositive of this direction: $C_1^{\downarrow} \not\simeq_T C_2^{\downarrow} \Rightarrow C_1 \not\simeq C_2$.

In order to prove that C_1 and C_2 are not contextually equivalent, it suffices to show that there exists a source-level context that behaves differently depending on whether its hole is filled with C_1 or C_2. Such a context is said to *differentiate* C_1 from C_2. This proof relies on an algorithm that creates a source-level context, a *witness* that differentiates C_1 from C_2, given the different traces of their compiled counterparts $C_1^{\downarrow}$ and $C_2^{\downarrow}$.

This proof strategy is known in the field [Agten et al. 2012; de Boer et al. 2005; Jeffrey and Rathke 2005a; Patrignani and Clarke 2014]. However, the complexity of the proof resides in handling features of the high-level language such as strong typing, dynamic memory allocation, and exceptions.

2.5. Contextual Equivalence: A Security Perspective

Contextual equivalence plays a key role in the definition of a fully abstract compiler, yet what ensures that a fully abstract compiler is a secure one? This section briefly explains how contextual equivalence can be used for security policies enforcement [Abadi 1999; Abadi and Plotkin 2012; Agten et al. 2012; Ahmed and Blume 2011; Fournet et al. 2013; Jagadeesan et al. 2011; Kennedy 2006; Patrignani et al. 2013].

In $\mathcal{J}$+E, fields are `private`, so every allocated object defines a secret state: the contents of its fields. Some objects can thus be indistinguishable from an external point of view even though their states differ: they are contextually equivalent.

Contextual equivalence can be adopted to state security properties such as confidentiality, integrity and invariant definition, as the code examples of Figure 3 show [Abadi and Plotkin 2012; Agten et al. 2012]. In the examples, classes are annotated with subscripts for identification but their names are meant to be the same. Assume the presence of an external object of type `External`, which presents a method `callback()`.

Class $\mathtt{C}_L$ and $\mathtt{C}_R$ differ in the value stored in `secret` after a call to `setSecret()`. If they are contextually equivalent, then no program interacting with either of them can infer the value of `secret`. This is a *confidentiality* property. Additionally, $\mathtt{C}_L$ checks whether changes to `secret` have been made during the call to `external.callback( )` in method `proxy()`. If $\mathtt{C}_L$ and $\mathtt{C}_R$ are contextually equivalent, then the call to `external.callback( )` does not modify the value of `secret`—this is an *integrity* property. Finally, when `min>max`, method `invariantCheck()` of $\mathtt{C}_L$ will return 1. If $\mathtt{C}_L$ and $\mathtt{C}_R$ are contextually equivalent, then the invariant $\mathtt{min} \leq \mathtt{max}$ can never be violated. This is an *invariance* property.

From the high-level language perspective, contextual equivalence holds for the presented examples. However, when these examples are run, they are compiled with an insecure compiler to untyped assembly code. As the assembly code is untyped and it

```
1  package p;
2  class C_L {
3    private secret, min, max : Int = 0;
4
5    public setSecret() : Int {
6      secret = 0;
7      return 0;
8    }
9    public proxy( external : External ) :
         Int {
10     secret = 0;
11     external.callback();
12     if ( secret == 0 ) {
13       return 0;
14     }
15     return 1;
16   }
17   public invariantCheck() : Int {
18     if ( min ≤ max ) {
19       return 0;
20     }
21     return 1;
22   }
23 }
```

```
1  package p;
2  class C_R {
3    private secret, min, max : Int = 0;
4
5    public setSecret() : Int {
6      secret = 1;
7      return 0;
8    }
9    public proxy( external : External ) :
         Int {
10     secret = 0;
11     external.callback();
12
13     return 0;
14
15
16   }
17   public invariantCheck() : Int {
18
19     return 0;
20
21
22   }
23 }
```

Fig. 3. Example of contextually equivalent $\mathcal{J}$+E code.

allows jumps to all addresses in memory, an attacker with assembly code injection privileges can violate the security properties of these examples [Younan et al. 2012]. If these classes are compiled with the secure compilation scheme of Section 3, their security properties cannot be violated, as contextual equivalence is preserved at the target level.

2.6. Threat Model

Having introduced all related notions, this section first gives an informal presentation of the threat model considered in this article, followed by a more precise definition of the elements that constitute the threat model.

The threat model represents an attacker with kernel-level code injection privileges introducing Malware into a software system. Kernel-level code injection is a critical vulnerability of complex software system where injected code operates with kernel-level privileges, and it can thus bypass all existing software-based security mechanisms. Notorious examples include OpenBSD's IPv6 remote kernel buffer overflow[2] and a buffer overrun in JPEG processing of Microsoft applications.[3] An attacker who exploits such a vulnerability injects code that can violate the security property of the whole software system and disclose confidential data, disrupt running applications, and so forth. The attacker's aim is in fact to violate the security policy of existing software running in the system, by means of the injected $\mathcal{A}$+I code. The injected code is also not subject to most source-level restrictions such as well-typedness. For this reason, assume the attacker is injecting $\mathcal{A}$+I code. Let us now define the system under attack.

The system under attacked is assumed to be equipped with a single PMA instance that provides one protected memory partition (a protected module). The PMA instance is responsible for enforcing the access control policy of Section 2.1 on the whole memory. The instance is assumed to be beyond the reach of $\mathcal{A}$+I code, so it cannot be tampered with by the attacker. This assumption seems reasonable because most PMA implementation have small TCBs that can be verified for the absence of exploitable

[2]http://www.securityfocus.com/archive/1/462728/30/150/threaded.
[3]https://technet.microsoft.com/library/security/ms04-028.

vulnerabilities. A compromised PMA would render all security guarantees void, as the attacker would be able to circumvent its access control policies.

Given this system, it is desirable to guarantee that at least the software within the protected module is secure. These security properties are defined in the source language of the software. In the case of $\mathcal{J}$+E code, these security properties include at least the following:

(1) confidentiality and integrity of field contents, of object names and of method bodies;
(2) no control flow alteration apart from when using exceptions; and
(3) nonreachability of stuck (error) program states.

The first point defines *secrets* in $\mathcal{J}$+E software. Method bodies can reveal certain of those secrets, such as field contents and object names, by returning them. Secrets can be discerned by an attacker *only* if methods reveal them—no other ways should be exploitable by an attacker to discern a secret. Because method bodies are confidential and there is no way to reveal them, there should be no way for an attacker to discern two different implementations of the same function. Problem 1 presents an example of how a naïve compiler would violate the confidentiality of field contents, more of these examples are presented throughout Section 3.

PROBLEM 1 (STACK SECURITY). *Consider two classes that define a* `secret` *field with different values and the same method* `doCallback` *that inputs an object and calls method* `callback` *on it. These two classes are implemented by two objects:* o_L *and* o_R*. Assume the presence of an external object of type* `External`*, which presents a method* `callback()`*.*

```
1  package p;
2  class C_L {
3    private secret : Int = 0;
4
5    public doCallback( cb : External ) :
         Int {
6      var x : Int = secret;
7      cb.callback();
8      return 0;
9    }
10 }
11 object o_L : C_L
```

```
1  package p;
2  class C_R {
3    private secret : Int = 1;
4
5    public doCallback( cb : External ) :
         Int {
6      var x : Int = secret;
7      cb.callback();
8      return 0;
9    }
10 }
11 object o_R : C_R
```

Objects o_L *and* o_R *are equivalent at the source level, but their compiled counterparts are not. Because local variables are placed on the call stack (in unprotected memory) and a low-level attacker can read unprotected memory, she can read the value of* `x` *during the callback* `cb.callback()`*. Variable* `x` *contains the value of* `secret`*, which is a* `private` *field and which is different for both objects.*

A naïve compilation scheme does not entail the confidentiality or integrity of the call stack, which allows attackers to read and write local variables. An attacker can use this vulnerability to read secrets from the stack, similarly to a buffer-overread attack [Strackx et al. 2009]. Alternatively, she can even tamper with the control flow by overwriting a return address on the stack, similarly to a return address clobbering attack [Erlingsson et al. 2010].

The second point ensures that the only way to divert the flow of execution is through exceptions. Since exception throwing must be specified in method interfaces, critical functionalities can be implemented so that they are carried out in their entirety: if the right exceptions are caught, no other way of diverting the execution flow exists.

The third point is similar to the second one: there is no way of disrupting some functionalities by supplying arbitrary parameters to method calls.

The presence of a protected module provides a basic protection of some of the aforementioned properties, but not all of them are covered. The adoption of a secure compiler for the software to be placed in the protected memory should ensure that all aforementioned security properties are enforced in that software. By defining such a secure compiler as a fully abstract compiler, we capture exactly the preservation of those properties in the generated target code: a fully abstract compiler makes the software in the protected memory secure.

For a more precise treatment, the threat model consists of the following definitions: the system under attack (Definition 2.4), the security property of the system (Definition 2.5), and the attacker to the system (Definition 2.6).

Definition 2.4 (*System under attack*). The system is a Von Neumann machine with a flat address space and *one* PMA instance that provides a *single* protected partition in memory (a protected module). The protected module contains $\mathcal{A}$+I code, called the *protected code*, that complies to $\mathcal{J}$+E specifications, so the protected code behaves like $\mathcal{J}$+E code. The unprotected memory contains arbitrary $\mathcal{A}$+I code.

In the system under consideration, for the sake of simplicity, only compiled $\mathcal{J}$+E software is considered to be present. Moreover, only one protected module is assumed to be present. A single module suffices to protect the concerns of a single user or of multiple users that trust each other. Addressing the challenges of adopting multiple protected modules in the system, each belonging to mutually distrusting stakeholders, is left for future work.

Definition 2.5 (*Security property*). The protected code behaves as its $\mathcal{J}$+E specifications and in no other way.

This property has the security implications described earlier because it is applied to the $\mathcal{J}$+E language. In different languages, this property may not have the same security relevance. For example, this property in the context of the C language would not entail confidentiality and integrity, as any structure can be inspected by virtue of simple pointer arithmetic.[4]

Definition 2.6 (*Attacker*). The attacker can arbitrarily change the state of the unprotected partition of the same memory of the protected module. The attacker also knows how to interact with the secure module.

The attacker is assumed to know the interfaces implemented by the protected code: the location of each entry point, the types each function expects, and the addresses of eventual static objects. Knowledge of the functionalities of existing software in the system is given to the attacker so the injected code can interact (possibly safely) with existing software.

Limitations. This threat model does not cover all possible security threats that the system is subject to, as exemplified in the following. Source-level security violations (e.g., a method returning a field-stored private key that was supposed to be secret) are not considered in this article. These violations should not be countered at the compiler level, but with source-level artefacts such as type systems. Availability attacks (e.g. unprotected code that never calls protected code) are also not considered, since the attacker is assumed to interact with the software to be protected in order to violate its security properties. Finally, side channel attacks are also not considered. The definition of the attacker's power, in fact, limits the kind of attacks she can mount. The attacker

[4]The C standard states that the behaviour of these scenarios is "undefined," but most C compilers allow arbitrary pointer arithmetic.

cannot exploit covert channels to mount side-channels attacks such as timing attacks, since these attacks cannot be expressed with $\mathcal{A}$+I code.

This article now describes how to develop a secure compiler that counters the threat imposed by the aforementioned attacker.

3. SECURE COMPILATION OF $\mathcal{J}$+E

This section first describes the general structure of the secure compilation scheme for $\mathcal{J}$+E. Then it presents details of the secure compiler in three parts, all of which introduce a series of naïve mistakes to secure compilation of a certain feature and then correct them. The first feature to be presented is callbacks (Section 3.1), the second one is dynamic memory allocation (Section 3.2), and the final one is exceptions (Section 3.3).

For the sake of simplicity, we start by developing a secure compiler for a core of $\mathcal{J}$+E. In the following, assume no dynamic memory allocation (i.e., no `new` expressions) and no presence of exceptions (i.e., no `try/catch` blocks and no `throw` e expressions). Since no new objects are created at runtime, components use static objects and externs for now.

We follow some standard conventions about how objects are compiled [Ducournau 2011]. The compilation of a $\mathcal{J}$+E component C outputs a *protected module* $C^{\downarrow}$, written in $\mathcal{A}$+I, consisting of a partial memory space and a memory descriptor. The program $C^{\downarrow}$ interacts with should not be able to distinguish modules just by their size, so, a constant amount of memory is reserved for each protected module, independent of the actual memory space required. The protected module is placed in protected memory and the memory descriptor divides the reserved space over the code and the data section. The stack pointer register is set up by the context and is pointing to free space in unprotected memory.

The compilation process consists of translating each package, class, object, interface, and method of the input component. To prevent a low-level module from being distinguished by the order of its methods in memory, packages, interfaces, methods in interfaces, classes, objects and externs are sorted alphabetically. Methods that do not appear in interfaces are compiled based on the order of occurrence in the class.

When an object is compiled, a word is reserved to indicate its class, which is used to dynamically dispatch methods. Methods are dispatched based on offsets through the v-tables, which associate class and method offsets with the corresponding method body. For each object, fields are then given a unique index number starting at 1, based on their order of occurrence. For a field f_i, one word of memory is reserved at the i-th memory address of the memory section of a given object. `Int`eger-typed constants are translated to their corresponding numeric value, `unit` is translated to 0, and `true` and `false` are translated to 1 and 0, respectively.

To translate a method body, the compiler processes each expression in turn, translating it into a list of behaviourally equivalent instructions. Registers $\mathtt{r}_0$ to $\mathtt{r}_3$ are used as general working registers, return values are passed through $\mathtt{r}_0$. In a method call at the target level, register $\mathtt{r}_4$ identifies the current object (`this`). Method calls are limited to seven parameters, which are passed through registers $\mathtt{r}_5$ to $\mathtt{r}_{11}$. These choices are not critical, the compiler may use registers in a different way and still be fully abstract. A calling convention is set so that register $\mathtt{r}_0$ contains the address where to return after a call at the target level (i.e., any jump instruction between the protected and unprotected sections, located at address x is preceded by an instruction `movi` $\mathtt{r}_0$ $\mathtt{x}+1$). This restriction simplifies the proof of full abstraction of the compilation scheme, we envisage it can be lifted at the price of complicating the proof, without making it unprovable.

Parameters and local variables are given a method-local index number. For each translated method body, a prologue and an epilogue are prepended and appended to

it. The prologue allocates and initialises a new activation record on the call stack, the record contains local variables and parameters for the method body. The epilogue deallocates the activation record when the method is done. The code of prologues, epilogues, and method bodies is placed in free space in the code section.

To support programming to an interface, and because protected memory can be entered only through entry points, a *method entry point* is generated for each interface-defined method. The entry point for the i-th method is placed at address $i * 128$ of the code section. The offset of 128 memory words is chosen arbitrarily, with the only condition that there is enough space between entry points to perform a number of simple operations, as will be described in Section 3.1. Each entry point forwards the call to the actual method body, so code at entry points consists of two parts: (1) a call to the method's body and (2) a return instruction. When the call to the body returns, the return instruction returns control to the location from which the entry point was called.

In order to specify how the component interacts with external code, assume the component being compiled provides one import package without a corresponding export one. Refer to this package as *the distinguished import package* (DIP). The DIP contains interface and extern definitions, thus callbacks are calls to methods defined in the DIP, on externs defined in the DIP. Component code is not supposed to implement interfaces defined in the DIP, as those are functionalities it requires of external programs. External code that provides an implementation for the DIP can implement interfaces defined in the component. This can lead to the dynamic dispatch procedure being called on objects that are not in the protected memory partition. When this case is detected, the compiled component must behave as in the case of a callback. The address where protected code must jump when performing a callback is assumed to be known based on a calling convention set up between protected and unprotected code.

So, a *return entry point* is generated to support returning from a callback. To perform a callback, the actual return address is first pushed on the call stack, then the address of the return entry point is pushed on the call stack. Control is then transferred to unprotected memory. When the code in unprotected memory returns from the callback, control will first be transferred to the return entry point, which will then subsequently return back to the actual return address.

The compilation scheme as just described ensures that a module is exited either through a callback, or through the return statement at the end of an entry point. Therefore, the second part of each method entry point is named *exit point*.

To provide a better understanding of the compilation scheme thus far, Figure 4 presents the memory layout of the compiled counterpart of the code of Listing 1.

3.1. Secure Compilation of Callbacks

The compilation scheme places code and data of the compiled object in the protected memory partition and configures the entry points of the protected memory partition to forward calls to method implementations. This scheme is sound, in the sense that two nonequivalent $\mathcal{J}$+E components will be compiled into two nonequivalent $\mathcal{A}$+I programs. It also provides some basic protection; for instance, a target-level attacker cannot just scan memory to find the values of object fields, as this is prevented by the protected module architecture. However, this scheme fails to be secure—this is shown by means of counterexamples.

3.1.1. Limitations of the Compilation Scheme. The compilation scheme defined so far is not fully abstract, as illustrated by the following series of examples; Problem 1 also arises in this context. In these examples, $\mathcal{A}$+I modules can be differentiated, while their $\mathcal{J}$+E

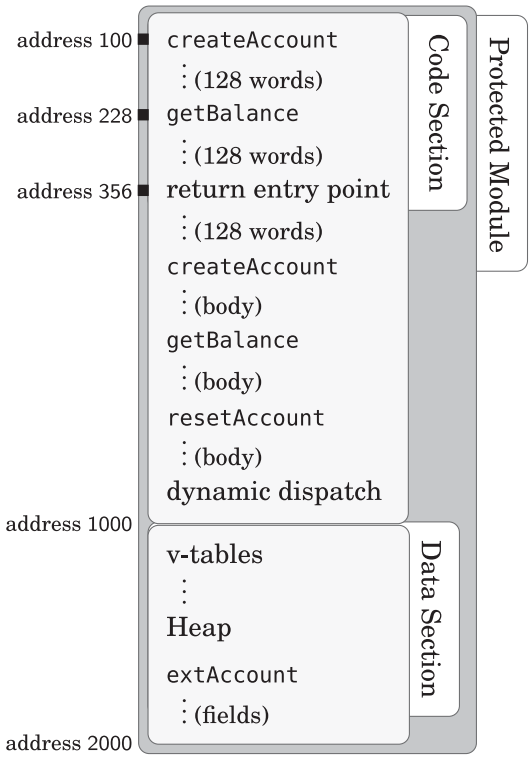

Fig. 4. Memory layout of the compilation of code from Listing 1. ■ indicate entry points. The protected memory partition spans from address 100 to address 2000, the code section spans for 900 addresses.

counterparts cannot. As before, classes and objects are annotated with subscripts for identification but their names are meant to be the same.

PROBLEM 2 (INFORMATION LEAKAGE). *Consider two classes that define the same method* `testVariable`*, which assigns different values to a local variable, tests whether that value is 0, and returns 0 in either case. These two classes are implemented by two objects:* o_L *and* o_R*.*

```
1  package p;
2  class C_L {
3    public testVariable() : Int {
4      var x : Int = 0;
5      if ( x == 0 ) {
6        return 0;
7      } else {
8        return 0;
9      }
10   }
11 }
12 object o_L : C_L
```

```
1  package p;
2  class C_R {
3    public testVariable() : Int {
4      var x : Int = 1;
5      if ( x == 0 ) {
6        return 0;
7      } else {
8        return 0;
9      }
10   }
11 }
12 object o_R : C_R
```

Objects o_L *and* o_R *are equivalent in* $\mathcal{J}$+E*, as method* `testVariable` *always returns* 0 *for both. However, a low-level attacker can differentiate their translations, due to the equality test in the condition of the if-statement. This test sets the* ZF *flag in* o_L *and clears it in* o_R*. This example illustrates that the flags register can leak information. Information can also be leaked through the general-purpose registers* $\mathtt{r}_0$ *to* $\mathtt{r}_{11}$ *or through the stack pointer register* SP.

PROBLEM 3 (VALUE OF BOOLEANS). *Consider two classes that provide a method* `identBool` *that inputs a* `Boolean` *value and returns* `true` *if that value is* `true` *or false otherwise. These two classes are implemented by two objects:* o_L *and* o_R.

```
1  package p;
2  class C_L {
3    public identBool( x : Bool ) : Bool {
4      if( x == true ){
5        return true;
6      }
7      return false;
8    }
9  }
10 object o_L : C_L
```

```
1  package p;
2  class C_R {
3    public identBool( x : Bool ) : Bool {
4
5      return x;
6
7
8    }
9  }
10 object o_R : C_R
```

The two classes implement this method differently, yet the behaviour is the same, so objects o_L *and* o_R *are equivalent in* $\mathcal{J}$+E. *Their* $\mathcal{A}$+I *translations however are not, because a low-level attacker can use any low-level value for parameter* `x`.

This problem arises whenever the source language is high-level and it has primitive types inhabited by a reduced number of values, such as `Booleans` *and* `Unit` *[Agten et al. 2012; Fournet et al. 2013]. This problem is similar to a full abstraction failure for the .NET C# compiler reported by Kennedy [2006], where the Boolean type is two valued in C# but is byte valued in the .NET virtual machine.*

3.1.2. Secure Compilation of Callbacks. To counter the vulnerabilities described earlier, the compilation scheme is enhanced in the following ways.

Stack security. The compiler must ensure the confidentiality and integrity of variables and control structures on the call stack. Instead of storing the entire stack in unprotected memory, it is split into an unprotected stack in unprotected memory and a secure stack in the data section of the protected memory section. The protected module output by the compiler places its activation records exclusively on the secure stack. The secure stack is given a large enough upper bound so that most programs can run without overflowing it. If the stack is overflowed, all registers and flags are cleared and the execution `halts`.

At the start of each entry point, the stack is switched to the secure stack. When leaving the protected module, the stack is restored to its previous address, which is checked to be in unprotected memory. When returning from protected to unprotected code, the return address is also checked to be in unprotected memory. A callback is performed by first pushing the actual return address onto the secure stack. Then, $\mathtt{r}_4$ is stored in the secure stack so as to be able to restore `this` to the right value once the callback returns. Next, the stack is switched to the unprotected stack and the address of the return entry point is pushed onto it. Control is then transferred to the context. When the callback returns, control will first be transferred to the return entry point, which switches back to the secure stack. Then it restores `this` to the expected value and it transfers control back to the actual return address. Because data is written to the unprotected stack during this process, the compiler must ensure that the location of the unprotected stack is valid before it writes to it. That is, the address of the unprotected stack must lie outside of protected memory, for otherwise parts of protected memory might get overwritten.

The context could tamper with the control flow by jumping to the return entry point when there is no callback to return from. To prevent this, the compiler initialises the first location of the secure stack to the address of a procedure clears all registers and flags and then `halts`. The return entry point will jump to this address if it is called when there is no callback to return from.

Table I. Pseudocode of Entry Point Routines

Method p entry point		Preamble to returnback entry point	
1	Switch stack to the protected one	a	Switch stack to the unprotected one
2	Check Bool-typed parameters	b	Clear flags and unused registers
	(*run method p code*)		(*run callback code*)
Exit point		Returnback entry point	
3	Clear flags and registers $\mathtt{r}_1 - \mathtt{r}_{11}$	c	Switch stack to the protected one
4	Switch stack to the unprotected one	d	Check `Bool`-typed parameters

Information leakage. In $\mathcal{J}$+E, the only way for two objects to communicate is through well-typed method calls and returns. The compiler must ensure that a low-level attacker cannot use any other communication channels, as this might leak information that should be kept private to the protected module.

The model of $\mathcal{A}$+I inherently provides three ways to exchange information: (1) through unprotected memory, (2) through the flags register and (3) through the general-purpose registers $\mathtt{r}_0$ to $\mathtt{r}_{11}$ and SP. Method (1) is already precluded, as compiled programs never write in unprotected memory. The SP register does not convey private information, because it is restored to the location of the unprotected stack whenever control leaves the protected module. The compiler constrains methods (2) and (3) as follows:

—Flags are cleared at each callback and exit point.
—Every general-purpose register except $\mathtt{r}_0$ is cleared at each exit point.
—Every general-purpose register is cleared at each callback, except if it is used for passing a parameter.

The compiler generates code at each entry point to enforce these constraints.

Value of Booleans. The compiler must ensure that all memory locations corresponding to $\mathcal{J}$+E fields and variables contain only values for which there is a corresponding $\mathcal{J}$+E value. The only values of type `Bool` are `true` and `false` and the corresponding $\mathcal{A}$+I values are 1 and 0. The compiler enforces this constraint by adding a runtime check at each entry point, to verify that the value of any `Bool`-typed parameter is either 0 or 1. An analogous check is added at each callback to a method with return type `Bool`. The same checks are introduced for type `Unit`, inhabited by `unit` in $\mathcal{J}$+E, which is compiled to 0 in $\mathcal{A}$+I. If any check fails, all registers and flags are cleared and the execution `halt`s.

These checks are needed for all ground types inhabited by a number of values that do not fit a $\mathcal{A}$+I word representation. This is why `Integer`-typed values are not checked, because any $\mathcal{A}$+I word can map to a $\mathcal{J}$+E integer.

In the light of these additions to the compilation scheme, Table I presents the pseudocode routine that is executed at entry points.

3.2. Secure Compilation of Dynamic Memory Allocation

To present secure compilation of the dynamic memory allocation feature, from this section onward, source-level components can contain expressions of the form `new` $t(\overline{v})$. For this, a secure heap is created: it is a heap located in the protected data section, where the objects dynamically created by the protected program will be allocated. As for the secure stack, the secure heap is given a large enough upper bound so that most programs can run without exhausting it. If the heap is exhausted, all registers and flags are cleared and the execution `halt`s.

The translation of those expressions causes a free location of the secure heap to be initialised for the newly created object. As previously stated, a word is reserved to

indicate the translated representation of its type t and a word is reserved for each of its fields.

3.2.1. Limitations of the Compilation Scheme. The compilation of `new` $t(\overline{v})$ expressions as stated earlier is not secure, as described in the following examples.

PROBLEM 4 (TYPE OF THE RECEIVER). *Consider two packages that provide a class with an unaccessible secret and a class that implements* `Pairs` *of* `Objects` *with a method to get the first element of the pair.*

```
1  package p;
2  class Pair_L {
3    private fst, snd : Obj = null;
4    public getFirst(): Obj {
5      return this.fst;
6    }
7  }
8  class Secret_L {
9    private secret : Int = 0;
10 }
11 object o_L : Secret_L
```

```
1  package p;
2  class Pair_R {
3    private fst, snd : Obj = null;
4    public getFirst(): Obj {
5      return this.fst;
6    }
7  }
8  class Secret_R {
9    private secret : Int = 1;
10 }
11 object o_R : Secret_R
```

The value of `secret` *cannot be leaked at the* $\mathcal{J}$+E *level; however, the compiled counterparts of those packages can leak its value. A low-level attacker can perform a call to method* `getFirst()` *with current object* o_L *or* o_R*. This will return the* `secret` *field, since fields are accessed by offset. As* $\mathcal{A}$+I *code is untyped, nothing prevents this attack from happening.*

PROBLEM 5 (TYPE OF THE ARGUMENTS). *Similarly to Problem 4, arguments of methods can be exploited in order to mount a low-level attack. Extend both packages from Problem 4 with the class* `ProxyPair`*, which has a method* `takeFirst` *that inputs a* `Pair` *and returns the output of the method* `getFirst` *called on the input.*

```
1  ...
2  class ProxyPair {
3    public takeFirst( v : Pair ): Obj {
4      return v.getFirst();
5    }
6  }
```

In $\mathcal{J}$+E*, both packages are still equivalent and no additional attacks can be mounted. However, a low-level attacker can pass an object of type* `Secret` *as argument to method* `takeFirst( )` *and the code will leak the contents of field* `secret`*.*

PROBLEM 6 (LEAKAGE OF OBJECT REFERENCES). *Consider two packages with a class* `Secret` *implemented by two objects and with different implementations of method* `createSecret`*. While that method in* $\texttt{Secret}_L$ *returns a new object, in* $\texttt{Secret}_R$ *it allocates two objects and returns one of them.*

```
1  package p;
2  class Secret_L {
3    private secret : Int = 0;
4    public createSecret() : Secret {
5
6      return new Secret();
7    }
8  }
9  object o_L1 : Secret_L
10 object o_L2 : Secret_L
```

```
1  package p;
2  class Secret_R {
3    private secret : Int = 0;
4    public createSecret() : Secret {
5      var x : Secret = new Secret();
6      return new Secret();
7    }
8  }
9  object o_R1 : Secret_R
10 object o_R2 : Secret_R
```

Object references at the low-level are the address where objects are allocated. Once $\texttt{p}_L$ *and* $\texttt{p}_R$ *are compiled, an attacker can discover that, for example, the identities of* o_L1

and o_R1 *are* 100*, while those of* o_L2 *and* o_R2 *are* 104*. After method* `createSecret` *is executed, an attacker can see that* `createSecret()`$_L$ *returns* 108 *and* `createSecret()`$_R$ *returns* 112*. With this knowledge, the attacker can guess that* p_R *created an additional object at address* 108*.*

The attacker can thus call methods on objects it does not know of by guessing the address where an object is allocated. Passing object addresses from a secure program to an external one can give away the allocation strategy of the compiler, as well as the size of allocated objects. An attacker that learns this information can then use it to mount attacks such as those presented in Problem 4 and 5. From a technical point of view, this means that leaking object addresses and accepting guessed addresses breaks full abstraction of the compilation scheme.

3.2.2. Secure Compilation of Dynamic Memory Allocation. To counter the vulnerabilities described earlier, the compilation scheme is enhanced in a number of ways. Since the countermeasure to Problem 6 affects the others, it is presented first.

Object identity. To mask low-level object identities, a data structure $\mathcal{O}$ is added to the protected code section. It is a map between natural numbers and low-level object identities that have been passed to external code. Such object identities that are passed to external code are added to $\mathcal{O}$ right before they are passed outside. The index in the data structure is then passed in place of the object identity, the same index must be passed whenever an already recorded object is passed. Indices in $\mathcal{O}$ are thus passed in a deterministic order, based on the interaction between external and internal code. Code at entry points is responsible for retrieving object identities from $\mathcal{O}$ before the actual method call. Access and retrieval of entries in $\mathcal{O}$ is very fast, and it can be implemented in $O(1)$. As the only objects in the data structures are the ones the attacker knows, she cannot guess object identities. This does not hamper the functionality of external code, as it can only call methods on objects.

Consider for example the right code snippet in Problem 6. There, o_R1 and o_R2 are given index 0 and 1 and they are added in $\mathcal{O}$ at compile time (since they are static objects). Once method `createSecret` is called, two objects are created. However, the object saved in variable `x` is not returned, so it is not added to $\mathcal{O}$. The other object is, so it is added to $\mathcal{O}$. Thus, at the $\mathcal{A}$+I level, the compiled counterpart of `createSecret` will return 2 the first time it is called in both p_L and p_R.

Entry points. Table II describes the code executed at entry points. Both method entry points and the return entry point are logically divided in two parts; they maintain the functionalities introduced in Section 3.1 and expand them as follows. The first part performs the checks described in the following text and then jumps either to the code that performs the dynamic dispatch or to the callback. The second part returns control to the location from which the entry point was called.

For method calls to be well typed, the code at entry points performs dynamic typechecks. This checks that a method is invoked on objects of the right type (line 2), with parameters of the right type (line 4), addressing Problems 4 and 5. Similar checks are executed when returning from a callback, in the returnback entry point (line f). These checks are performed only on objects whose class is defined in the compiled component, as they are allocated in protected memory; no control over externally allocated objects can be assumed. Dynamic typechecks are performed based on the word that indicates the class of a compiled object, that value is checked to comply with the value the method expects.

Resetting flags and registers are as in Section 3.1. If any check fails, all registers and flags are cleared and the execution `halts`.

Table II. Extension to the Pseudocode Executed at Entry Points Presented in Table I

Method p entry point		Preamble to returnback entry point	
1	Load receiver $v = \mathcal{O}(\mathtt{r}_4)$	a	Push current object $v = \mathtt{r}_4$, return address a and return type m
2	Check that v's class defines method p	b	Reset flags and unused registers
3	Load parameters $\bar{v}$ from $\mathcal{O}$	c	Replace object identities with indexes in $\mathcal{O}$
4	Dynamic typecheck on $\bar{v}$	d	Jump to callback address
5	Perform dynamic dispatch (*run method p code*)		(*run external code*)
Exit point		Returnback entry point	
6	Reset flags and unused registers	e	Pop return type m and check it
7	Replace object identities with indexes in $\mathcal{O}$	f	Dynamic typecheck
		g	Pop return address a, current object v and resume execution

Loading means that a value is retrieved from the memory, push and pop are operations on the secure stack.

A convention between protected and unprotected code is needed in order to identify indexes in $\mathcal{O}$ from unprotected addresses. For this, assume that the leftmost bit of a word is 1 if it denotes an index in $\mathcal{O}$.

3.3. Secure Compilation of Exceptions

To present secure compilation of exceptions, from this section onward, source-level components can contain `try/catch` blocks and `throw` e expressions. Method signatures can specify if the method throws a particular exception.

Secure compilation of languages supporting exceptions must handle the difficulties that result from the modification of the flow of execution of a program. This can be modified when a part of a program `throws` an exception and another part `catches` it. Exception handling can be implemented by modifying the runtime of the language so that it knows where to dispatch a thrown exception. Activation records are responsible for pointing to the exception handlers in order to propagate a thrown exception to the right handler.

```
1  package P-Exc;
2    class AccountTest {
3      public withdraw() : Unit {
4        try{
5          new P-Exc.EmptyAccount().getBalance();
6        } catch ( e : P-Exc.NoMoneyException ) {
7          // handle e
8        }
9      }
10   }
11   class EmptyAccount {
12     public getBalance() : Unit throws P-Exc.NoMoneyException {
13       throw new P-Exc.NoMoneyException();
14     }
15   }
16   class NoMoneyException implements Throwable {···}
```

Listing 2. Example of exceptions usage.

In Listing 2, the `catch` block of method `withdraw()` in class `AccountTest` defines a handler for exceptions of type `NoMoneyException`. When the activation record for `withdraw()` is allocated, the handler is registered in the related allocation record. When an exception of type `NoMoneyException` is thrown, the stack is traversed to find

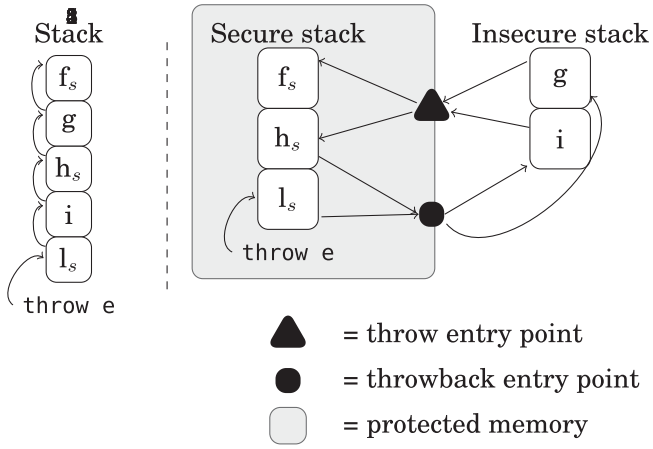

Fig. 5. Comparison of ways to handle exceptions.

the closest handler for exceptions of type `NoMoneyException`. As activation records are traversed and a handler is not found, those records are popped from the stack.

To implement throwing an exception in secure code that is caught in insecure code (or vice versa), throwing is securely compiled as callbacks (or calls). Thus, two additional entry points are created: the *throw entry point* and the *throwback entry point*. These entry points forward calls to the secure and insecure exception dispatchers, respectively. The secure exception dispatcher traverses the secure stack looking for handlers for the thrown exception. After an activation record has been inspected and deallocated, if the "next" allocation record to be inspected is in unprotected memory, the exception is forwarded to the external code through the throwback entry point.

Since exceptions are objects, in order to prevent exploits as in Problem 4, the throwback entry point must remember internally allocated exceptions that are thrown to external code. So, data structure $\mathcal{O}$ is used to register leaked exceptions as well as leaked object identities. This prevents external code from passing a nonexisting object identity to the secure exception handler in place of the object identity of an exception, effectively throwing a nonexistent exception.

For protected code to catch unprotected exceptions (and vice-versa) based on the (unprotected) interface type declared by the thrown object, some modifications are needed. A table is added in the protected code section, where unprotected object ids are associated to their known interface types. When an unprotected object is made known to the protected code (as a parameter in a call or by being returned), the object ID is added to the table, alongside the interface types it is known to implement. So, when a `catch` block is defined for an unprotected interface type, the type information of unprotected objects is known and, in case an unprotected object is thrown, it can be caught.

Figure 5 presents a graphical overview of how exceptions are handled normally (on the left) and in the presented compilation scheme (on the right). Lowercase letters indicate the allocation record for the corresponding function. A subscript *s* indicates a secure function; the stack grows downward. The order in which exception handlers are searched is indicated on arrows. The throw and throwback entry point split the same arrow in two parts.

The introduction of two additional entry points may seem to introduce functionality at the target level that the source-level lacks; however this is not the case. Only exceptions of existing types can be thrown and handling exceptions follows the normal

course of the stack. The external code could replace an exception, but this is equivalent to the high-level language functionality to `catch`ing an exception and `throw`ing another one. Thus, the target level is not granted additional functionality.

3.3.1. Limitations of the Compilation Scheme. In the context of secure compilation, if exception handlers are compiled as described earlier, one problem arises.

PROBLEM 7 (EXCESSIVE EXCEPTION CATCHING). *Consider two methods* `safeCallback` *that invoke method* `callback` *on an input parameter. Even though* `callback` *does not specify that it will throw exceptions, one implementation of* `safeCallback` *wraps it in a* `try/catch` *block.*

```
1  package p;
2  class C_L {
3    public safeCallback( e : External ) :
         Int {
4      try{
5        e.callback();
6      }catch ( v : Throwable ){
7        return 1;
8      }
9      return 0;
10   }
11 }
12 object o_L : C_L
```

```
1  package p;
2  class C_R {
3    public safeCallback( e : External ) :
         Int {
4
5      e.callback();
6
7
8
9      return 0;
10   }
11 object o_R : C_R
```

In $\mathcal{J}$+E, calling `safeCallback` *on either o_L or o_R always returns 0. However, when they are compiled, a low-level attacker could throw an exception during the execution of* `callback`*. This will cause differentiation between the two implementations, as* `safeCallback` *called on o_L returns 1, while called on o_R it returns 0.*

3.3.2. Secure Compilation of Exceptions. To counter the vulnerability described earlier, the compilation scheme is enhanced as follows.

Excessive Exceptions Catching. To address Problem 7, the code responsible for compiling callbacks needs to be augmented. Information on the possible exceptions thrown by callbacks are known at compile time, they appear in the method signature in the DIP. When compiling a callback to method $\mathtt{m}(\bar{x})$ `throws` u, type u must be saved on the secure stack. Code at the throwback entry point must then check that an exception thrown by unprotected code is one that could be thrown according to the corresponding unprotected method signature. This is done by performing a dynamic typecheck on the type of the thrown exception. As for method parameters, the dynamic typecheck is performed only on exceptions that are allocated in protected memory. If the exception to be caught comes from unprotected memory, no check is made, yet the information that a possible exception can be thrown needs to be recorded. If no exceptions can be thrown, a special value 0 is recorded. If the typecheck succeeds, then the exception is treated normally, otherwise all registers and flags are cleared and the execution `halt`s.

Since exceptions can be securely compiled, the compilation scheme is complete, as it addresses the totality of $\mathcal{J}$+E. This article then proceeds to the formalisation of the languages before proving this compilation scheme to be fully abstract.

4. LANGUAGE FORMALISATION

This section presents the formalisation of both $\mathcal{A}$+I (Section 4.1) and $\mathcal{J}$+E (Section 4.2).

4.1. Formalisation of the Target Language $\mathcal{A}$+I

This section presents syntax, semantics and trace semantics of $\mathcal{A}$+I. The formalisation is derived from the work of Patrignani and Clarke [2014].

Words	$w ::= [0 \text{ or } 1]^\ell$	*Memories*	$m ::= \emptyset$
Instructions	$i \in \mathcal{I} \subset \mathit{Words}$		$\mid m; a \mapsto w$
Empty word	$\mathbf{0} ::= 0^\ell$	*Numbers*	$n ::= n \in \mathbb{N}$
Addresses	$a \in 0..2^\ell - 1$	*Programs*	$P ::= (m, s)$
Memory descriptors	$s ::= (a_b, n_c, n_d, n)$		

Fig. 6. Elements of the formalisation of $\mathcal{A}$+I.

4.1.1. Syntax. As previously mentioned, $\mathcal{A}$+I is run on an architecture that models a Von Neumann machine consisting of a program counter p, a register file r, a flags register f and memory space m. The program counter indicates the address of the instruction that is executed. The register file contains 12 general-purpose registers $\mathtt{r}_0$ to $\mathtt{r}_{11}$ and a stack pointer register SP, which contains the address of the top of the current call stack. The flags register contains a zero flag ZF and a sign flag SF, which are set or cleared by arithmetic instructions and are used by branching instructions.

Figure 6 presents elements of the formalisation of $\mathcal{A}$+I. Words w are either instructions i or sequences of bits of length ℓ. Instructions i are elements of the set $\mathcal{I}$ and define the programming language executed on the architecture (Figure 1 in Section 2.2). The empty word $\mathbf{0}$ is a sequence of 0's whose length is based on the architecture being considered. Addresses a are natural numbers, ranging from 0 to $2^\ell - 1$. Memories m are maps from addresses to words. Memory access, denoted as $m(a)$, is defined as follows: $m(a) = w$ if $a \mapsto w \in m$; it is undefined otherwise. Define the domain of a memory as $\mathtt{dom}(m) = \{a \mid a \mapsto w \in m\}$. If two memories m and m' have disjoint domains, they can be merged to yield another memory. Formally, if $\mathtt{dom}(m) \cap \mathtt{dom}(m') = \emptyset$, then $m + m' = \{a \mapsto w \mid a \mapsto w \in m \text{ or } a \mapsto w \in m'\}$. Memory descriptors s are quadruples: (a_b, n_c, n_d, n): a_b is the address where the protected memory partition starts, n_c and n_d are the sizes (in number of addresses) of the code and data section, respectively, and n is the number of entry points. Entry points are allocated starting from the base address a_b. Each entry point is $\mathcal{N}_e$ words long (in the secure compilation scheme, $\mathcal{N}_e$ is 128). Assume the entry points do not overflow the protected code section; thus, the constraint $n \cdot \mathcal{N}_e < n_c$ holds for the all memory descriptors. Programs P are pairs of a memory m and a memory descriptor s.

4.1.2. Semantics. Before introducing the semantics, Figure 7 defines the memory access control enforcement rules. Read judgments $s \vdash \mathtt{predicate}(a, b, \ldots)$ as: "according to s, `predicate` holds for addresses $a, b, \ldots$". Whenever an access control rule is violated by a program, the execution is halted and the program counter is set to –1.

Define functions $\mathtt{m}_{\mathtt{sec}}(m, s)$ and $\mathtt{m}_{\mathtt{ext}}(m, s)$, which return the protected and unprotected parts of a memory m according to the descriptor s, respectively, as: $\mathtt{m}_{\mathtt{sec}}(m, s) = \{a \mapsto w \mid a \mapsto w \in m, s \vdash \mathtt{protected}(a)\}$ and $\mathtt{m}_{\mathtt{ext}}(m, s) = \{a \mapsto w \mid a \mapsto w \in m, s \vdash \mathtt{unprotected}(a)\}$.

The operational semantics is a small-step semantics that describes how each instruction of the language transforms an execution state into a new one. Thus, the operational semantics handles programs in the whole memory: both the protected and unprotected partitions.

Definition 4.1 (*Execution state*). An execution state, denoted as Ω, is a quintuple $\Omega = (p, r, f, m, s)$, where p is a program counter, r is a register file, f is a flags register, m is a memory, and s is a memory descriptor.

Given $\Omega = (p, r, f, m, s)$, let $\lfloor\Omega\rfloor$ be the state which encompasses only the protected memory: $(p, r, f, \mathtt{m}_{\mathtt{sec}}(m, s), s)$. Analogously, let $\lceil\Omega\rceil$ be the state that encompasses

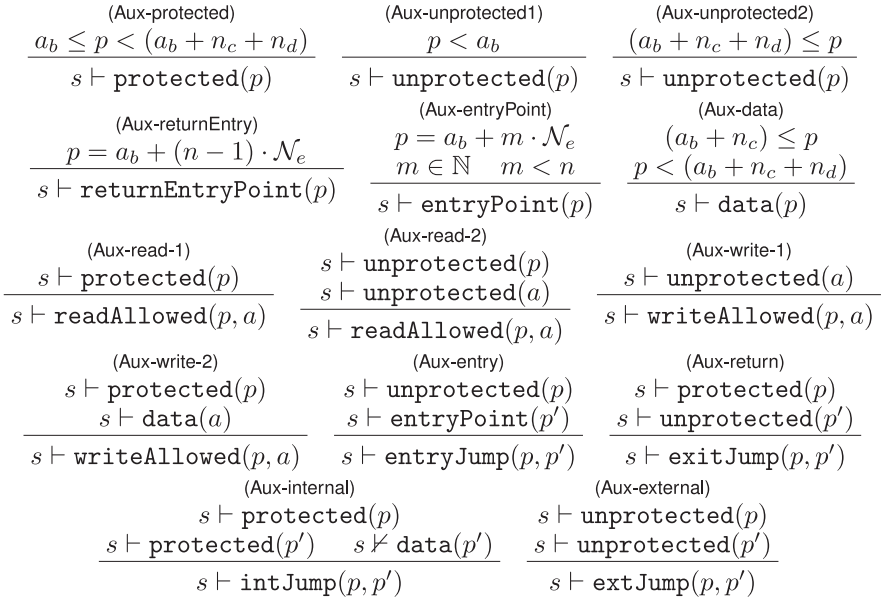

Fig. 7. Access control enforcement rules. Assume $s \equiv (a_b, n_c, n_d, n)$.

only the unprotected memory: $(p, r, f, \mathtt{m_{ext}}(m, s), s)$. Relations $\xrightarrow{i} \subseteq \lfloor\Omega\rfloor \times \lfloor\Omega\rfloor$ and $\xrightarrow{e} \subseteq \lceil\Omega\rceil \times \lceil\Omega\rceil$ describe the evaluation of instructions that only affect the protected and unprotected parts of memory, respectively. Figure 8 presents the rules for $\xrightarrow{i}$, rules for $\xrightarrow{e}$ are obtained by replacing an `intJump` assumption with an `extJump` one. In the rules, notation $m[a \mapsto w]$ indicates that memory m is updated to a new one that is equal to m except that the value stored at address a is w. Notation $r[\mathsf{R} \mapsto w]$ indicates that the register file r is updated to a new one that is equal to r except that the value stored in register R is w. Notation $r(\mathsf{R})$ indicates the value contained in register R in register file r. Let $m(p) = \mathtt{inst}$ denote that `inst` is the word allocated in $m(p)$, where $\mathtt{inst} \in \mathcal{I}$. Note that the program counter is set to -1 whenever the `halt` instruction is encountered, in order to capture termination. This way, no progress can be made, as $m(-1)$ does not return a valid instruction: the program is in a stuck state.

Definition 4.2 (*Stuck state*). A state $\Omega = (p, r, f, m, s)$ is stuck, denoted as $\Omega^{\perp}$, when the program counter does not point to a valid instruction: $m(p) \notin \mathcal{I}$.

The operational semantics of $\mathcal{A}$+I is a binary relation over states $\twoheadrightarrow \subseteq \Omega \times \Omega$ defined by the rules of Figure 9. It relies on relations $\xrightarrow{i}$ and $\xrightarrow{e}$, for transitions that affect only a section of memory, as captured by rules Eval-protected and Eval-unprotected. The compiler ensures that rules Eval-movl-out and Eval-movs-out will never be executed and that unused registers and flags are always reset to **0** when jumping between protected and unprotected memory sections. Rule Eval-callback (and return) is performed when executing a callback (and a return). The only difference between the callback and the return cases is that in the latter one, all registers besides $\mathtt{r_0}$ contain 0.

The transitive closure of relation $\twoheadrightarrow$ is indicated with $\twoheadrightarrow^*$. A state Ω performing n reduction steps is indicated as $\Omega \twoheadrightarrow^n \Omega'$. The evaluation of program P is a sequence of steps that takes the initial state of P to another state.

Definition 4.3 (*Initial state*). The initial state of a program (m, s), denoted as $\Omega_0(m, s)$, is the state (p_0, r_0, f_0, m, s), where $s = (a_b, n_c, n_d, n)$, $p_0 = (a_b + n_c + n_d + 2)$, $r_0 = [\mathsf{SP} \mapsto 0; \mathtt{r_i} \mapsto 0_{\mathtt{i}=0..11}]$, and $f_0 = [\mathsf{ZF} \mapsto 0; \mathsf{SF} \mapsto 0]$.

$$\text{(Eval-movl)}\quad \frac{m(p) = (\texttt{movl}\ \texttt{r}_\texttt{d}\ \texttt{r}_\texttt{s}) \quad s \vdash \texttt{intJump}(p, p+1) \quad s \vdash \texttt{readAllowed}(p, r(\texttt{r}_\texttt{s})) \quad r' = r[\texttt{r}_\texttt{d} \mapsto m(r(\texttt{r}_\texttt{s}))]}{(p, r, f, m, s) \xrightarrow{i} (p+1, r', f, m, s)}$$

$$\text{(Eval-movs)}\quad \frac{m(p) = (\texttt{movs}\ \texttt{r}_\texttt{d}\ \texttt{r}_\texttt{s}) \quad s \vdash \texttt{intJump}(p, p+1) \quad s \vdash \texttt{writeAllowed}(p, r(\texttt{r}_\texttt{d})) \quad m' = m[r(\texttt{r}_\texttt{d}) \mapsto r(\texttt{r}_\texttt{s})]}{(p, r, f, m, s) \xrightarrow{i} (p+1, r, f, m', s)}$$

$$\text{(Eval-movi)}\quad \frac{m(p) = (\texttt{movi}\ \texttt{r}_\texttt{d}\ \texttt{i}) \quad s \vdash \texttt{intJump}(p, p+1) \quad r' = r[\texttt{r}_\texttt{d} \mapsto i]}{(p, r, f, m, s) \xrightarrow{i} (p+1, r', f, m, s)}$$

$$\text{(Eval-compare)}\quad \frac{m(p) = (\texttt{cmp}\ \texttt{r}_1\ \texttt{r}_2) \quad s \vdash \texttt{intJump}(p, p+1) \quad f' = f[\mathsf{ZF} \mapsto (r(\texttt{r}_1) == r(\texttt{r}_2)); \ \mathsf{SF} \mapsto (r(\texttt{r}_1) < r(\texttt{r}_2))]}{(p, r, f, m, s) \xrightarrow{i} (p+1, r, f', m, s)}$$

$$\text{(Eval-add)}\quad \frac{m(p) = (\texttt{add}\ \texttt{r}_\texttt{d}\ \texttt{r}_\texttt{s}) \quad s \vdash \texttt{intJump}(p, p+1) \quad v = (r(\texttt{r}_\texttt{d}) + r(\texttt{r}_\texttt{s}))\%2^\ell \quad r' = r[\texttt{r}_\texttt{d} \mapsto v] \quad f' = f[\mathsf{ZF} \mapsto (v == 0)]}{(p, r, f, m, s) \xrightarrow{i} (p+1, r', f', m, s)}$$

$$\text{(Eval-sub)}\quad \frac{m(p) = (\texttt{sub}\ \texttt{r}_\texttt{d}\ \texttt{r}_\texttt{s}) \quad s \vdash \texttt{intJump}(p, p+1) \quad v = (r(\texttt{r}_\texttt{d}) - r(\texttt{r}_\texttt{s}))\%2^\ell \quad r' = r[\texttt{r}_\texttt{d} \mapsto v] \quad f' = f[\mathsf{ZF} \mapsto (v == 0); \ \mathsf{SF} \mapsto (r(\texttt{r}_\texttt{d}) - r(\texttt{r}_\texttt{s}) < 0)]}{(p, r, f, m, s) \xrightarrow{i} (p+1, r', f', m, s)}$$

$$\text{(Eval-function-call)}\quad \frac{m(p) = (\texttt{call}\ \texttt{r}_\texttt{d}) \quad p' = m(r(\texttt{r}_\texttt{d})) \quad s \vdash \texttt{intJump}(p, p') \quad r' = r[\mathsf{SP} \mapsto r(\mathsf{SP}) + 1] \quad m' = m[r''(\mathsf{SP}) \mapsto p + 1]}{(p, r, f, m, s) \xrightarrow{i} (p', r', f, m', s)}$$

$$\text{(Eval-function-ret)}\quad \frac{m(p) = (\texttt{ret}) \quad p' = m(r(\mathsf{SP})) \quad s \vdash \texttt{intJump}(p, p') \quad r' = r[\mathsf{SP} \mapsto r(\mathsf{SP}) - 1]}{(p, r, f, m, s) \xrightarrow{i} (p', r', f, m, s)}$$

$$\text{(Eval-je-true)}\quad \frac{m(p) = (\texttt{je}\ \texttt{r}_\texttt{i}) \quad f(\mathsf{ZF}) == 1 \quad p' = r(\texttt{r}_\texttt{i}) \quad s \vdash \texttt{intJump}(p, p')}{(p, r, f, m, s) \xrightarrow{i} (p', r, f, m, s)}$$

$$\text{(Eval-jl-true)}\quad \frac{m(p) = (\texttt{jl}\ \texttt{r}_\texttt{i}) \quad f(\mathsf{SF}) == 1 \quad p' = r(\texttt{r}_\texttt{i}) \quad s \vdash \texttt{intJump}(p, p')}{(p, r, f, m, s) \xrightarrow{i} (p', r, f, m, s)}$$

$$\text{(Eval-je-false)}\quad \frac{m(p) = (\texttt{je}\ \texttt{r}_\texttt{i}) \quad f(\mathsf{ZF}) == 0 \quad s \vdash \texttt{intJump}(p, p+1)}{(p, r, f, m, s) \xrightarrow{i} (p+1, r, f, m, s)}$$

$$\text{(Eval-jl-false)}\quad \frac{m(p) = (\texttt{jl}\ \texttt{r}_\texttt{i}) \quad f(\mathsf{SF}) == 0 \quad s \vdash \texttt{intJump}(p, p+1)}{(p, r, f, m, s) \xrightarrow{i} (p+1, r, f, m, s)}$$

$$\text{(Eval-jump)}\quad \frac{m(p) = (\texttt{jmp}\ \texttt{r}_\texttt{d}) \quad p' = r(\texttt{r}_\texttt{d}) \quad s \vdash \texttt{intJump}(p, p')}{(p, r, f, m, s) \xrightarrow{i} (p', r, f, m, s)}$$

$$\text{(Eval-halt)}\quad \frac{m(p) = (\texttt{halt})}{(p, r, f, m, s) \xrightarrow{i} (-1, r, f, m, s)}$$

Fig. 8. Operational semantics of $\mathcal{A}$+I instructions for programs in the protected memory partition.

The evaluation of P terminates if $\Omega_0(P) \rightarrow\!\!\!\!\rightarrow^* \Omega^\perp$; the result of the computation is stored in $\texttt{r}_0$. If the evaluation of program P does not terminate, P diverges. A program P diverges, denoted as $P \Uparrow$, if it executes an unbounded number of reduction steps. Formally: $P \Uparrow$ if $\forall n \in \mathbb{N}, \exists \Omega'. \Omega_0(P) \rightarrow\!\!\!\!\rightarrow^n \Omega'$.

Fully abstract compilation relies on the notion of contextual equivalence, which is now defined. Contextual equivalence relates two programs that cannot be distinguished by any third program interacting with them [Plotkin 1977]. This notion relies on the concept of contexts, which is introduced before presenting the equivalence itself.

Since we consider $\mathcal{A}$+I programs P that are placed in protected memory and interact with arbitrary unprotected code, contexts model that unprotected code. Thus, for any descriptor s, contexts $\mathbb{M}$ are partial memories: $\mathbb{M} = m$, where all addresses of $\mathbb{M}$ are

$$\frac{\lfloor \Omega \rfloor \xrightarrow{i} \lfloor \Omega' \rfloor}{\Omega \twoheadrightarrow \Omega'}\ (\text{Eval-protected}) \qquad \frac{\lceil \Omega \rceil \xrightarrow{e} \lceil \Omega' \rceil}{\Omega \twoheadrightarrow \Omega'}\ (\text{Eval-unprotected})$$

$$\frac{\begin{array}{c} m(p) = (\texttt{movl}\ \texttt{r}_\texttt{d}\ \texttt{r}_\texttt{s}) \\ s \vdash \texttt{intJump}(p, p+1) \\ s \vdash \texttt{readAllowed}(p, r(\texttt{r}_\texttt{s})) \\ s \vdash \texttt{unprotected}(r(\texttt{r}_\texttt{s})) \\ r' = r[\texttt{r}_\texttt{d} \mapsto m(r(\texttt{r}_\texttt{s}))] \end{array}}{(p, r, f, m, s) \twoheadrightarrow (p+1, r', f, m, s)}\ (\text{Eval-movl-out})$$

$$\frac{\begin{array}{c} m(p) = (\texttt{movs}\ \texttt{r}_\texttt{d}\ \texttt{r}_\texttt{s}) \\ s \vdash \texttt{intJump}(p, p+1) \\ s \vdash \texttt{writeAllowed}(p, r(\texttt{r}_\texttt{d})) \\ s \vdash \texttt{unprotected}(r(\texttt{r}_\texttt{d})) \\ m' = m[r(\texttt{r}_\texttt{d}) \mapsto r(\texttt{r}_\texttt{s})] \end{array}}{(p, r, f, m, s) \twoheadrightarrow (p+1, r, f, m', s)}\ (\text{Eval-movs-out})$$

$$\frac{\begin{array}{c} m(p) = (\texttt{jmp}\ \texttt{r}_\texttt{d}) \\ p' = r(\texttt{r}_\texttt{d}) \qquad s \vdash \texttt{exitJump}(p, p') \end{array}}{(p, r, f, m, s) \twoheadrightarrow (p', r, f, m, s)}\ (\text{Eval-callback (and return)})$$

$$\frac{\begin{array}{c} m(p) = (\texttt{jmp}\ \texttt{r}_\texttt{d}) \\ p' = r(\texttt{r}_\texttt{d}) \qquad s \vdash \texttt{entryJump}(p, p') \end{array}}{(p, r, f, m, s) \twoheadrightarrow (p', r, f, m, s)}\ (\text{Eval-call (and returnback)})$$

Fig. 9. Operational semantics of whole $\mathcal{A}$+I programs.

unprotected. Formally, given s, $\forall a \in \texttt{dom}(\mathbb{M}), s \vdash \texttt{unprotected}(a)$. A program P and a context $\mathbb{M}$ are compatible, denoted as $P \frown \mathbb{M}$, if the memories of P and $\mathbb{M}$ have disjoint domains. Let $\texttt{dom}(\mathbb{M}) = \texttt{dom}(m)$ if $\mathbb{M} = m$; formally, $P \frown \mathbb{M}$ if $P = (m', s)$ and $\texttt{dom}(m') \cap \texttt{dom}(\mathbb{M}) = \emptyset$. If P and $\mathbb{M}$ are compatible, $\mathbb{M}$ can be plugged with P in order to model interaction between P and $\mathbb{M}$. Formally, if $P \frown \mathbb{M}$, then $\mathbb{M}[(m', s)] = (m' + m, s)$.

Programs P_1 and P_2 are contextually equivalent, denoted as $P_1 \simeq P_2$, when, for *all* contexts they interact with, P_1 diverges if and only if P_2 also diverges.

Definition 4.4 (*Contextual equivalence*). $P_1 \simeq P_2$ if $\forall \mathbb{M}.\ P_1 \frown \mathbb{M} \wedge \mathbb{M}[P_1]\Uparrow \iff P_2 \frown \mathbb{M} \wedge \mathbb{M}[P_2]\Uparrow$.

An implication of this definition is that for P_1 and P_2 to be contextually equivalent, they must have the same memory descriptor. For the sake of simplicity we will always assume the compatibility of a program and the context it is plugged in, shortening the aforementioned definition to: $P_1 \simeq P_2$ if $\forall \mathbb{M}.\ \mathbb{M}[P_1]\Uparrow \iff \mathbb{M}[P_2]\Uparrow$.

4.1.3. Trace Semantics. As for the operational semantics, a notion of execution states is required for the trace semantics as well. Execution states, denoted as Θ, are the same as Ω except that Θ do not deal with the whole memory but just with its protected partition. So, the memory m of (p, r, f, m, s) spans only the protected memory partition indicated by s. Additionally, Θ can be $(\mathsf{unknown}, m, s)$, an unknown state that models when code is executing in unprotected memory [Jeffrey and Rathke 2005b].

Definition 4.5 (*Initial state for traces*). The initial state for traces of a program (m, s), denoted as $\Theta_0(m, s)$, is the state $(\mathsf{unknown}, m, s)$.

The following are the labels exhibited by the traces semantics.

$$\Lambda ::= \alpha \mid \tau_i \qquad \alpha ::= \surd \mid \gamma? \mid \gamma! \qquad \gamma ::= \texttt{call}\ p(r) \mid \texttt{ret}\ p\ \texttt{r}_0$$

A label Λ can be either an observable action α or a nonobservable action τ_i. Action τ_i indicates the unobservable action occurred in protected memory. Decorations ? and ! indicate the direction of the observable action: from unprotected to protected code (?) or vice-versa (!). Observable actions include a tick $\surd$ indicating that the execution has terminated. Additionally, observable actions are function calls or returns to a certain address p, combined with the registers r and flags f. Registers and flags are in the labels as they convey information on the behaviour of programs. There are no labels

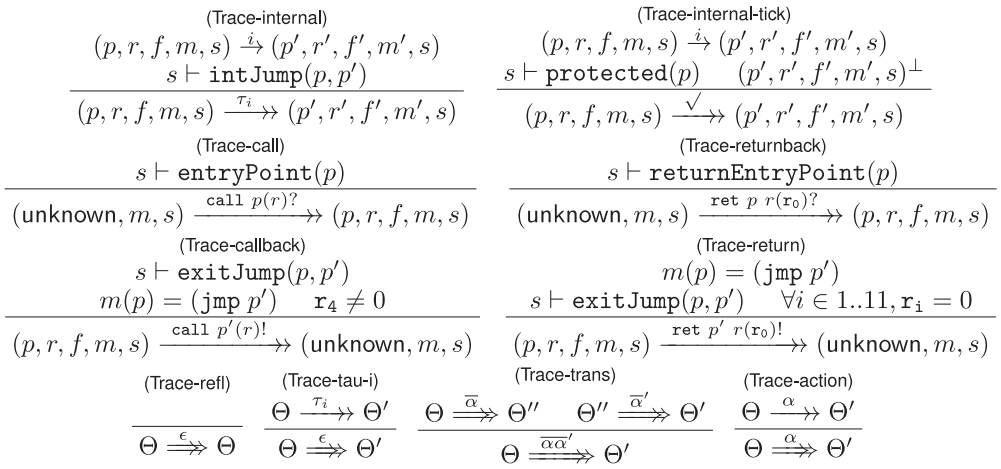

Fig. 10. Rules of the trace semantics of $\mathcal{A}$+I.

for reads or writes to unprotected memory as those instructions are never generated by the compiler.

Figure 10 presents the rules defining the relation $\Theta \overset{\overline{\alpha}}{\Longrightarrow} \Theta'$, which describe when a state Θ generates trace $\overline{\alpha}$ and results in state Θ'. The traces of a $\mathcal{A}$+I program P is defined as follows: $\mathsf{Traces}(P) = \{\overline{\alpha} \mid \exists \Theta'.\Theta_0(P) \overset{\overline{\alpha}}{\Longrightarrow} \Theta'\}$. Two programs P_1 and P_2 are trace-equivalent, denoted as $P_1 \simeq_{\mathrm{T}} P_2$, if their traces are the same.

Definition 4.6 (*Trace equivalence*). $P_1 \simeq_{\mathrm{T}} P_2$ if $\mathsf{Traces}(P_1) = \mathsf{Traces}(P_2)$.

An important result of this trace semantics is that when it is applied to compiled components, traces capture all possible behaviours of a compiled component. Thus the trace semantics captures precisely the same notion as the operational semantics. The formal statement of this property is captured by Proposition 4.7.

Proposition 4.7 (Trace semantics is equivalent to operational semantics). *For any two $\mathcal{A}$+I components $C_1^{\downarrow}$ and $C_2^{\downarrow}$ obtained from compiling $\mathcal{J}$+E components C_1 and C_2 with the compilation scheme of Section 3, we have that: $C_1^{\downarrow} \simeq_{\mathrm{T}} C_2^{\downarrow} \iff C_1^{\downarrow} \simeq C_2^{\downarrow}$.*

Proposition 4.7, whose proof is detailed in [Patrignani and Clarke 2014], drives the proof strategy for the main result, which is presented in Section 5. That work considers an assembly language enhanced with a protected modules architecture, yet that formalisation differs slightly from the one presented in this article. In that work, the language has two stacks and switching between them is done by using `call` and `ret` instructions. In this work, only one stack is present and `jmp` instructions are used to switch between memory partitions. Intuitively, that formalisation has a built-in notion of secure and insecure stacks, while this work only has an insecure one and the secure stack is set up by the secure compiler. The results of Proposition 4.7 can thus be applied in this article as well.

4.2. Formalisation of $\mathcal{J}$+E

This section presents the formalisation of the dynamic semantics of $\mathcal{J}$+E, which borrows extensively from that of Java Jr. [Jeffrey and Rathke 2005b]. Its syntax was already presented in Figure 2 in Section 2.3.

4.2.1. Dynamic Semantics. The dynamic semantics is given in terms of a relation $(C; S \vdash E) \rightarrow (C'; S' \vdash E')$ that models the evolution of component C executing expression E with stack S to C' executing E' with stack S'. A binding B is a list of associations from variables to values, $B ::= \emptyset \mid B; (x \mapsto v)$. The lookup of the value associated to a variable, denoted as $B(x)$, returns v if $(x \mapsto v) \in B$ and is undefined otherwise. A stack S is a list of bindings $S ::= \overline{B}$, lookup and addition are always made to the top of the stack, so if $S = B_1, \ldots, B_n$, then $S(x)$ stands for $B_1(x)$ and $S, (x \mapsto v)$ stands for $B_1, (x \mapsto v)$. The expression being executed is immersed in an evaluation context $\mathbb{E}$, which models the environment in which the evaluation takes place. The syntax of evaluation contexts is:

$$\begin{aligned}\mathbb{E} ::= {} & [\cdot] \mid \mathbb{E}.m(\overline{E}) \mid p.o.m(\overline{v}, \mathbb{E}, \overline{E}) \mid \mathbb{E}.f \mid \mathbb{E}.f = E \mid v.f = \mathbb{E} \mid \texttt{new}\ t(\overline{v}, \mathbb{E}, \overline{E}) \\ & \mid \texttt{if}(\mathbb{E})\{E_T\}\texttt{else}\{E_F\} \mid \mathbb{E}; E \mid \mathbb{E}\ \texttt{in}\ p \mid \texttt{var}\ x : t = \mathbb{E} \mid \texttt{return}\ \mathbb{E} \\ & \mid \mathbb{E}\ \texttt{op}\ E \mid v\ \texttt{op}\ \mathbb{E} \mid \texttt{try}\ \mathbb{E}\ \texttt{catch}(x : t)\ E \mid \texttt{throw}\ \mathbb{E} \mid \texttt{exit}\ \mathbb{E}\end{aligned}$$

Rules for reductions of the form $(C; S \vdash E) \rightarrow (C'; S' \vdash E')$ are presented in Figures 11 and 12. Contextual equivalence for $\mathcal{J}$+E programs is defined based on $\mathcal{J}$+E contexts $\mathbb{C}$, which are components with a hole, denoted with $\mathbb{C}[\cdot]$ [Jeffrey and Rathke 2005b]. The hole can be filled with another component C to denote the interaction between C and the context. Assume the context defines a `Main` package with a `Main` class and a `main` method that identify where the execution starts. We overload the $\simeq$ notation and define contextual equivalence for $\mathcal{J}$+E programs as follows: $C_1 \simeq C_2$ if $\forall \mathbb{C}.\, \mathbb{C}[C_1]\Uparrow \iff \mathbb{C}[C_2]\Uparrow$ knowing that this relation will not be mistaken for the $\mathcal{A}$+I one.

5. FULL ABSTRACTION OF THE COMPILATION OF $\mathcal{J}$+E TO $\mathcal{A}$+I

This section firstly presents the algorithm mentioned in Section 2.4 through a series of examples (Section 5.1). Then it presents the proof of full abstraction of the compilation scheme of Section 3, which relies on the algorithm (Section 5.2).

5.1. The Algorithm

This section presents the algorithm which takes as input two different, low-level traces $\overline{\alpha}_1$ and $\overline{\alpha}_2$ and two components C_1 and C_2 and outputs a high-level component C that differentiates C_1 and C_2. Traces $\overline{\alpha}_1$ and $\overline{\alpha}_2$ were generated by $C_1^{\downarrow}$ and $C_2^{\downarrow}$ when interacting with the same, unknown external memory. This section presents several examples of the expected output of the algorithm when different traces and components are input. The examples illustrate crucial cases the algorithm needs to consider when creating the output component.

In the following, the adjective *internal* denotes objects (classes) that are allocated (defined) by components C_1 and C_2. The adjective *external* denotes objects (classes) that are allocated (defined) by the output component.

General idea. The algorithm analyses actions in the low-level traces $\overline{\alpha}_1$ and $\overline{\alpha}_2$. Those actions can be call, return, callback, returnback, and termination ($\surd$). Actions that appear at even-numbered positions in a trace are calls or returnbacks, generated from the external memory. Actions that appear at odd-numbered positions are returns or callbacks, generated by C_1 or C_2. This partitioning is because execution starts in unprotected memory.

For the algorithm to be correct, it must detect when two different actions are encountered at an odd position in a trace. Assuming the first different actions are at index i, the algorithm produces code that replicates the first $i - 1$ actions. Then, it produces code that, based on the difference in the i-th action, either diverges or terminates based on which component it is interacting with.

$$\frac{C.v = \{\textbf{package}\ p; \textbf{object}\ v : t\ \textbf{implements}\ \overline{t}\ \{\overline{F}\}\} \quad \textbf{public}\ m(\overline{x} : \overline{t}) : t'\{\textbf{return}\ E;\} \in C.t.\texttt{mths}}{(C; S \vdash \mathbb{E}[v.m(\overline{v})]) \to (C; \emptyset, S \vdash \mathbb{E}[\textbf{return}\ E[v/\texttt{this}, \overline{v}/\overline{x}]\ \texttt{in}\ p])}\ \text{(Eval-method)}$$

$$\frac{}{(C; B, S \vdash \mathbb{E}[\textbf{return}\ v]) \to (C; S \vdash \mathbb{E}[v])}\ \text{(Eval-return)}$$

$$\frac{C.v = \{\textbf{package}\ p; \textbf{object}\ v : t\ \textbf{implements}\ \overline{t}\ \{\overline{F}\}\} \qquad f = u \in \overline{F}}{(C; S \vdash \mathbb{E}[v.f]) \to (C; S \vdash \mathbb{E}[u])}\ \text{(Eval-field)}$$

$$\frac{C.v = \{\textbf{package}\ p; \textbf{object}\ v : t\ \textbf{implements}\ \overline{t}\ \{\overline{F}\}\} \qquad (f = u;) \in \overline{F} \qquad C' = C + \{\textbf{package}\ p; \textbf{object}\ v : t\ \textbf{implements}\ \overline{t}\ \{\overline{F}'\}\} \qquad \overline{F}' = \overline{F} + (f = u)}{(C; S \vdash \mathbb{E}[v.f = u]) \to (C'; S \vdash \mathbb{E}[u])}\ \text{(Eval-field-update)}$$

$$\frac{C.p.c.\texttt{flds} = \overline{f} : \overline{t} \qquad p.o \notin \texttt{dom}(C) \qquad C' = C + \{\textbf{package}\ p; \textbf{object}\ o : p.c\ \textbf{implements}\ \epsilon\{\overline{f} = \overline{v}\}\}}{(C; S \vdash \mathbb{E}[\textbf{new}\ p.c(\overline{v})]) \to (C'; S \vdash \mathbb{E}[p.o])}\ \text{(Eval-new)}$$

$$\frac{v = \texttt{true}}{(C; S \vdash \mathbb{E}[\textbf{if}(v)\{E_T\}\textbf{else}\{E_F\}]) \to (C; S \vdash \mathbb{E}[E_T])}\ \text{(Eval-if-true)}$$

$$\frac{v = \texttt{false}}{(C; S \vdash \mathbb{E}[\textbf{if}(v)\{E_T\}\textbf{else}\{E_F\}]) \to (C; S \vdash \mathbb{E}[E_F])}\ \text{(Eval-if-false)}$$

$$\frac{}{(C; S \vdash \mathbb{E}[v\ \texttt{in}\ p]) \to (C; S \vdash \mathbb{E}[v])}\ \text{(Eval-coercion)}$$

$$\frac{}{(C; S \vdash \mathbb{E}[\textbf{var}\ x : t = v]) \to (C; S, (x \mapsto v) \vdash \mathbb{E}[\texttt{unit}])}\ \text{(Eval-local-var)}$$

$$\frac{S(x) = v}{(C; S \vdash \mathbb{E}[x]) \to (C; S \vdash \mathbb{E}[v])}\ \text{(Eval-lookup)}$$

$$\frac{}{(C; S \vdash \mathbb{E}[v; E]) \to (C; S \vdash \mathbb{E}[E])}\ \text{(Eval-concatenation)}$$

$$\frac{v\ \texttt{op}\ v' = v''}{(C; S \vdash \mathbb{E}[v\ \texttt{op}\ v']) \to (C; S \vdash \mathbb{E}[v''])}\ \text{(Eval-op)}$$

$$\frac{}{(C; S \vdash \mathbb{E}[\textbf{try}\{v\}\textbf{catch}(x : t)\{E\}]) \to (C; S \vdash \mathbb{E}[v])}\ \text{(Eval-try)}$$

$$\frac{v \neq \texttt{throw}\ v'}{(C; S \vdash \mathbb{E}[\textbf{try}\{v\}\textbf{catch}(x : t)\{E\}]) \to (C; S \vdash \mathbb{E}[v])}\ \text{(Eval-catch)}$$

$$\frac{C.v = \{\textbf{package}\ p; \textbf{object}\ v : t\ \textbf{implements}\ \overline{t}\{\overline{F}\}\} \qquad t <: t'}{(C; S \vdash \mathbb{E}[\textbf{try}\{\textbf{throw}\ v\}\textbf{catch}(x : t')\{E\}]) \to (C; S \vdash \mathbb{E}[E[v/x]])}\ \text{(Eval-catch-fail)}$$

$$\frac{C.v = \{\textbf{package}\ p; \textbf{object}\ v : t\ \textbf{implements}\ \overline{t}\{\overline{F}\}\} \qquad t \not<: t'}{(C; S \vdash \mathbb{E}[\textbf{try}\{\textbf{throw}\ v\}\textbf{catch}(x : t')\{E\}]) \to (C; S \vdash \mathbb{E}[\textbf{throw}\ v])}\ \text{(Eval-exit)}$$

$$\frac{}{(C; S \vdash \mathbb{E}[\textbf{exit}\ v]) \to (C; S \vdash v)}$$

Fig. 11. Dynamic semantics of $\mathcal{J}$+E. The subtyping relation is denoted by $<:$.

$$\frac{}{(C; S \vdash \mathbb{E}[\textbf{throw}\ v; E]) \to (C; S \vdash \mathbb{E}[\textbf{throw}\ v])}$$
(Eval-throw-sequence)

$$\frac{}{(C; S \vdash \mathbb{E}[\textbf{throw}\ \textbf{throw}\ v]) \to (C; S \vdash \mathbb{E}[\textbf{throw}\ v])}$$
(Eval-throw-throw)

$$\frac{}{(C; S \vdash \mathbb{E}[\textbf{throw}\ x]) \to (C; S \vdash \mathbb{E}[\textbf{throw}\ v])}$$
(Eval-throw-var)

$$\frac{}{(C; S \vdash \mathbb{E}[\textbf{var}\ x : t = \textbf{throw}\ v]) \to (C; S, (x \mapsto \textbf{throw}\ v) \vdash \mathbb{E}[\textbf{throw}\ v])}$$
(Eval-throw-new)

$$\frac{}{(C; S \vdash \mathbb{E}[\textbf{new}\ \textbf{throw}\ v]) \to (C; S \vdash \mathbb{E}[\textbf{throw}\ v])}$$
(Eval-throw-if)

$$\frac{}{(C; S \vdash \mathbb{E}[\textbf{if}(\textbf{throw}\ v)\{E_T\}\textbf{else}\{E_F\}]) \to (C; S \vdash \mathbb{E}[\textbf{throw}\ v])}$$

Fig. 12. Dynamic semantics of $\mathcal{J}$+E, rules for exception propagation.

The algorithm has been implemented in Scala, and it outputs Java components that adhere to the $\mathcal{J}$+E formalisation.[5] For implementation purposes, instead of diverging in a case and terminating in the other the implementation terminates with value 1 or 2. This formulation of contextual equivalence is equivalent to ours, yet more amenable to an implementation [Curien 2007].

Starting point. The algorithm starts by creating a knowledge base about C_1 and C_2. The knowledge base contains all signatures of internally and externally defined methods, as well as high- and low-level identities of static objects and externs. This is because the algorithm needs to be able to differentiate, for example, whether a type is internally or externally defined, or what are the identities of static objects. Then, a code skeleton for the output component is created, based on the structure of the distinguished import package (DIP) of C_1 and C_2.

For all interfaces i defined in the DIP, a class i_c is created. An object staticFor_i of type i_c is then created. Classes i_c contain dummy implementations of all methods defined in i and in all interfaces i extends. These method implementations return a value whose type matches the expected return type: 0 for type Int, unit for type Unit and null otherwise. A method called defaultCreate() is added to all classes i_c, it is implemented as follows:

```
1 public defaultCreate() : i_c {
2   if ( this ≠ staticFor_i)
3     return staticFor_i.defaultCreate();
4   return new i_c ();
5 }
```

Methods defaultCreate() are responsible for allocating external objects, they will be called only on objects staticFor_i. Constructors inside defaultCreate() are supplied standard values for their parameters: 0 for type Int, false for type Bool, unit for type Unit, and null otherwise. These classes have no fields, as they are never accessed by protected code. For the sake of simplicity, the following examples have constructors with no parameters.

[5]Available at http://people.cs.kuleuven.be/∼marco.patrignani/Publications.html.

The output component is extended with extra classes. First, class `Tester` containing the `main` method is added; it is required for the execution to start. Other needed classes will be introduced and motivated by the following examples.

Code examples. The following examples present different implementations of C_1, on the left, and of C_2, on the right. Components C_1 and C_2 are modifications of the code in Listing 1, whose DIP is defined in Listing 3. Omitted code is the same in both C_1 and C_2 and can be found in Listing 1. Each code fragment is followed by the low-level trace it generates: $\overline{\alpha_1}$ and $\overline{\alpha_2}$ for C_1 and C_2, respectively. The examples also describe what the algorithm must do in order to create the correct output before presenting the output produced for each case.

```
1 package PIMP;
2 interface Transaction extends Atomic {
3   public createTrans() : Transaction;
4   public callback( arg : Transaction ) : Unit;
5 }
6 interface Atomic {
7   public lock() : Int;
8 }
9 extern extTrans : Transaction;
```

Listing 3. Example of a distinguished import package.

The low-level traces will be massaged to aid understanding. For example, given that object `extAccount` is compiled to identity 0x123 and that the entry point of method `createAccount` is located at address 0x456, the low-level label `call` 0x456($r[\mathtt{r}_4 = 0\mathtt{x}123]$) is written as `extAccount.createAccount()`. This abstraction is safe, as it does not introduce additional information, it merely massages the present one into a more human-readable form. Numbers in italic font (e.g., *1*) refer to indexes from $\mathcal{O}$, whereas identities of externally allocated objects are numbers in hexadecimal form.

Example 5.1 (*Different returned values*). Consider the following implementations for C_1 and C_2.

```
1 object extAccount: AccountClass{
2   counter = 1
3 }
4 public getBalance(): Int{
5   return counter;
6 }
```

```
1 object extAccount: AccountClass{
2   counter = 0
3 }
4 public getBalance(): Int{
5   return counter += 1;
6 }
```

Trace $\overline{\alpha_1}$ for C_1 is `extAccount.getBalance()?` · `ret 1!` · `extAccount.getBalance()?` · `ret 1!`, while trace $\overline{\alpha_2}$ for C_2 is `extAccount.getBalance()?` · `ret 1!` · `extAccount.getBalance()?` · `ret 2!`. In this example, the produced code needs to differentiate between C_1 and C_2 based on the type of expected returned values. These types can be either primitive, internal, or external. With primitive-typed values, the differentiation is based on the different values returned by C_1 or C_2, in this case `1` and `2`, respectively.

This example highlights how both the algorithm and the produced code need to keep track of the index of the action they replicate. To that end, the algorithm maintains a global variable. The produced code is extended with a class `Helper` and a static object `oc` implementing it. `Helper` contains a field `step` with methods `getStep()` and `incrementStep()`, the latter increases the value of `step` by one. Additionally, it contains a method `diverge()` that recursively calls itself, which is used to achieve divergence. As `oc` is static, its fields are global variables for the output component.

The algorithm outputs the code of Listing 4. The first actions generate the code in lines 2 to 6; thus, it is wrapped in an if-statement that makes the generated code take

```
1  public main ( args : String[] ) : Unit {
2    if ( oc.getStep() == 0 ) {
3      oc.incrementStep();
4      var vara : Int = extAccount.getBalance();
5      oc.incrementStep();
6    }
7    if ( oc.getStep() == 2 ) {
8      oc.incrementStep();
9      var varb : Int = extAccount.getBalance();
10     if ( varb == 1 ) {
11       exit( 1 );
12     } else {
13       oc.diverge();
14     }
15   }
16 }
```

Listing 4. Output of the algorithm for Example 5.1.

place only when the considered action is the first (i.e., `step` is 0). The second actions are responsible for incrementing `step` in line 5. The third actions generate the code in lines 7 to 11, while the fourth actions, the different ones, generate the code in line 10.

The approach of this example is similar to what the algorithm does in case the difference in the traces is in primitive-typed parameters of a callback. In that case, instead of creating fresh variable `varb`, the produced code performs the differentiation by using the name of the parameter, which has the different value.

Example 5.2 (*Different internally typed returned object*).

```
1 public createAccount() : Account {
2   return this;
3 }
```

```
1 public createAccount() : Account {
2   return new AccountClass();
3 }
```

Trace $\overline{\alpha_1}$ is `extAccount.createAccount()?` · `ret extAccount!`, while trace $\overline{\alpha_2}$ is `extAccount.createAccount()?` · `ret` *1*!. In this case, the produced code must be able to differentiate between two return values that are internal objects. They are given different indexes in $\mathcal{O}$. Here, C_1 returns a known object: `extAccount`, while C_2 returns a new object: index *1* in $\mathcal{O}$.

To achieve differentiation in this case, the produced code needs to keep track of internally allocated objects. For this, it relies on a list `internals` provided by `oc`. In order for internal objects to be accessible, they are wrapped with a new class: `Internal` that has two fields. The first, of type `Obj`, contains a reference to an internal object. The second, `name`, can be used to filter the search for objects. No two objects with the same `name` can be added to `internals`, which is initialised with entries for all known static objects. Elements of this list can be accessed via method `getNameByObject( o )`, which returns the `name` of object `o` or `null` if `o` is not in `internals`. The algorithm has a table with the low-level identities of all dynamically allocated objects in order to generate correct code when retrieving `internals` as in line 6 in the following code.

The algorithm outputs the code of Listing 5. Line 5 has no effect, since `internals` already has an entry for `extAccount`. In case C_1 and C_2 were swapped, line 5 would bind `f` to name *1*, ensuring the execution of the else-branch in the if statement in line 6.

This example scales to different internally typed parameters in a callback.

Example 5.3 (*Different method of a callback*).

```
1 public createAccount() : Account {
2   extTrans.lock();
3
4 }
```

```
1 public createAccount() : Account {
2   var b : Transaction = extTrans.
        createTransaction();
3 }
```

```
1  public main ( args : String[] ) : Unit {
2    if ( oc.getStep() == 0 ) {
3      oc.incrementStep();
4      var f : Account = extAccount.createAccount();
5      oc.addInternal( new Intern ( f, "extAccount" ) );
6      if ( "extAccount" == oc.getNameByObject( f ) ) {
7        exit( 1 );
8      }else {
9        oc.diverge();
10     }
11   }
12 }
```

Listing 5. Output of the algorithm for Example 5.2.

Trace $\overline{\alpha_1}$ is `extAccount.createAccount()?` · `extTrans.lock()!`, while trace $\overline{\alpha_2}$ is `extAccount.createAccount()?` · `extTrans.createTransaction()!`. In this example, C_1 performs a callback to method `lock`, while C_2 performs it to method `createTransaction`.

To achieve differentiation in this case, the algorithm needs to keep track of the *current method* because it indicates where the differentiating code will be placed. The current method is recorded in a stack that is initially set to method `main` in class `Tester`. Callbacks indicate that the current method is changed to a new entry, and returnbacks indicate that the current method is restored to a previous one. Thus, whenever a callback to method `m` of class `c` is performed, an entry of the form `c.m` is pushed on the stack. A returnback pops the head of the current method stack.

The algorithm outputs the code of Listing 6. Notice that the if-statements of lines 8 and 12, whose addition was discussed in Example 5.1, help the produced code achieve differentiation in this case as well. Should methods `createTransaction()` or `lock()` be called multiple times, the if-guard ensures that the differentiation only happens at the right time.

```
1  public main ( args : String[] ) : Unit {
2    if ( oc.getStep() == 0 ) {
3      oc.incrementStep();
4      var f : Account = extAccount.createAccount();
5    }
6  }
7  public createTransaction() : Transaction {
8    if ( oc.getStep() == 1 ) {
9      oc.diverge();
10   }
11   return null;
12 }
13 public lock() : Int {
14   if ( oc.getStep() == 1 ) {
15     exit( 1 );
16   }
17   return 0;
18 }
```

Listing 6. Output of the algorithm for Example 5.3.

Example 5.4 (*Different callee of a callback*).

```
1  public createAccount() : Account {
2    var b : Transaction = extTrans1.
         createTransaction();
3  }
```

```
1  public createAccount() : Account {
2    var b : Transaction = extTrans2.
         createTransaction();
3  }
```

Trace $\overline{\alpha_1}$ is `extAccount.createAccount()?` · `extTrans1.createTransaction()!`, while trace $\overline{\alpha_2}$ is `extAccount.createAccount()?` · `extTrans2.createTransaction()!`. In this case, the difference is the external object on which the second callback is performed. Here, C_1 calls `createTransaction()` on `extTrans1`, while C_2 calls it on `extTrans2`.

In order to achieve this differentiation, the produced code needs to keep track of external objects similarly to how it needed to keep track of internal objects in Example 5.2. All external objects must be bound to a name, just as the internally allocated ones are. For this purpose, a class `Listable` is created, all the classes `i_c` extend `Listable`. `Listable` contains a `name` and a `type` field, with getters and setters. It also contains a method `setAndRegister( n , t )`, that sets `name = n`, `type = t` and adds the object to a list of `Listable` called `externals` that is kept in object `oc`. Object `oc` contains method `getExternal( n , t )` to retrieve these objects based on `name` and `type`.

The algorithm outputs the code of Listing 7. Fields `name` and `type` for external static objects are initialised in the first instructions of the `main`. That code is omitted for brevity.

```
1  // same main as in Example 5.3
2  public createTransaction() : Transaction {
3    if ( oc.getStep() == 1 ) {
4      if ( this.getName() == "extTrans" ) {
5        exit( 1 );
6      } else {
7        oc.diverge();
8      }
9    }
10   return null;
11 }
```

Listing 7. Output of the algorithm for Example 5.3.

Example 5.5 (*Different callee of a callback #2*).

```
1 public createAccount() : Account {
2   var b : Transaction = extTrans.
        createTransaction();
3   b = b.createTransaction();
4 }
```

```
1 public createAccount() : Account {
2   var b : Transaction = extTrans.
        createTransaction();
3   b = extTrans.createTransaction();
4 }
```

Trace $\overline{\alpha_1}$ is `extAccount.createAccount()?` · `extTrans.createTransaction()!` · `ret 0 x 6?` · `0 x 6.createTransaction()!`, while $\overline{\alpha_2}$ is `extAccount.createAccount()?` · `extTrans.createTransaction()!` · `ret 0 x 6?` · `extTrans.createTransaction()!`. In this case, the difference is the external object on which a callback is performed. Here, C_1 calls `createTransaction()` on `0 x 6`, while C_2 calls the same method on `extTrans`.

The algorithm outputs the code of Listing 8. Lines 14 to 17 ensure that if an external object is not found in the list `externals`, it is allocated by calling to the default factory method and then added to `externals`. Fields `name` and `type` for external static objects are assumed to be initialised in the first instructions of the `main`. That code is omitted for brevity.

Example 5.6 (*Different externally typed callback parameter*).

```
1 public createAccount() : Account {
2   var b : Transaction = extTrans.
        createTransaction();
3   b.callback( b );
4 }
```

```
1 public createAccount() : Account {
2   var b : Transaction = extTrans.
        createTransaction();
3   b.callback( extTrans );
4 }
```

Trace $\overline{\alpha_1}$ is `extAccount.createAccount()?` · `extTrans.createTransaction()!` · `ret 0 x 6?` · `0 x 6.callback( 0 x 6 )!`, while trace $\overline{\alpha_2}$ is `extAccount.createAccount()?` · `extTrans.createTransaction()!` · `ret 0 x 6?` · `0 x 6.callback ( extTrans )!`. This example presents the expected output in case the difference is in a parameter of a callback. The produced code relies on the notions defined in Example 5.4, using the field `name` of external objects to achieve differentiation.

```
1  // same main as in Example 5.3
2  public createTransaction() : Transaction {
3    if ( oc.getStep() == 1 ) {
4      oc.incrementStep();
5    }
6    if ( oc.getStep() == 2 ) {
7      oc.incrementStep();
8      var h :Transaction = oc.getExternal( "0x6", "Transaction" );
9      if ( h == null ) {
10       h = staticForTransaction.defaultCreate();
11       ( ( Listable ) h ).setAndRegister( "0x6", "Transaction" );
12     }
13     return h;
14   }
15   if ( oc.getStep() == 3 ) {
16     if ( this.getName() == "0x6" ) {
17       exit( 1 );
18     } else {
19       oc.diverge();
20     }
21   }
22   return null;
23 }
```

Listing 8. Output of the algorithm for Example 5.5.

The algorithm outputs the code of Listing 9. Casting `arg` to `Listable` is needed in order to make sure the call to `getName()` succeeds. In fact, `arg` is known to implement interface `Transaction`, which has no connection with class `Listable` that defines method `getName()`.

Casts are not present in $\mathcal{J}$+E, yet casting an object o to a type t can be modelled by throwing o and catching an exception of type t. The cast is presented for the sake of simplicity.

```
1  // same main and createTransaction from Example 5.4,
2  // except that lines 20 - 22 are removed
3  public callback( arg : Transaction ) : Unit {
4    if ( oc.getStep() == 3 ) {
5      if ( ( ( Listable ) arg ).getName() == "0x6" ) {
6        exit(1);
7      } else {
8        oc.diverge();
9      }
10   }
11 }
```

Listing 9. Output of the algorithm for Example 5.6.

Example 5.7 (*Traces with different actions*).

```
1 public createAccount() : Account {
2   return extAccount;
3
4 }
```

```
1 public createAccount() : Account {
2   var b : Transaction = extTrans.
        createTransaction();
3 }
```

Trace α_1 is `extAccount.createAccount()?`$\cdot$ `ret` *1*!, while trace α_2 is `extAccount.createAccount()?` $\cdot$ `0 x 6.createTransaction()`!. In this case, the algorithm needs to identify the two different locations where execution will be after the different actions are executed. The concept of current method introduced in Example 5.3 can be used in this case as well in order to determine where to place the code that performs the differentiation.

The algorithm outputs the code of Listing 10.

This examples covers also other cases of different actions, such as a `ret` and a $\surd$ and a `call` and a $\surd$.

```
1  public main ( args : String[] ) : Unit {
2    if ( oc.getStep() == 0 ) {
3      oc.incrementStep();
4      var f : Account = extAccount.createAccount();
5      exit( 1 );
6    }
7  }
8  public createTransaction() : Transaction {
9    oc.diverge();
10   return null;
11 }
```

Listing 10. Output of the algorithm for Example 5.7.

Example 5.8 (*Traces of different length*).

```
1 public createAccount() : Account {
2   while ( 1 == 1 ) { skip; };
3   return null;
4 }
```

```
1 public createAccount() : Account {
2
3   return new AccountClass();
4 }
```

Trace α_1 is `extAccount.createAccount()?`, while trace α_2 is `extAccount.createAccount()?` $\cdot$ `ret` *1*!.

The algorithm outputs the code of Listing 11. When control is returned to `main` after a call to `createAccount()`, it means that the output component is interacting with C_2. In this case, the produced code terminates via the expression of line 5. Divergence is accomplished by C_1.

```
1  public main ( args : String[] ) : Unit {
2    if ( oc.getStep() == 0 ) {
3      oc.incrementStep();
4      var f : Account = extAccount.createAccount();
5      exit( 2 );
6    }
7  }
```

Listing 11. Output of the algorithm for Example 5.8.

As previously mentioned, exceptions are compiled as call and callbacks with a single parameter: the object ID of what is being thrown. Whenever a call to the throw entry point is detected in the traces, a `throw` expression is written in place of a normal method call. The signature of the method must declare that it will throw an exception; otherwise, the last action in the trace will be this thrown exception. This means that after this trace the execution will halt, as no exception to be thrown is registered, but this contradicts the presence of a difference in $\overline{\alpha_1}$ and $\overline{\alpha_2}$. Calls to methods that can throw exceptions are wrapped in a `try/catch` block. If a jump to the external exception handler is detected, then the code is written in the `catch` block. Otherwise, if a normal action is detected, the code is written in the `try` block, below the method call.

5.2. Full Abstraction of the Compilation Scheme

Assumption 1 (Compiler preserves behaviour). *The secure compiler is assumed to output target-level programs that behave as the corresponding input program. Thus, a source-level expression is translated into a list of target-level instructions that preserve the behaviour. By this, we mean that* $C_1^{\downarrow} \simeq C_2^{\downarrow} \Rightarrow C_1 \simeq C_2$.

Notation. Indicate the i-th action of a trace $\overline{\alpha}$ as $\alpha^{(i)}$.

Theorem 5.9 (Algorithm correctness). *For any two source-level components* C_1 *and* C_2 *that, once compiled, exhibit a different trace semantics, the algorithm of Section 5.1*

outputs a component C that differentiates between C_1 and C_2 (assuming there is no overflow of the secure stack and of the secure heap). Formally, $C_1^{\downarrow} \not\approx_T C_2^{\downarrow} \Rightarrow C_1 \not\approx C_2$.

PROOF. As presented in Section 4.1.3, trace semantics deals with *sets* of traces, while the algorithm inputs *single* traces. Moreover, these single traces must be the same up to a !-decorated action. The two different single traces are obtained as follows. Since $C_1^{\downarrow} \not\approx_T C_2^{\downarrow}$, we have that $\mathsf{Traces}(C_1^{\downarrow}) \neq \mathsf{Traces}(C_2^{\downarrow})$; thus, there exists a trace $\overline{\alpha}$ that belongs to either only $\mathsf{Traces}(C_1^{\downarrow})$ or only $\mathsf{Traces}(C_2^{\downarrow})$ but not to both. Assume without loss of generality that $\overline{\alpha} \in \mathsf{Traces}(C_1^{\downarrow})$. The trace $\overline{\alpha}$ can be split in two parts $\overline{\alpha_s}$ and $\overline{\alpha_d}$ such that $\overline{\alpha} = \overline{\alpha_s \alpha_d}$ and such that $\overline{\alpha_s}$ is the longest prefix of all traces of $\overline{\alpha_2}$. So, there exists a trace $\overline{\alpha'} \in \mathsf{Traces}(C_2^{\downarrow})$ that can be split in two parts $\overline{\alpha_s}$ and $\overline{\alpha'_d}$ such that $\overline{\alpha'} = \overline{\alpha_s \alpha'_d}$ and $\overline{\alpha_d} \neq \overline{\alpha'_d}$. Additionally, $\overline{\alpha_2}$ must have even length, so that the different action in the traces is !-decorated and thus generated by either C_1 or C_2. Trace $\overline{\alpha'}$ always exists, it could be an empty trace, it could be composed by an empty $\overline{\alpha_s}$, and, possibly, by an empty $\overline{\alpha'_d}$. The traces input for the algorithm are $\overline{\alpha_1} = \overline{\alpha_s \alpha_d}$ and $\overline{\alpha_2} = \overline{\alpha_s \alpha'_d}$.

The proof analyses all possible differences in $C_1^{\downarrow} \not\approx_T C_2^{\downarrow}$ and proves that the output of the algorithm differentiates between C_1 and C_2, so $C_1 \not\approx C_2$. For each possible difference, the proof refers to an example from Section 5.1 that generate the context capable of performing the differentiation.

—traces of different length (Example 5.8);
—different kind of actions (Example 5.7);
—same kind of actions with differences in their structure:
  —return action
    —different primitive-typed value (Example 5.1);
    —different internally typed value (Example 5.2);
    —different externally typed value (Example 5.6);
  —call action
    —different callee (Example 5.4 and 5.5);
    —different primitive typed parameter (Example 5.1);
    —different internally typed parameter (Example 5.2);
    —different externally typed parameter (Example 5.6);
    —different method (Example 5.3).

As those just listed are the only differences that can appear in two different traces, and beause they all present a counterexample that reaches the contradiction, the theorem holds. □

THEOREM 5.10 (FULL ABSTRACTION OF THE COMPILATION SCHEME). *For any two source-level components C_1 and C_2, we have: $C_1 \simeq C_2 \iff C_1^{\downarrow} \simeq C_2^{\downarrow}$.*

PROOF. The equivalence is split into two subgoals. The direction $\Leftarrow$ holds due to Assumption 1. The direction $\Rightarrow$ is reversed to the equivalent statement: $C_1^{\downarrow} \not\approx C_2^{\downarrow} \Rightarrow C_1 \not\approx C_2$. Apply Proposition 4.7 to restate the statement as $C_1^{\downarrow} \not\approx_T C_2^{\downarrow} \Rightarrow C_1 \not\approx C_2$. Apply Theorem 5.9 to prove the statement. □

6. IMPLEMENTATION AND BENCHMARKS

This section details the architecture adopted to develop the secure compiler of Section 3 (Section 6.1), it compares that architecture to the Intel SGX (Section 6.2) and it presents benchmarking of the overhead introduced by the secure compiler (Section 6.3).

6.1. Protected Module Architecture

The secure compilation scheme of Section 3 relies on the target language having a protected module architecture for it to be secure. In order to time the overhead of the secure compiler, we implemented it for the Fides architecture [Strackx and Piessens 2012].

The Fides architecture implements precisely the protection mechanism described in Section 2.1 in a very small TCB: ∼7,000 lines of code. Fides consists of a hypervisor that runs two virtual machines: the secure VM handles the protected memory section and the Legacy VM handles the unprotected one. Switching between the two virtual machines of Fides (i.e. performing calls and callbacks) is more costly than in a hardware-based implementation. However, we are not interested in this, the overhead that we are interested in timing is the one provided by the additional checks introduced by the secure compiler.

We now give a brief description of the Fides architecture, followed by an informal presentation of the implementation of the secure compiler.

Legacy VM. The Legacy VM executes all legacy applications and other code in unprotected memory. Using virtualisation techniques, this virtual machine is able to execute commodity operating systems and legacy applications without any modification. From the point of view of the Legacy VM, the only difference compared to running on bare hardware is that certain memory locations are inaccessible. More specifically, two memory regions are inaccessible to the Legacy VM: (1) the memory region reserved for the hypervisor and (2) the *protected* memory region as defined in our low-level machine model. Whenever an access to these memory locations is attempted, execution traps to the hypervisor.

Hypervisor. The hypervisor serves two simple purposes. First, it offers a coarse-grained memory protection: it prevents *any* code executing in the Legacy VM from accessing the protected module or the security measure itself (as discussed earlier) and it prevents the Secure VM from accessing the hypervisor.

Second, the hypervisor implements a simple scheduling algorithm. When the Legacy VM calls an entry point in the protected module, control goes to the hypervisor who then schedules the Secure VM. Execution control only returns to the Legacy VM when the protected module either returns or performs a callback to unprotected memory.

Secure VM. The Secure VM can access all memory, with the exception of memory containing the hypervisor. The fine-grained memory access control mechanism is implemented by a security kernel running in this VM, as follows. First, when a request is received from the hypervisor to execute a method in the protected module, the requested entry point is checked against a list of valid entry points provided in the module's memory descriptor. When this check passes, the hardware memory management unit is set up to allow memory accesses to the module's memory region and execution proceeds from the entry point that was called. When execution tries to jump back out of the protected module, a page fault is generated, which causes the security kernel to ultimately return execution control to the Legacy VM.

6.1.1. Secure Build Tools. To simplify the development and benchmarking of modules, we developed a fully abstract compilation tool chain using the LLVM compiler and the ELF Tool Chain library.[6]

Compilation of modules is done in two steps. First, for every function that is annotated as an entry point, an entry in the module's entry point table is created. Each

[6]Respectively available at http://llvm.org/ and http://sourceforge.net/p/elftoolchain/wiki/Home/.

entry checks whether an initialisation function needs to be called, sets up the stack pointer, and stores the return address in unprotected memory on the stack. After the correct function is called, registers not carrying a result value are cleared. Wrapper functions for each entry point are also generated, in order to simplify the calling of the module. Second, the source code is analysed and modified so that every call site that results in a callback to unprotected memory flows through the callback entry point. Registers not carrying a function parameter are reset, so that no information is leaked.

After compilation of the module—possibly resulting in multiple ELF files if the source code was split over multiple files—a secure linker lays out the module in memory according to Fides' requirements. Protected modules must start with the entry table, followed by all compiled code and read-only data such as strings (i.e., the Code section) and the runtime stack and security sensitive variables (i.e., the Data section).

6.2. Intel Software Guard eXtensions

In June 2013, Intel publicly disclosed its work on Software Guard eXtensions (SGX) [McKeen et al. 2013; Anati et al. 2013; Hoekstra et al. 2013]. SGX provides a hardware-implemented isolation mechanism that is very similar to Fides [Strackx and Piessens 2012] and related protected module architectures [Strackx et al. 2013; Noorman et al. 2013; Strackx et al. 2010; Avonds et al. 2013]. *Enclaves* is the SGX terminology for what we called modules in this article. Enclaves live in the same address space as unprotected parts of the application and can only be accessed through an explicitly exposed interface. Direct memory accesses from unprotected memory to enclaves memory regions are prevented. Enclaves, like modules, have full access to unprotected parts of the application. We believe that the presented fully abstract compilation scheme can be easily ported to SGX-enabled platforms, modulo small technical changes.

There are a few notable differences between Fides and SGX. For instance, SGX requires special entry and exit instructions to cross enclaves boundaries, while Fides does not. SGX also provides only a single entry point to enclaves [Intel Corporation 2013], but additional entry points can be emulated by taking the index of the intended function as an additional function argument. The main difference between Fides and SGX is the following. To prevent denial-of-service attacks by buggy or malicious modules that never return control to code in unprotected memory, SGX supports interruption of enclaves. Unfortunately, without added security measures, this may violate the integrity of SGX enclaves, as explained in the following text.

PROBLEM 8 (SGX INTERRUPTS). *Consider two classes that define the same methods* `plusTwo` *and* `reset` *that, respectively, increment a variable* `even` *by two and reset that variable to 0. These two classes are implemented by two objects:* o_L *and* o_R*. Assume these objects are compiled to SGX enclaves.*

```
1  package p;
2  class C_L {
3    private even : Int = 0;
4
5    public plusTwo() : Int {
6      even = even + 1;
7      even = even + 1;
8      return 0;
9    }
10
11   public reset() : Int {
12     even = 0;
13     return 0;
14   }
15 }
16 object o_L : C_L
```

```
1  package p;
2  class C_R {
3    private even : Int = 0;
4
5    public plusTwo() : Int {
6      even = even + 2;
7
8      return 0;
9    }
10
11   public reset() : Int {
12     even = 0;
13     return 0;
14   }
15 }
16 object o_R : C_R
```

Objects o_L and o_R are equivalent in source-code level, but their compiled counterparts are not. Enclave $o_L \downarrow$ may be interrupted before executing the `plusTwo` *method at line 7. The interrupt can cause a call to* `reset` *to be made before execution is resumed. This will result in* `even` *holding value* `1` *in o_L, while o_R will either have value 0 or 2 in* `even`.

This integrity constraint violation can be prevented by ensuring that new entry points cannot be called while an interrupt is handled. In practice, this can be accomplished by adding a Boolean field `busy` to the compiled code. The `busy` field is set when the module is entered and reset before control is passed back to unprotected memory. In case a module is interrupted, it has still the `busy` variable set and the module may refuse to service the function request. To avoid race conditions, an atomic `test-and-set` instruction should be used to keep track of the `busy` variable.

Finally, SGX, in contrast to Fides, also provides protection against hardware attacks such as an attacker snooping the memory bus [Winter and Dietrich 2012] or performing a cold boot attack [Halderman et al. 2008]. This enables various TPM primitives to be offloaded to the main CPU and advocates the importance of fully abstract compilation. This does not impact the development of a fully abstract compilation scheme.

6.3. Measurements

To benchmark the cost of the additional checks introduced by the secure compilation scheme, we have implemented stub objects in C, a data structure that models the low-level representation of objects. Stub objects have an `Integer` field that indicates the class of the object followed by the fields of the object. We have implemented a secure runtime containing the data structure $\mathcal{O}$ and functions to mask object references through it. The secure runtime also implements the runtime checks presented in Section 3. These tests are simply used as an indicator that the overhead introduced by the compiler is reasonable. More detailed measurements, detailing the strengths and limitations of particular PMA implementations are left for future work.

We have then taken a simple program and, using a hardware high-frequency timestamp, we have timed its performance in three cases, as presented in Figure 13. The figure presents the average program execution time without any protection (in blue), with Fides (in red), and with Fides extended with the runtime checks provided by the security runtime (in beige).

In Figure 13, the y axis indicates which operations have been tested. The number following calls and callbacks indicates the number of arguments used and which trigger runtime checks. The security runtime adds checks to calls, callbacks, returns, and returnbacks, so they are the only instructions that are considered. The “Normal” (blue) row indicates the cost of each operation without using the Fides architecture. The “Fides” (red) row indicates the cost of each operation while using the Fides architecture without the secure compilation scheme. The “Prototype” (beige) row the cost of each operation when the secure compilation scheme is used in addition to Fides. Each operation was performed 1,000 times on a MacBook Pro with a 2.3GHz Intel Core i5 processor and 4GB 1,333MHz DDR3 RAM. The difference between rows “Normal” and “Fides” shows the already high overhead of adopting the Fides architecture. The difference between rows “Fides” and “Prototype” is the overhead of the security checks introduced by the secure compiler, on average, this is a $\sim$3% overhead. Security checks are triggered only when the boundary between the protected and the unprotected memory partitions is crossed. Method calls within the same memory partition suffer no overhead. The overhead introduced by the compiler is proportional to the number of boundaries crossing.

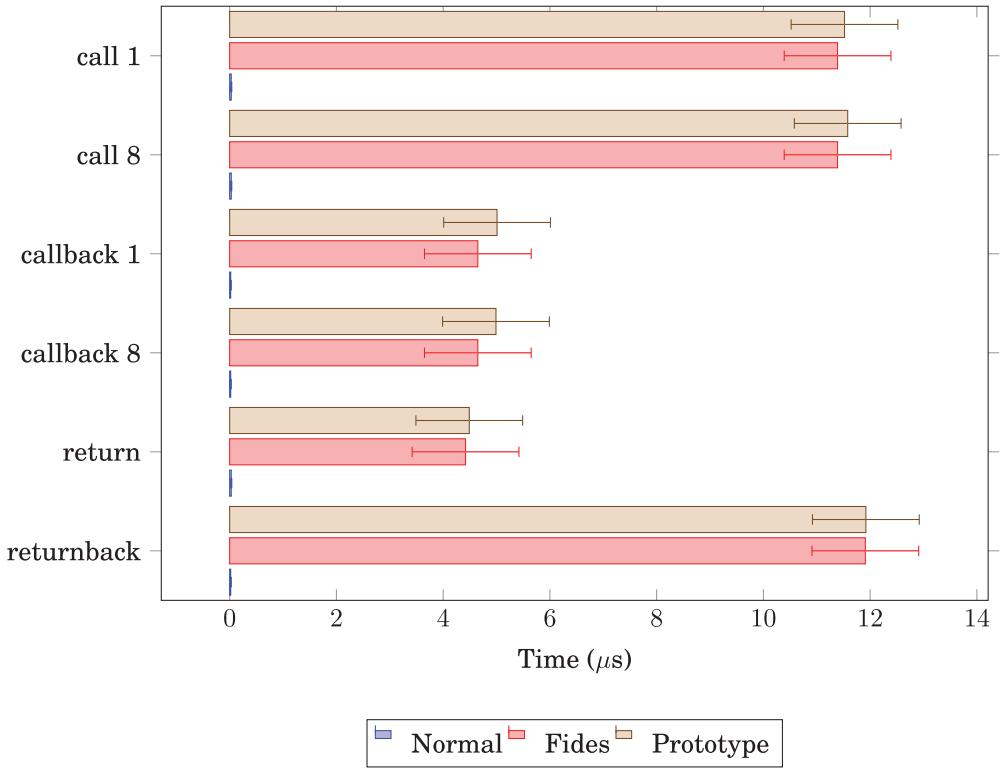

Fig. 13. Cost of different benchmarked instructions.

7. DISCUSSION

This section describes how to extend the secure compilation scheme of Section 3 to other language features such as first-order method references, cross-package inheritance, and inner classes (Section 7.1). Then it discusses limitations to the current work and how to overcome some of them (Section 7.2).

7.1. Secure Compilation of Additional Language Features

This section describes how to securely compile first-order method references, cross-package inheritance, and inner classes.

7.1.1. First-Order Method References. Right now the address of a callback cannot be supplied by external code as the calling convention specifies where the dynamic dispatch of external code is located. If first-order method references are allowed, method names can be supplied as parameters of other methods in order to be called. This means that at the $\mathcal{A}$+I level, attackers can supply arbitrary addresses instead of these parameters, raising the following problem.

PROBLEM 9 (ILLEGAL ADDRESSES). *Consider two classes that define a method* `doCallback` *that inputs a reference to a method* `cb`*, calls that method, and then continues by performing the same computation. These two classes are implemented by two objects:* o_L *and* o_R.

```
1  package p;
2  class C_L {
3    private f : Int = 1;
4
5    public doCallback( cb : Unit → Unit )
         : Int {
6      cb();
7      f += 1;
8      f -= 1;
9      return f;
10   }
11 }
12 object o_L : C_L
```

```
1  package p;
2  class C_R {
3    private f : Int = 1;
4
5    public doCallback( cb : Unit → Unit )
         : Int {
6      cb();
7      f -= 1;
8      f += 1;
9      return f;
10   }
11 }
12 object o_R : C_R
```

Objects o_L and o_R are equivalent in $\mathcal{J}$+E, as in both objects the method `doCallback` *always returns 1. Once compiled, an A+I attacker can differentiate between them by giving the address of the instructions corresponding to line 7 as the callback* `cb`*. In this case, o_L will decrement* `f` *without first incrementing it, while o_R will increment* `f` *without first decrementing it. This will result in* `f` *having a value of* 0 *in o_L and* 2 *in o_R. This is similar to a return-oriented programming attack [Roemer et al. 2012].*

To counter this attack, the compiler must ensure the integrity of control flow when jumping from a protected module to an externally supplied address. For a call to a method whose address was externally supplied, a valid destination address is (1) an address outside of the memory bounds of the module or (2) the address of one of the entry points. The compiler must add runtime checks for these conditions at each indirect call. In case of a jump to the entry point, the additional checks provided at the entry point will ensure that the supplied address had a signature that matches that specified in the source-level component.

7.1.2. Secure Compilation of Cross-Package Inheritance. Cross-package inheritance arises whenever a class from an export package `extends` a class from a different export package, as in Listing 12, where `JointAccount` from `P-SUB` extends `Account` from `P-SUPER`. Cross-package inheritance is not provided by $\mathcal{J}$+E, as it breaks the encapsulation property of the language, creating a breach in the security of the language. Nevertheless, as there can be cases in which this feature is desirable, this article now discusses how to securely implement it.

In order to allow cross-package inheritance and preserve as many benefits as possible from the encapsulation property of $\mathcal{J}$+E, classes that can be extended must appear in import packages. Thus, given an import package, entry points are created not only for interface-defined methods but also for class-defined ones and for constructors. Class `JointAccount` can optionally override methods of the super class `Account`, as is the case with method `withdraw()`. Within those methods, calls to `super` can be used in order

```
1  package P-SUPER;
2    class Account {            // called the super class
3      public withdraw():Int { ··· }
4      public getBalance(): Int { ··· }
5    }
6
7  package P-SUB;
8    class JointAccount extends P-SUPER.Account {    // called the sub class
9      public withdraw():Int{
10       super.withdraw();
11       ···
12     }
13   }
```

Listing 12. Example of cross-package inheritance.

to call method `withdraw()` of the super class `Account`. Alternatively, if a method is not overridden (e.g. such as `getBalance`), calling *j*`.getBalance()` on an object *j* of type `JointAccount` executes method `getBalance()` defined in the super class `Account`.

If the normal compilation scheme were followed, at the target-level *j* is allocated to a single memory area where fields from classes `Account` and `JointAccount` are both allocated. If cross package inheritance involves only secure or only insecure classes (e.g., if `P-SUPER` and `P-SUB` from Listing 12 belong to the same component), it does not generate any problem. However, when secure classes can extend insecure ones and vice-versa, some complications arise, as presented in Problem 10.

PROBLEM 10 (ALLOCATION OF *j*). *Consider the case when* `Account` *is protected and* `JointAccount` *is not. If j is allocated outside the protected memory partition, private fields of the* `Account` *subobject become accessible to external code. If j is allocated inside the protected memory partition, two options arise. The first one is placing untrusted methods of* `JointAccount` *in the protected memory partition, violating the security of the compilation scheme. Otherwise, if methods of* `JointAccount` *are placed in the unprotected memory partition, they cannot access* `JointAccount`*'s fields via offset. Getters and setters for fields of* `JointAccount` *could be exposed through entry points, but this would violate full abstraction, as those methods are not available at the high level.*

These problems also arise when `Account` *is not protected but* `JointAccount` *is, so compilation of cross-package inheritance cannot be achieved normally.*

To overcome these difficulties, when *j* is allocated, it is split in two subobjects: *ja*, with fields of class `Account`, and *jj*, with fields of class `JointAccount`; the object identity of *j* is *jj* [van Dooren et al. 2013].

First, consider the case when `Account` is protected and `JointAccount` is not. External code needs to compile the expression *j* `= new JointAccount()` so that it calls `new Account()` to create object *ja* in the protected memory section. External code must then save the resulting identifier for *ja* to perform `super` calls, since they are translated as method calls. The additional checks inserted at entry points presented in Section 3 ensure that `super` calls are always well typed.

Consider then the case when `Account` is not protected and `JointAccount` is. The secure compiler needs to call `new JointAccount()` and save the returned object identity for *jj* in a memory location, since `super` calls in this case are compiled as callbacks. When expression *j* `= new JointAccount()` is compiled, the unprotected address *ja* is stored at the low-level, right after the type of *jj*. The expression `super.withdraw()` is compiled as *ja*`.withdraw()`.

The creation of two separate objects may seem to break full abstraction of the compilation scheme in a way similar to what Abadi found out for inner classes [Abadi 1999] in the early JVM. In fact, target level external code is given the functionality to call *ja*`.withdraw()`, which is not explicitly possible in the high-level language. However, *j*`.super.withdraw()` is an implicit call to the `withdraw()` method of `Account`, functionality that the high-level language already has. This way of handling cross-package inheritance does not add functionality at the target level, so it does not break full abstraction of the compilation scheme.

7.1.3. Secure Compilation of Inner Classes. Inner classes are classes that are defined inside another class, as in Listing 13. Inner classes have access to private fields of the class they are defined within. They have not been included in the formalisation of $\mathcal{J}$+E as to keep it as simple as possible.

Inner classes of the secure component are compiled as normal classes in the protected memory partition, in the usual fashion. To implement access from the inner class to the private fields of the surrounding class in a JVM style [Flanagan 1998], a getter and

```
1 class AccountClass implements P-Import.Account {
2   AccountClass() { counter = 0; }
3   private counter : Int;
4
5   class Inner { // Inner has access to counter }
6 }
```

Listing 13. Example of an inner class.

a setter for each private field are created. In the case of Listing 13, class `AccountClass` is extended with getters and setters for the `counter` field when compiled. Access from `Inner` to `counter` is compiled as method calls via the getter and setter.

This approach is inspired by Abadi [1999],who shows that it breaks full abstraction of compilation in an early version of the JVM. In that setting, the additional low-level methods are not available at the high level; thus, other low-level code besides the inner classes can call those methods, achieving something that was not possible at the high level. In our secure compilation scheme, the additional methods are available in the surrounding class. However, the additional methods are not made available through entry points; thus, the external code cannot invoke them. This means that the addition of inner classes to the secure compilation scheme preserves the full abstraction property.

7.2. Limitations

This section presents limitations of the presented compilation scheme. Then it informally discusses how to perform garbage collection when part of the program is compiled securely.

Like many model languages [Abadi and Plotkin 2012; Jagadeesan et al. 2011], $\mathcal{J}$+E lacks features that real-world programming languages have, such as multithreading, foreign-function interfaces, and garbage collection. A thorough investigation of the changes needed in order to support secure compilation of languages with those features is left for future work. For now, we informally present the research challenges of performing garbage collection in concert with securely compiled code and sketch the path future research could follow to address these challenges.

Garbage collection is a runtime addition that handles whole programs. Since in the secure compilation scenario whole programs are split between the protected and the unprotected memory partition, the garbage collector (GC) is also split into a protected and an unprotected part. Given the powers of an attacker to the system (Definition 2.6 in Section 2.6), the attacker can tamper with the unprotected GC but not with the protected one. The attacker can thus inspect all references that the unprotected GC has, introduce fake pointers, and impersonate the unprotected GC to interact with the protected one.

Since the unprotected GC can be tampered with, the implementation of a secure garbage collector is reduced to extending the securely compiled program with a secure GC in charge of the secure memory partition. The secure GC must be trusted and allocated inside the protected memory partition, so it can access $\mathcal{O}$ and the object graphs of the protected objects. However, to allow the secure GCs to communicate with a GC in unprotected memory (for the case when there is no attacker), additional entry points need to be set up. Unfortunately, this violates full abstraction in a similar way to that pointed out by Abadi for Java (Section 7.1.3), these functionalities are available only at the target level and not at the source level. This is a first challenge: proving that the secure GC does not introduce security leaks, a possible approach to such a proof is described at the end of this section.

Additional challenges arise for the implementation of the secure GC. A common implementation for GCs is reference counting. With reference counting, the GC keeps track of how many references an object has, when this counter reaches 0, then the objects can be safely deallocated, as no other object has a reference to it. Reference counting introduces a failure of full abstraction, as highlighted by the following code.

```
1 package p;
2 class C_L {
3   public doCb( Object x , Object y ) {
4     while ( ··· ) {   //infinite loop
5       x.callback( this );
6     }
7   }
8   y.callback();
9 }
```

```
1 package p;
2 class C_R {
3   public doCb( Object x , Object y ) {
4     while ( ··· ) {   //infinite loop
5       x.callback( this );
6     }
7   }
8
9 }
```

In these code snippets, both functions `doCb` receive two arguments `x` and `y` (line 3) and loop infinitely (line 4) on performing callbacks on `x` (line 5). Additionally, the left hand side snippet has unreachable code where `callback` is called on `y` as well (line 8). A garbage collector that does reference counting will behave differently in these two cases. In fact, it will keep a reference to `y` in C_L and not in C_R, as it cannot know that `y` lies in unreachable code without solving the halting problem. This is a failure of full abstraction: C_L and C_R behave the same at the source level but not at the target level, when a garbage collector is considered. A simple solution to problem would be to change the way references are counted and let parameters also increase the counter for an object reference. The return would decrease the counter for all parameters.

However, a second challenge arises: once a reference to an internal object is leaked to the unprotected code (such as for `this` in the `callback` just mentioned) the GC does not know when to deallocate such a reference. Here, an arguably safe methodology is to not deallocate a reference that is passed from the secure component to unprotected code. However, this creates problems when the allocated object is large or when many references are passed out.

An analogy that can be made now is that the secure GC faces challenges similar to those faced by distributed garbage collectors. Passing a reference to unprotected code is in fact analogous to passing a reference to a remote program: it is complex to know when such a reference can be deallocated. In the distributed setting, this is due to communication problems and the impossibility for a garbage collector to inspect the object graph of a program on a remote machine. In this setting, this is due to the fact that the object graph in the unprotected memory section can be tampered with by the attacker. Unfortunately, in this setting, the GC needs not only to be performant in case of interaction with unprotected code, it needs also to defend from potential attackers. Research on distributed garbage collection has developed several ways to addresses this problem [Abdullahi and Ringwood 1998], for example, by giving each leaked reference a lease time. An unprotected object receiving a leased reference must periodically renew the lease on the reference, because once the lease time has expired, the reference is collected by the secure GC. This solution could be adopted in order to implement a secure memory manager, for it already addresses the need to provide a performant GC algorithm in case the unprotected code is well behaved. The details of the implementation, of how to make such a GC resilient against attacks and a more thorough treatment of the problems arising in various implementations are left for future work.

The proof that the garbage collector is secure remains a great challenge for the integration of secure compilation and GC. In any way such a proof is approached, GC notions (allocation, deallocation, references, the object graph, etc.) need to be carried into both the target and the source language in order to prove the compilation scheme

between the two to be fully abstract. However, this causes the source-level programming model to become hindered by memory management—an additional way to let programmers introduce security flaws in their code. A way to overcome this challenge is to capture the behaviour of the GC in a different semantics of the source-level language, an *extended semantics*. Full abstraction of the compilation scheme should then be proven with respect to the extended semantics. This treatment would not clutter the source language with explicit GC, but it would still capture the effect of GC at the source level. The extended semantics should capture the behaviour of the GC as it is at the target level, so that problems as those highlighted earlier do not arise, since they are captured at the source level as well. Additionally, the extended semantics should be proven to be secure with respect to the normal operational semantics: this would guarantee that the behaviour of the GC does not introduce security leaks.

Devising such an extended semantics and proving (i) that it is not introducing security leaks and (ii) a fully abstract translation involving it are left for future work.

8. RELATED WORK

Secure compilation through full abstraction was pioneered by Abadi [1999], where, alongside a result in the π-calculus setting, Java bytecode compilation in the early JVM is shown to expose methods used to access private fields by private inner classes. Kennedy [2006] listed six full abstraction failures in the compilation to .NET, half of which have been fixed in modern $C^{\#}$ implementations.

Address space layout randomisation has been adopted by Abadi and Plotkin [2012] and subsequently by Jagadeesan et al. [2011] to guarantee probabilistic full abstraction of a compilation scheme. In both works the low-level language is more high-level than ours and the protection mechanism is different. Compilation does not necessarily need to target machine code, as Fournet et al. [2013] show by providing a fully abstract compilation scheme from an ML dialect named F^{*} to JavaScript that relies on type-based invariants. Similarly, Ahmed and Blume [2011] prove full abstraction of a continuation-passing style translation from simply typed λ-calculus to System F. In both works, the target-level language is typed and more high-level than the one of this article. The checks introduced by our compilation scheme are simpler than the checks of Fournet et al., so secure compilation seems easier to implement for object-oriented languages than for functional ones. An alternative target language for secure compilation is Typed Assembly Language (TAL), which was proposed in the work of Morrisett et al. [1999]. The presence of the Intel SGX processor strengthens our choice of the PMA architecture for the target language over TAL.

Adopting a fine-grained, program counter-based memory access control mechanism to achieve secure compilation was pioneered by Agten et al. [2012] and then investigated by Patrignani et al. [2013]. This article wraps both results in a coherent presentation, extending certain aspects of both articles.

A large amount of work on secure compilation involved the compilation of unsafe languages such as C. An extensive survey can be found in Younan et al. [2012]. Instead of focussing on a fully abstract compilation, that research is devoted to strengthening the security properties of C.

Correct (and certified) compilation, are very broad research fields that aim at providing compilers that preserve contextual equivalence between the source and target language [Chlipala 2007; Leroy 2009]. The main difference between those fields and secure compilation is the nature of low-level contexts. In order to model a powerful low-level attacker, we have considered an arbitrary complex low-level context interacting with the compiled components. In those works, the low-level context is assumed to be obtained through compilation, so it does not misbehave. A great contribution of correct compilation works is that they provide compilers for which Assumption 1 holds.

Different security architectures with access control mechanisms comparable to ours have been developed in the last years: TrustVisor [McCune et al. 2010], Flicker [McCune et al. 2008], Nizza [Singaravelu et al. 2006], SPMs [Strackx et al. 2010; Strackx and Piessens 2012], and the Intel SGX [McKeen et al. 2013]. Similar architectures with fine-grained protection schemes have also been employed to develop less monolithic operating systems [Witchel et al. 2002]. A common characteristic of these security architectures is that they rely on a small TCB, either only the hardware, or the hardware plus a hypervisor small enough to be formally verifiable [Vasudevan et al. 2013]. The existence of industry prototypes alongside research ones underlines the feasibility bringing efficient and secure low-level memory access control in commodity hardware. No results comparable to ours have been proven for these systems.

9. CONCLUSION AND FUTURE WORK

This article presented a fully abstract compilation scheme for a strongly typed, single-threaded, component-based, object-oriented programming language with dynamic memory allocation and exceptions. The compilation scheme targets untyped assembly code enhanced with a protected modules architecture. From the security perspective a fully abstract compilation scheme ensures that target-level attackers are restricted to the same powers source-level attackers have. Additionally, it guarantees source-level reasoning: in order to understand how a program behaves, it is sufficient to inspect its source code. This article highlighted mistakes that make a naïve compilation scheme not fully abstract and how to correct them. The compilation scheme is proven to be fully abstract, guaranteeing preservation and reflection of contextual equivalence between high-level components and their compiled counterparts.

As presented in Section 7, integrating advanced language features such as garbage collection with securely compiled code is a challenging and relevant research direction. Additionally, providing secure compilers for multicore architectures and for multi-principal scenarios (i.e., involving more than one protected partition, each with code coming from different stakeholders) are important, future work challenges. Finally, the efficient transfer of large data values will impact on the performance and usability of implementations of secure compilers, this is also a challenge that is left for future work.

ACKNOWLEDGMENTS

We would like to thanks the anonymous reviewers for their insightful comments and suggestions on previous versions of this article.

REFERENCES

Martín Abadi. 1999. Protection in programming-language translations. In *Secure Internet Programming*. Springer-Verlag, London, UK, 19–34.

Martín Abadi and Gordon D. Plotkin. 2012. On protection by layout randomization. *ACM Trans. Inf. Syst. Secur.* 15, 2, Article 8 (July 2012), 29 pages. DOI: http://dx.doi.org/10.1145/2240276.2240279

Saleh E. Abdullahi and Graem A. Ringwood. 1998. Garbage collecting the internet: A survey of distributed garbage collection. *ACM Comput. Surv.* 30, 3 (Sept. 1998), 330–373. DOI: http://dx.doi.org/10.1145/292469.292471

Pieter Agten, Bart Jacobs, and Frank Piessens. 2015. Sound modular verification of C code executing in an unverified context. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'15)*.

Pieter Agten, Raoul Strackx, Bart Jacobs, and Frank Piessens. 2012. Secure compilation to modern processors. In *Proceedings of the 2012 IEEE 25th Computer Security Foundations Symposium (CSF'12)*. IEEE Computer Society, Washington, DC, Article 12, 15 pages. DOI: http://dx.doi.org/10.1109/CSF.2012.12

Amal Ahmed and Matthias Blume. 2011. An equivalence-preserving CPS translation via multi-language semantics. *SIGPLAN Not.* 46, 9, Article 30 (Sept. 2011), 14 pages. DOI:http://dx.doi.org/10.1145/2034574.2034830

Ittai Anati, Shay Gueron, S. Johnson, and V. Scarlata. 2013. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP'13)*, Vol. 13.

Niels Avonds, Raoul Strackx, Pieter Agten, and Frank Piessens. 2013. Salus: Non-hierarchical memory access rights to enforce the principle of least privilege. In *Security and Privacy in Communication Networks (SecureComm'13)*.

Adam Chlipala. 2007. A certified type-preserving compiler from lambda calculus to assembly language. *SIGPLAN Not.* 42, 6, Article 5 (June 2007), 12 pages. DOI:http://dx.doi.org/10.1145/1273442.1250742

Pierre-Louis Curien. 2007. Definability and full abstraction. *Electron. Notes Theor. Comput. Sci.* 172 (April 2007), 301–310. DOI:http://dx.doi.org/10.1016/j.entcs.2007.02.011

Frank S. de Boer, Marcello M. Bonsangue, Martin Steffen, and Erika Ábrahám. 2005. A fully abstract semantics for UML components. In *Proceedings of the 3rd International Conference on Formal Methods for Components and Objects (FMCO'04)*. Springer-Verlag, Berlin, 49–69. DOI:http://dx.doi.org/10.1007/11561163_3

Roland Ducournau. 2011. Implementing statically typed object-oriented programming languages. *ACM Comput. Surv.* 43, 3, Article 18 (April 2011), 48 pages. DOI:http://dx.doi.org/10.1145/1922649.1922655

Karim Eldefrawy, Aurélien Francillon, Daniele Perito, and Gene Tsudik. 2012. SMART: Secure and minimal architecture for (establishing a dynamic) root of trust. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS'12)*.

Ulfar Erlingsson, Yves Younan, and Frank Piessens. 2010. Low-level software security by example. In *Handbook of Information and Communication Security*. Springer, Berlin, 663–658.

David Flanagan. 1998. *Java in a Nutshell. Deutsche Ausgabe der 2. A.* O'Reilly.

Cedric Fournet, Nikhil Swamy, Juan Chen, Pierre-Evariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. 2013. Fully abstract compilation to JavaScript. *SIGPLAN Not.* 48, 1, Article 26 (Jan. 2013), 14 pages. DOI:http://dx.doi.org/10.1145/2480359.2429114

J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. 2008. Lest we remember: Cold boot attacks on encryption keys. In *Proceedings of the USENIX Security Symposium*. 45–60.

Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. 2013. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 11. Retrieved from

Andrei Homescu, Steven Neisius, Per Larsen, Stefan Brunthaler, and Michael Franz. 2013. Profile-guided automated software diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'13)*. IEEE Computer Society, Washington, DC, 1–11. DOI:http://dx.doi.org/10.1109/CGO.2013.6494997

Intel Corporation. 2013. *Software Guard Extensions Programming Reference*. Retrieved from http://software.intel.com/sites/default/files/329298-001.pdf.

Radha Jagadeesan, Corin Pitcher, Julian Rathke, and James Riely. 2011. Local memory via layout randomization. In *Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium (CSF'11)*. IEEE Computer Society, Washington, DC, 161–174. DOI:http://dx.doi.org/10.1109/CSF.2011.18

Alan Jeffrey and Julian Rathke. 2005a. A fully abstract may testing semantics for concurrent objects. *Theor. Comput. Sci.* 338, 1–3 (June 2005), 17–63. DOI:http://dx.doi.org/10.1016/j.tcs.2004.10.012

Alan Jeffrey and Julian Rathke. 2005b. Java Jr.: Fully abstract trace semantics for a core Java language. In *ESOP'05 (LNCS)*, Vol. 3444. Springer, 423–438. DOI:http://dx.doi.org/10.1007/978-3-540-31987-0_29

Andrew Kennedy. 2006. Securing the .NET programming model. *Theor. Comput. Sci.* 364, 3 (Nov. 2006), 311–317. DOI:http://dx.doi.org/10.1016/j.tcs.2006.08.014

Per Larsen, Stefan Brunthaler, and Michael Franz. 2014. Security through diversity: Are we there yet? *IEEE Security & Privacy* 12, 2 (2014), 28–35. DOI:http://dx.doi.org/10.1109/MSP.2013.129

Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*.

Xavier Leroy. 2009. A formally verified compiler back-end. *J. Autom. Reason.* 43, 4 (Dec. 2009), 363–446. DOI:http://dx.doi.org/10.1007/s10817-009-9155-4

Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. 2010. TrustVisor: Efficient TCB reduction and attestation. In *SP'10*. IEEE, Washington, DC, 143–158. DOI:http://dx.doi.org/10.1109/SP.2010.17

Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. 2008. Flicker: An execution infrastructure for TCB minimization. *SIGOPS Oper. Syst. Rev.* 42, 4, Article 24 (April 2008), 14 pages. DOI:http://dx.doi.org/10.1145/1357010.1352625

Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative instructions and software model for isolated execution. In *HASP'13*. ACM, Article 10, 1 pages. DOI:http://dx.doi.org/10.1145/2487726.2488368

Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. From system f to typed assembly language. *ACM Trans. Program. Lang. Syst.* 21, 3 (May 1999), 527–568. DOI:http://dx.doi.org/10.1145/319301.319345

Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. 2013. Sancus: Low-cost trustworthy extensible networked devices with a zero-software Trusted Computing Base. In *Proceedings of the 22nd USENIX Conference on Security Symposium*. USENIX Association.

Marco Patrignani and Dave Clarke. 2014. Fully abstract trace semantics of low-level isolation mechanisms. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC'14)*.

Marco Patrignani, Dave Clarke, and Frank Piessens. 2013. Secure compilation of object-oriented components to protected module architectures. In *APLAS'13 (LNCS)*, Vol. 8301. 176–191.

Gordon D. Plotkin. 1977. LCF considered as a programming language. *Theoretical Computer Science* 5 (1977), 223–255.

Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.* 15, 1, Article 2 (March 2012), 34 pages. DOI:http://dx.doi.org/10.1145/2133375.2133377

Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS'04)*. ACM, New York, NY, 298–307. DOI:http://dx.doi.org/10.1145/1030083.1030124

Lenin Singaravelu, Calton Pu, Hermann Härtig, and Christian Helmuth. 2006. Reducing TCB complexity for security-sensitive applications: Three case studies. *SIGOPS Oper. Syst. Rev.* 40, 4, Article 13 (April 2006), 14 pages. DOI:http://dx.doi.org/10.1145/1218063.1217951

Raoul Strackx, Job Noorman, Ingrid Verbauwhede, Bart Preneel, and Frank Piessens. 2013. Protected software module architectures. In *Proceedings of the ISSE 2013 Securing Electronic Business Processes*, Helmut Reimer, Norbert Pohlmann, and Wolfgang Schneider (Eds.). Springer, 241–251. DOI:http://dx.doi.org/10.1007/978-3-658-03371-2_21

Raoul Strackx and Frank Piessens. 2012. Fides: Selectively hardening software application components against kernel-level or process-level Malware. In *Proceedings of the 2012 ACM conference on Computer and Communications Security (CCS'12)*. ACM, New York, NY, 2–13. DOI:http://dx.doi.org/10.1145/2382196.2382200

Raoul Strackx, Frank Piessens, and Bart Preneel. 2010. Efficient isolation of trusted subsystems in embedded systems. In *SecureComm*. 344–361.

Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. 2009. Breaking the memory secrecy assumption. In *Proceedings of the 2nd European Workshop on System Security (EUROSEC'09)*. ACM, New York, NY. DOI:http://dx.doi.org/10.1145/1519144.1519145

Marko van Dooren, Dave Clarke, and Bart Jacobs. 2013. Subobject-oriented programming. In *Formal Methods for Components and Objects (Lecture Notes in Computer Science)*, Vol. 7866. Springer, Berlin, 38–82. DOI:http://dx.doi.org/10.1007/978-3-642-40615-7_2

Amit Vasudevan, Sagar Chaki, Limin Jia, Jonathan McCune, James Newsome, and Anupam Datta. 2013. Design, implementation and verification of an extensible and modular hypervisor framework. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP'13)*. IEEE Computer Society, Washington, DC, 430–444. DOI:http://dx.doi.org/10.1109/SP.2013.36

Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. 2012. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12)*. ACM, New York, NY, 157–168. DOI:http://dx.doi.org/10.1145/2382196.2382216

Johannes Winter and Kurt Dietrich. 2012. A hijacker's guide to the LPC bus., In *Proceedings of the 8th European conference on Public Key Infrastructures, Services, and Applications (EuroPKI'11)*. 176–193.

Emmett Witchel, Josh Cates, and Krste Asanović. 2002. Mondrian memory protection. *SIGPLAN Not.* 37, 10 (Oct. 2002), 304–316. DOI:http://dx.doi.org/10.1145/605432.605429

Yves Younan. 2008. *Efficient Countermeasures for Software Vulnerabilities due to Memory Management Errors*. Ph.D. Dissertation. Informatics Section, Department of Computer Science, Faculty of Engineering Science.

Yves Younan, Wouter Joosen, and Frank Piessens. 2012. Runtime countermeasures for code injection attacks against C and C++ programs. *Comput. Surveys* 44, 3 (2012), 17:1–17:28.

Received January 2014; revised July 2014; accepted October 2014