Secure Compilation of Object-Oriented Components to Protected Module Architectures – Extended Version*

Marco Patrignani, Dave Clarke, and Frank Piessens

iMinds-DistriNet, Dept. Computer Science, K. U. Leuven
{first.last}@cs.kuleuven.be

Abstract. A fully abstract compilation scheme prevents the security features of the high-level language from being bypassed by an attacker operating at a particular lower level. This paper presents a fully abstract compilation scheme from a realistic object-oriented language with dynamic memory allocation, cross-package inheritance, exceptions and inner classes to untyped machine code. Full abstraction of the compilation scheme relies on enhancing the low-level machine model with a fine-grained, program counter-based memory access control mechanism. This paper contains the outline of a formal proof of full abstraction of the compilation scheme. Measurements of the overhead introduced by the compilation scheme indicate that it is negligible.

Note: This technical report supersedes number 630: *Secure Compilation of Object-Oriented Components to Untyped Machine Code.*

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, or encapsulation primitives. These mechanisms can be used as security building blocks to withstand the threat of attackers acting at the high level. For the software to be secure, attackers acting at lower levels need to be considered as well. Thus it is important that high-level security properties are preserved after the high-level code is compiled to machine code. Such a secrecy-preserving compilation scheme is called *fully abstract* [1]. An implication of such a compilation scheme is that the power of a low-level attacker is reduced to that of a high-level one. The notion of fully abstract compilation is well suited for expressing the preservation of security policies through compilation, as it preserves and reflects contextual

* This work has been supported in part by the Intel Lab'sUniversity 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 holds a Ph.D. fellowship from the Research Foundation Flanders (FWO).

equivalence. Two programs are contextually equivalent if they cannot be distinguished by a third one. Contextual equivalence models security policies as follows: saying that variable f of program C is confidential is equivalent to saying that C is contextually equivalent to any program C' that differs from C in its value for f. A fully abstract compilation scheme does not eliminate high-level security flaws. It is, in a sense, conservative, introducing no more vulnerabilities at the low level than the ones already exploitable at the high level.

Fully abstract compilation of modern high-level languages is hard to achieve. Compilation of Java to JVM or of C$^{\#}$ to the .NET framework [12] are some of the examples where compilation is not fully abstract. Recent techniques that achieve fully abstract compilation rely on address space layout randomisation [2,9], type-based invariants [4,7], and enhancing the low-level machine model with a fine-grained program counter-based memory access control mechanism [3].

The threat model considered in this paper is that of an attacker with low-level code execution privileges. Such an attacker can inject and execute malicious code at machine level and violate the security properties of the machine code generated by the compiler. In order to withstand such a low-level attacker, high-level security features must be preserved in the code generated during compilation. Agten *et al.* [3] were the first to show that fully abstract compilation of a safe high-level programming language to untyped machine code is possible. They achieved this by enhancing the low-level machine model with a fine-grained program counter-based memory access control mechanism inspired by existing systems [14,15,17,20,21] and recent industrial prototypes [16]. One limitation of the work of Agten *et al.* is that it only considers a toy high-level language. The main contribution of this paper is showing how essential programming language features can be securely compiled to the same low-level machine model of Agten *et al.* The adopted low-level model is similar to a modern processor, so the compilation scheme handles subtleties such as flags and registers that an implementation would have to face. More precisely, this paper makes the following contributions:

- a secure compilation scheme from a model object-oriented language with dynamic memory allocation, cross-package inheritance, exceptions and inner classes to low-level, untyped machine code;
- the outline of a formal proof of full abstraction for this compilation scheme;
- measurements of the run-time overhead introduced by the compilation scheme.

The paper is organised as follows. Section 2 introduces background notions. Section 3 presents a secure compilation scheme for a language with dynamic memory allocation, cross-package inheritance, exceptions and inner classes. Section 4 outlines the proof of full abstraction of the compilation scheme. Section 5 presents benchmarks of the overhead introduced by the compilation scheme. Section 6 discusses limitations of the compilation scheme. Section 7 presents related work. Section 8 discusses future work and concludes.

2 Background

This section describes the low-level protection mechanism and the secure compilation scheme of Agten *et al.* [3], which is the starting point of this paper. Then the high-level language targeted by the compilation scheme is presented.

2.1 Low-level Model

To model a realistic compilation scheme, the targeted low-level language should be close to what is used by modern processors. For this reason this paper adopts 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 [3].

In order to support full abstraction of the compilation scheme, the low-level language is enhanced with a protection mechanism: a fine-grained, program counter-based memory access control mechanism inspired by existing systems [14,15,17,20,21] and recent industrial prototypes [16]. We review this addition from the work of Agten *et al.* [3] and Strackx and Piessens [21]. This mechanism assumes that the memory is logically divided 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 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 table below summarises the access control model enforced by the protection mechanism.

From\ To	Protected			Unprotected
	Entry Point	Code	Data	
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.

Details of the low-level language are omitted for space reasons, the reader can find them in the appendices.

2.2 A Secure Compiler for a Simple Language

Agten *et al.* [3] presented a secure (fully abstract) compilation scheme for a simple object based language. In an effort to be self-contained, this paper summarises their key points.

General notions. In the work of Agten *et al.*, programs consist of a single object with fields and methods declarations which are compiled to the protected memory partition. Compiled programs must be indistinguishable from the size

point of view, thus a constant amount of space is reserved for each program, independent of its implementation. All methods and fields are sorted alphabetically. Thus equivalent compiled programs cannot be distinguished based on the ordering of low-level method calls. Methods and fields are given a unique index, starting from 0, based on their order of occurrence. Those indexes serve as the offset used to access methods and fields. Parameters and local variables are also given method-local indexes to be used as above.

Registers $\mathtt{r}_0$ to $\mathtt{r}_3$ are used as working registers for low-level instructions and registers $\mathtt{r}_4$ to $\mathtt{r}_{11}$ are used for parameters. The call stack is split into a protected and an unprotected part, the former is allocated in the protected memory partition. A shadow stack pointer that points to the base of the protected stack is introduced to implement stack switches. When entering the protected memory, the protected stack is set as the active one; when leaving it, the unprotected stack is set to be the active one. To prevent tampering with the control flow, the base of the protected stack points to a procedure that writes 0 in $\mathtt{r}_0$ and `halt`s. For each method, a prologue and an epilogue are appended to the method body. They allocate and deallocate activation records on the secure stack, The program counter is initialised to a given address in unprotected memory.

Entry points. For each method, an entry point in protected memory is created. Additionally, in order to enable returnbacks (returns from callbacks, which are calls to external code), a returnback entry point is created. Entry points act as proxies to the actual method implementations and are extended with security routines and checks.

These security routines reset unused registers and flags when leaving the protected memory to prevent them conveying unwanted information. For example, a callback to a function with two arguments resets all registers but $\mathtt{r}_4$ and $\mathtt{r}_5$ since they are the only ones that carry desired information. Checks are made to ensure that primitive-typed parameters have the right byte representation, e.g. `Unit`-typed parameters must have value 0, the chosen value of `Unit` type.

2.3 High-level Language

The high-level language targeted by this paper is Jeffrey and Rathke's Java Jr. [11]. Java Jr. is a strongly-typed, single-threaded, component-based, object-oriented language that enforces `private` fields and `public` methods. Java Jr. supports all the basic constructs one expects from a modern programming language, including dynamic memory allocation. A program in Java Jr., called a component, is a collection of sealed packages that communicate via interfaces and public objects. Java Jr. enforces a partition of packages into *import* and *export* ones. Import packages are analogous to the `.h` header file of a C program; they define *interfaces* and *externs*, which are references to externally defined objects. Export packages define *classes* and *objects*; they provide an implementation of an import package. Listing 1.1 illustrates the package system of Java Jr.

Listing 1.1 contains two package declarations: `PI` is an import package and `PE` is an export package implementing `PI`. Object `extAccount` allocated in `PE` provides an implementation for the `extern` with the same name defined in `PI`.

```
1  package PI;
2    interface Account {
3      public createAccount() : Foo;
4      public getBalance() : Int;
5    }
6    extern extAccount : Account;
7
8  package PE;
9    class AccountClass implements PI.Account {
10     AccountClass() { counter = 0; }
11     public createAccount() : Account { return new PE.AccountClass(); }
12     public getBalance() : Int { return counter; }
13     private counter : Int;
14   }
15   object extAccount : AccountClass;
```

Listing 1.1. Example of the package system of Java Jr.

In Java Jr., primitive values, types and operations on them are assumed to be provided by a `System` package, whose name is omitted for the sake of brevity. The only primitive type is `Unit`, inhabited by `unit`. Since the focus of this paper is security, we write access modifiers for methods and fields even though the syntax of Java Jr. does not require them.

The security mechanism of Java Jr. is given by `private` fields. In Java Jr., 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, since classes that implement those interfaces are unknown. Additionally, since constructors are not exposed in interfaces, cross-package object allocation must be through factory methods. For example, the name of class `AccountClass` from Listing 1.1 is not visible from outside package `PE`, thus expressions of the form `new PE.AccountClass()` cannot be written outside `PE`.

Java Jr. was chosen since it provides a clear notion of encapsulation for a high-level component, which makes for simpler reasoning about the secure compilation scheme. This allows us to pinpoint what the key insights are to achieve secure (fully abstract) compilation, so that they can be used when the language is extended with cross-package inheritance, exceptions and inner classes.

3 Secure Compilation of Java Jr.

After a series of examples describing possible attacks on a naïve compilation scheme, this section describes what is needed in order to provide a secure compiler for Java Jr., starting from the secure compiler described in Section 2.2, and extend it to support cross-package inheritance, exceptions and inner classes.

The following examples use some standard assumptions about how objects are compiled [6]. When an object is allocated, a word is reserved to indicate its

class, which is used to dynamically dispatch methods. Fields are accessed via offsets and methods are dispatched based on offsets.

Example 1 (Type of the current object) *Suppose the compiled program includes two classes:* `Pair` *and* `Caesar`*. Class* `Pair` *implements pairs of* `Int`*eger values with two fields* `first` *and* `second`*, with getters and setters for them, method* `getFirst()` *returns the value of field* `first`*. Class* `Caesar` *implements a caesar cypher. It has a single* `Int`*eger field* `key` *and a method* `encrypt(v:Int)` *that returns value* `v` *encrypted with* `key`*. The* `key` *of the* `Caesar` *cypher is not accessible outside the class (i.e. it is private).*

The `key` *cannot be leaked at the high level, since high-level programs are strongly typed, but it can be leaked to low-level programs. A low-level, external program can perform a call to method* `getFirst()` *on an object of type* `Caesar`*; this will return the* `key` *field, since fields are accessed by offset. As low-level code is untyped, nothing prevents this attack from happening.*

Example 2 (Type of the arguments) *Similarly to Example 1, arguments of methods can be exploited in order to mount a low-level attack. Extend the program of Example 1 with another class* `ProxyPair` *with a method* `takeFirst(v:Pair)` *that returns* `getFirst()` *on the* `Pair` *object* `v`*. At the high level, this code gives rise to no attacks. At the low level, this code can be used to mount the following attack: if an object of type* `Caesar` *is passed as argument to method* `takeFirst( )`*, the code will leak the* `key`*.*

Example 3 (Leakage of object references) *Object references at the low-level are the address where objects are allocated. The attacker can 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 also 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 Example 1 and 2. From a technical point of view this means that leaking object addresses and accepting guessed addresses breaks full abstraction of the compilation scheme.*

3.1 A Secure Compiler for Java Jr.

Before proposing countermeasures to the attacks just listed, this section lists the modifications to the scheme of Section 2.2 that are needed in order to support compilation of Java Jr. and, more generally, of object-oriented programs.

Compilation of OO Languages. Fig. 1 shows a graphical representation of the protected memory section which is generated when securely compiling a Java Jr. component. Only a single protected memory section is needed, and all classes, objects and methods defined in the component are placed there. The protected code section contains entry points, described in Section 3.1 below, method body implementations, the procedure for object allocation and the dynamic dispatch. The protected data section contains the v-tables, support data

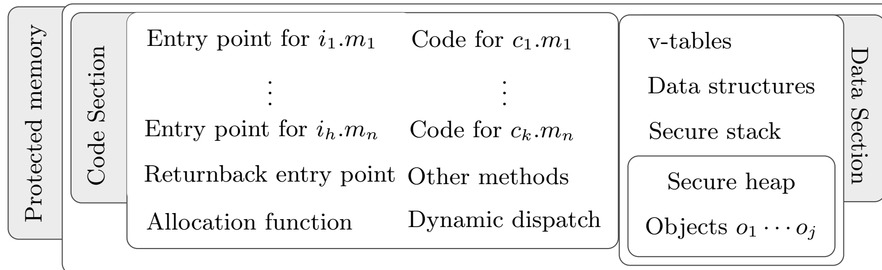

Fig. 1. Graphical representation of a compiled program.

structures, a secure stack and a secure heap. The v-tables are data structures used to perform the dynamic dispatch of method calls; they associate the address of the method to be executed on an object based on its type and the method name. Data structures are defined in Section 3.1 and 3.3.

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 on methods defined in the DIP. Component code is assumed not to implement interfaces defined in the DIP, while external code which provides an implementation for the DIP can also implement interfaces defined in the component. Assuming the calling convention with the outer world is known, dynamic dispatch can easily take care of external objects whose classes implement interfaces defined in the component. Method call implementation adopted by external code is more complex since function calls must jump to the correct entry point, but it still can be achieved for example using object wrappers.

Finally, register $\mathtt{r_4}$ is used to identify the current object (`this`) in a method call at the low level. Before a callback, $\mathtt{r_4}$ is stored in the secure stack so as to be able to restore `this` to the right value once the callback returns.

Securing the Compilation. Following are the countermeasures added to withstand the attacks described in the Examples above. Since the countermeasure to Example 3 affects the others, it is presented first.

Object identity. To mask low-level object identities, a data structure $\mathcal{O}$ is added to the data structures of Fig. 1. It is a map between low-level object identities that have been passed to external code and natural numbers. 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. As the only objects in the data structures are the ones the attacker knows, it cannot guess object identities.

Entry points. To support programming to an interface, the compilation scheme creates *method entry points* in protected memory for all interface-declared methods. A single *returnback entry point* for returning after a callback is also needed. Table 1 describes the code executed at those points. Both entry points are logically divided in two parts. The first part performs the checks described in the previous paragraph 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; call this the *exit point* for method entry points and *re-entry point* for the returnback entry point. 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). 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. If any check fails, all registers and flags are cleared and the execution `halt`s. Resetting flags and registers and `Unit`-typed value checks are as in Section 2.2. Dynamic typecheck involves checking primitive-typed values. These are needed for all primitive types inhabited by a finite number of values, such as `Unit` and `Bool`. For example, `bool`-typed parameters must have either value 1 or 0, which correspond to the high-level values `true` or `false` [7].

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

Table 1. Pseudo code executed at entry points. Loading means that a value is retrieved from the memory, push and pop are operations on the secure stack.

Insights. Following are the insights that we gained from developing a secure compilation scheme for Java Jr.; they will be useful when extending the language in the following sections.

- Internal objects that are passed to external code must be remembered; their address must be masked.
- Strong typing of methods must be enforced with additional runtime checks.

– The low-level code must not introduce additional functionality (low-level functions in entry points) that is not available at the high level.

3.2 Secure Compilation of Cross-package Inheritance

Cross-package inheritance arises whenever class `D` from an export package `PSUB` `extends` class `C` from a different export package `PSUP`, as in Listing 1.2. Cross-package inheritance is not provided by Java Jr., as it would break the main result proven in the Java Jr. paper [11]. In order to allow cross-package inheritance, 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 `D` can optionally override methods of the super class `C`, as is the case with method `m()`. Within those methods, calls to `super` can be used in order to call method `m()` of the super class `C`. Alternatively, if a method is not overridden, such as method `z()`, calling d`.z()` on an object d of type `D` executes method `z()` defined in the super class `C`.

```
1 package PSUP;
2  class C {            // called the super class
3    public m():Int { ··· }
4    public z(): Int { ··· }
5  }
6 package PSUB;
7  class D extends PSUP.C {    // called the sub class
8    public m():Int{ super.m(); ··· }
9  }
```

Listing 1.2. Example of cross-package inheritance.

If the normal compilation scheme were followed, at the low-level d is allocated to a single memory area where fields from subobjects `C` and `D` are both allocated. Example 4 highlights the problems that arise in this setting.

Example 4 (Allocation of d) *Consider the case when `C` is protected and `D` is not. If d is allocated outside the protected memory partition, private fields of the `C` subobject become accessible to external code. If d is allocated inside the protected memory partition, two options arise. The first one is placing untrusted methods of `D` in the protected memory partition, violating the security of the compilation scheme. Otherwise, if methods of `D` are placed in the unprotected memory partition, they cannot access `D`'s fields via offset. Getters and setters for fields of `D` could be exposed through entry points, but this would violate full abstraction, as those methods are not available at the high level.*

The problems just presented above also arise when `C` is not protected but `D` is, thus compilation of cross-package inheritance cannot be achieved normally.

To overcome these difficulties, when d is allocated, it is split in two subobjects: dc, with fields of class `C`, and dd, with fields of class `D`; the object identity of d is dd [22].

Consider firstly the case when `C` is protected and `D` is not. External code needs to compile the expression d `= new D()` so that it calls `new C()` to create object dc in the protected memory section. External code must then save the resulting identifier for dc to perform `super` calls, since they are translated as method calls. The additional checks inserted at entry points presented in Section 3.1 ensure that `super` calls are always well-typed.

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

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 [1]. In fact, low level external code is given the functionality to call dc`.m()`, which is not explicitly possible in the high-level language. However, d`.super.m()` is an implicit call to the `m()` method of `C`, functionality that the high-level language already has. Handling of cross-package inheritance does not add functionality at the low level, so it does not break full abstraction of the compilation scheme.

3.3 Exceptions

Secure compilation of languages supporting exceptions must handle the difficulties that result from the modification of the flow of execution of a program.

```
1  package P1;
2    class G {
3      public m():Void{
4        try{ new P2.H().e(); } catch (e : P3.MyException){ // handle e   } }
5    }
6
7  package P2;
8    class H {
9      public e():Void throws P3.MyException { throw new P3.MyException(); }
10   }
11
12 package P3;
13   class MyException implements Throwable {···}
```

Listing 1.3. Example of exceptions usage.

Exception handling can be securely 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. In Listing 1.3, the `catch` block of method `m()` in class `G` defines a handler for exceptions of type `MyException`. When the activation record for `m()` is allocated, the handler is registered. When an exception of type `MyException` is thrown, the stack is traversed to find the closest handler for exceptions of type `MyException`. As activation records are traversed and a handler is not found, those records are popped from the stack.

In the context of secure compilation, exception handlers are compiled in the usual manner. In order to implement throwing an exception in secure code that

is caught in insecure code (or vice versa), throwing is compiled as callbacks (or calls). Thus two additional entry points are created: the *throw entry point* and the *throwback entry point*. These entry point 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, the exception is forwarded to the external code through the throwback entry point. In order to prevent exploits similar to those of Example 2, the throwback entry point must remember internally allocated exceptions that are thrown to external code. So, a data structure $\mathcal{E}$, similar to $\mathcal{O}$, is created to register leaked exceptions. This prevents external code from passing a fake object identity to the secure exception handler in place of the object identity of an exception, effectively throwing a non existent exception. External code can implement a wrapper around the exception object identity in order to be able to associate it to its type and then be able to recognise the type of the exception in the handler.

Fig. 2 presents a graphical overview of how exceptions are handled normally (on the left) and in the presented compilation scheme (on the right). Lower case

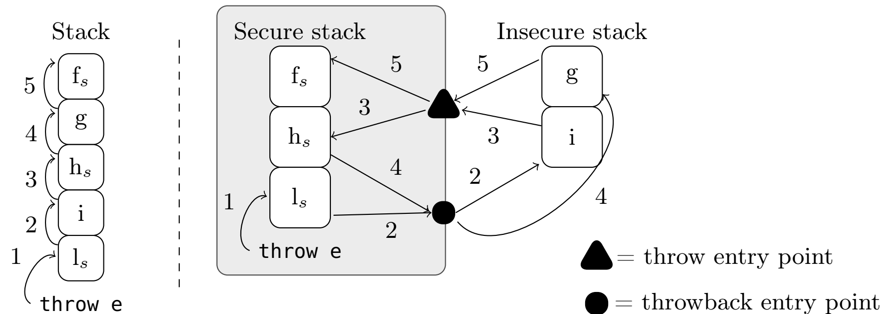

Fig. 2. Comparison of ways to handle exceptions.

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 call in two parts.

Full abstraction of the compilation scheme is preserved since the low-level is not extended with functionality that the high-level lacks. Only exceptions of existing type can be thrown and handling exceptions follows the normal course of the stack. The external code could replace an exception with a fake one, but this is equivalent to the high-level language functionality to catching an exception and throwing another one. Thus the low-level is not granted additional functionality.

3.4 Secure Compilation of Inner Classes

Inner classes are classes that are defined inside another class, as in Listing 1.4. Inner classes have access to private fields of the class they are defined within.

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

Listing 1.4. Example of an inner class.

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, a getter and a setter for each of its private fields are created. In the case of Listing 1.4, 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 [1], 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 other than 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.

4 Full Abstraction of the Compilation Scheme

This section presents an outline of the proof of full abstraction of the compilation scheme of Section 3. As mentioned in Section 1, a fully abstract compilation scheme preserves and reflects contextual equivalence of high- and low-level programs. This paper does not argue about the choice of contextual equivalence for modelling security properties. This is treated in other relevant literature [1,2,3,4,7,9,12].

Informally speaking, two programs C_1 and C_2 are contextually equivalent if they behave the same for all possible evaluation contexts they interact with. An evaluation context $\mathbb{C}$ can be thought of as a larger program with a hole. If the hole is filled either with C_1 or C_2, the behaviour of the whole program does not vary. Formally, contextual equivalence is defined as: $C_1 \simeq C_2 \triangleq \forall \mathbb{C}.\ \mathbb{C}[C_1]\Uparrow \iff \mathbb{C}[C_2]\Uparrow$, where $\Uparrow$ denotes divergence [19].

Denote the result of compiling a component C as $C^{\downarrow}$. Full abstraction of the compilation scheme is formally expressed as: $C_1 \simeq C_2 \iff C_1^{\downarrow} \simeq C_2^{\downarrow}$. The co-implication 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 low-level programs that behave as the corresponding

source programs. This is what most compilers achieve, at times even certifying the result [5,13]; we are not interested in this direction. This is thus assumed, the consequences of this assumption are made explicit (Assumption 1 below). The direction $C_1 \simeq C_2 \Rightarrow C_1^{\downarrow} \simeq C_2^{\downarrow}$ states that high-level properties are preserved through compilation to the low level. Proving this direction requires reasoning about contexts, which is notoriously difficult [4]. This is even more so in this setting, where low-level contexts are memories lacking any inductive structure. To avoid working with contexts, we equip the low-level language with a trace semantics that is equivalent to its operational semantics [18] (Proposition 1 below) and prove the contrapositive: $\mathsf{Traces}_{\mathsf{L}}(C_1^{\downarrow}) \neq \mathsf{Traces}_{\mathsf{L}}(C_2^{\downarrow}) \Rightarrow C_1 \not\simeq C_2$. This proof is based on an algorithm that creates a high-level component, a "witness" that differentiates C_1 from C_2, given that they have different low-level traces $\overline{\alpha_1}$ and $\overline{\alpha_2}$. This proof strategy is known [3,10], its complexity resides in handling features of the high-level language such as typing or dynamic memory allocation.

This proof, as well as the formalisation of Java Jr. and the assembly language, are omitted for space constraints. The interested reader can find them in the appendices. To further support the validity of this proof, the algorithm has been implemented in Scala, and it outputs Java components that adhere to the Java Jr. formalisation.[1]

Assumption 1 (Compiler preserves behaviour) *The compiler is assumed to output low-level programs that behave as the corresponding input program. Thus a high-level expression is translated into a list of low-level instructions that preserve the behaviour. By this, we mean that the following properties hold:*

- $C_1^{\downarrow} \simeq C_2^{\downarrow} \Rightarrow C_1 \simeq C_2$.
- *There exists an equivalence relation between high-level states and low-level states, such that:*
  - *The initial high- and low-level states are equivalent.*
  - *Given two equivalent states and two corresponding internal transitions, the states these transitions lead to are equivalent. Moreover, given two equivalent states and two equivalent actions, the states these transitions lead to are equivalent.*

Proposition 1 (Trace semantics is equivalent to operational semantics [18]). *For any two low-level components $C_1^{\downarrow}$ and $C_2^{\downarrow}$ obtained from compiling Java Jr. components C_1 and C_2 with the secure compilation scheme, we have that:* $\mathsf{Traces}_{\mathsf{L}}(C_1^{\downarrow}) = \mathsf{Traces}_{\mathsf{L}}(C_2^{\downarrow}) \iff C_1^{\downarrow} \simeq C_2^{\downarrow}$.

Theorem 1 (Differentiation of components). *Any two high-level components C_1 and C_2 that exhibit two different low-level traces $\overline{\alpha_1}$ and $\overline{\alpha_2}$ are not contextually equivalent. Formally:* $\mathsf{Traces}_{\mathsf{L}}(C_1^{\downarrow}) \neq \mathsf{Traces}_{\mathsf{L}}(C_2^{\downarrow}) \Rightarrow C_1 \not\simeq C_2$.

Theorem 2 (Full abstraction of the compilation scheme). *For any two high-level components C_1 and C_2, we have (assuming there is no overflow of the secure stack or of the secure heap):* $C_1 \simeq C_2 \iff C_1^{\downarrow} \simeq C_2^{\downarrow}$.

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

5 Benchmarks

This section presents benchmarking of the overhead of the secure compiler, which is proportional to the amount of boundaries crossing.

As a target low-level architecture we chose Fides [3,21]. The Fides architecture implements precisely the protection mechanism described in Section 2.1 in a very reduced TCB: ∼7000 lines of code. Fides consists of a hypervisor that runs two virtual machines: one handles the secure memory partition and one handles the other [21]. One consequence is that switching between the two virtual machines of Fides (performing calls and callbacks) is a costly operation.

For the benchmarks, we implemented a secure runtime in C. The secure runtime adds the checks presented in Section 3 to calls, callbacks (both with different number of parameters, ranging from one to eight), returns and returnbacks. These operations are executed on stub objects. A stub objects is a data structure that models the low-level representation of objects; it has an `integer` field that indicates the class of the object followed by the fields of the object. The secure runtime also contains the data structure $\mathcal{O}$ and functions that mask object references through it. Each operation was tested 1000 times on a MacBook Pro with a 2.3 GHz Intel Core i5 processor and 4GB 1333MHz DDR3 RAM. The overhead introduced for each operation ranged from 0.09% to 7.89%, averaging a 3.25% overhead. Details of the measurements are in the appendices.

6 Limitations

This section presents limitations of the compilation scheme of Section 3 and discusses garbage collection when part of the program is compiled securely.

Like many model languages [2,9], Java Jr. 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. Let us now informally discuss how garbage collection can be achieved in concert with a secure compiler.

Garbage collection is a runtime addition that handles whole programs. Firstly, assume that the external code is well-behaved and it does not disrupt the functionality of the garbage collector, such as by introducing fake pointers. To perform garbage collection when part of the whole program is securely compiled, a part of the garbage collector must be trusted and allocated in the protected memory partition so that it can access $\mathcal{O}$. In this way the garbage collector can traverse the whole object graph and identify the location of a reference that is an index of $\mathcal{O}$.

Assume now that external code can disrupt the functionality of the garbage collector. The classical notion of garbage collection becomes void. In this setting the securely compiled component can be extended with a secure memory manager in charge of the secure memory partition. Here, an arguable safe methodology is to not deallocate a reference that is passed from the secure component

to external code, a fact that creates problems when the allocated object is large or when many objects are passed out. In order to provide a solution to part of the problem, the compiler can introduce leasing [8]; this gives objects that are leaked a lifetime duration which, upon expiration, causes object deallocation. Alternatively, the caretaker pattern can be introduced. Instead of leaking an object reference o, the reference is wrapped in a proxy p (the caretaker) and the reference to p is leaked. In addition to method proxies for methods of o, p has a method to set the reference to o to `null`, allowing the secure memory manager to free o's memory. The problem that arises now is a breach in full abstraction: the caretaker pattern must be lifted to the high level to preserve full abstraction.

7 Related Work

This paper extends the work of Agten *et al.* [3], where the same result is achieved, but for a simpler, object-based, high-level language. This work adopts an object-oriented language with dynamic object allocation, cross-package inheritance, exceptions and inner classes, which makes the result significantly harder to achieve.

Secure compilation through full abstraction was pioneered by Abadi [1], 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 [12] 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 [2] and subsequently by Jagadeesan *et al.* [9] 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.* [7] 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 [4] prove full abstraction of a continuation-passing style translation from simply-typed λ-calculus to System F. In both works, the low-level language is typed and more high-level than ours. The checks introduced by our compilation scheme seem simpler than the checks of Fournet *et al.*

A large amount of work on secure compilation applies to unsafe languages such as C, as surveyed by Younan *et al.* [23]. That research is devoted to strengthening the run-time of C and not on fully abstract compilation.

A different area of research provides security architectures with fine-grained low-level protection mechanisms. Different security architectures with access control mechanisms comparable to ours have been developed in recent years: TrustVisor [14], Flicker [15], Nizza [20], SPMs [17,21] and the Intel SGX [16]. The existence of industrial prototypes underlines the feasibility of this approach to bringing efficient, secure, low-level memory access control in commodity hardware. No results comparable to ours were proven for these systems.

8 Conclusion and Future Work

This paper presented a fully abstract compilation scheme for a strongly-typed, single-threaded, component-based, object-oriented programming language with dynamic memory allocation, exceptions, cross-package inheritance and inner classes to untyped machine code enhanced with a low-level protection mechanism. Full abstraction of the compilation scheme is proven correct, guaranteeing preservation and reflection of contextual equivalence between high-level components and their compiled counterparts. From the security perspective this ensures that low-level attackers are restricted to the same capabilities high-level attackers have. To the best of our knowledge, this is the first result of its kind for such an expressive high-level language and such a powerful low-level one.

Future work includes extending the results to a language with more real-world programming language features such as concurrency and distribution.

References

1. Martín Abadi. Protection in programming-language translations. In *Secure Internet programming*, pages 19–34. Springer-Verlag, London, UK, 1999.
2. Martín Abadi and Gordon Plotkin. On protection by layout randomization. In *CSF '10*, pages 337–351. IEEE, 2010.
3. Pieter Agten, Raoul Strackx, Bart Jacobs, and Frank Piessens. Secure compilation to modern processors. In *CSF '12*, pages 171–185. IEEE, 2012.
4. Amal Ahmed and Matthias Blume. An equivalence-preserving CPS translation via multi-language semantics. *SIGPLAN Not.*, 46(9):431–444, September 2011.
5. Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. *SIGPLAN Not.*, 42(6):54–65, June 2007.
6. Roland Ducournau. Implementing statically typed object-oriented programming languages. *ACM Comput. Surv.*, 43(3):18:1–18:48, April 2011.
7. Cedric Fournet, Nikhil Swamy, Juan Chen, Pierre-Evariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. Fully abstract compilation to JavaScript. In *POPL '13*, pages 371–384, New York, NY, USA, 2013. ACM.
8. C. Gray and D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. *SIGOPS Oper. Syst. Rev.*, 23(5):202–210, 1989.
9. Radha Jagadeesan, Corin Pitcher, Julian Rathke, and James Riely. Local memory via layout randomization. In *CSF '11*, pages 161–174. IEEE, 2011.
10. Alan Jeffrey and Julian Rathke. A fully abstract may testing semantics for concurrent objects. *Theor. Comput. Sci.*, 338(1-3):17–63, June 2005.
11. Alan Jeffrey and Julian Rathke. Java Jr.: fully abstract trace semantics for a core Java language. In *ESOP'05*, volume 3444 of *LNCS*, pages 423–438. Springer, 2005.
12. Andrew Kennedy. Securing the .NET programming model. *Theor. Comput. Sci.*, 364(3):311–317, November 2006.
13. Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4):363–446, December 2009.
14. Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. Trustvisor: Efficient TCB reduction and attestation. In *SP '10*, pages 143–158. IEEE, 2010.

15. Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: an execution infrastructure for TCB minimization. *SIGOPS Oper. Syst. Rev.*, 42(4):315–328, April 2008.
16. Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP '13*, pages 10:1–10:1. ACM, 2013.
17. Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. 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, 2013.
18. Marco Patrignani and Dave Clarke. Fully abstract trace semantics for low-level isolation mechanisms. Under submission, 2013.
19. Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.
20. Lenin Singaravelu, Calton Pu, Hermann Härtig, and Christian Helmuth. Reducing TCB complexity for security-sensitive applications: three case studies. *SIGOPS Oper. Syst. Rev.*, 40(4):161–174, April 2006.
21. Raoul Strackx and Frank Piessens. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *CCS '12*, pages 2–13. ACM Press, October 2012.
22. Marko van Dooren, Dave Clarke, and Bart Jacobs. Subobject-oriented programming. In *Formal Methods for Objects and Components*. To appear, 2013.
23. Yves Younan, Wouter Joosen, and Frank Piessens. Runtime countermeasures for code injection attacks against C and C++ programs. *ACM Computing Surveys*, 44(3):17:1–17:28, 2012.

Introduction to Appendixes

Section A presents contextual equivalence as a good candidate to model security properties enforcement. Section B and Section C present the details of the formalisation of both Java Jr. and the low-level language. Section D presents examples for the algorithm mentioned in Section 4. Section E presents a transliteration of the pseudo code of the algorithm of Section 4. Section F presents the proofs of Section 4. Finally, Section G presents details of the benchmarking measurements.

A Contextual Equivalence: a Security Perspective

This section briefly motivates contextual equivalence as a good candidate for security policies enforcement. This notion is widely accepted in the field, it is only included in order to familiarise the reader with these concepts [1,2,3,4,7,9,12].

In Java Jr., 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*, denoted as $C_1 \simeq C_2$. Formally, for all contexts $\mathbb{C}$ with a hole, contextual equivalence is defined as follows: $C_1 \simeq C_2 \triangleq \forall \mathbb{C}. \mathbb{C}[C_1]\Uparrow \iff \mathbb{C}[C_2]\Uparrow$, where $\Uparrow$ denotes divergence [19]. Contexts in Java Jr. are components [11].

Contextual equivalence can be adopted to state security properties such as confidentiality, integrity and invariant definition, as in Fig. 3 [2,3]. Classes are annotated with subscripts for identification but their names are meant to be the same. Class $\mathtt{C}_1$ and $\mathtt{C}_2$ in Fig. 3 differ in the value stored in `secret` after a

```
1  package p;
2  class C1 {
3    secret, min, max:Int=0;
4
5    public m1():Int {
6      secret = 0;
7      return 0;
8    }
9    public m2(arg:D):Int {
10     secret = 0;
11     arg.cb();
12     if (secret==0) { return 0; }
13     return 1;
14   }
15   public m3():Int {
16     if (min ≤ max) { return 0; }
17     return 1;
18   }
19 }
```

```
1  package p;
2  class C2 {
3    secret, min, max:Int=0;
4
5    public m1():Int {
6      secret = 1;
7      return 0;
8    }
9    public m2(arg:D):Int {
10     secret = 0;
11     arg.cb();
12     return 0;
13
14   }
15   public m3():Int {
16     return 0;
17
18   }
19 }
```

Fig. 3. Context equivalence used for security properties enforcement.

call to `m1`. 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, $\texttt{C}_1$ checks whether changes to `secret` have been made during the call to `arg.cb( )` in method `m2`. If $\texttt{C}_1$ and $\texttt{C}_2$ are contextually equivalent then the call to `arg.cb( )` does not modify the value of `secret`, this is an *integrity* property. Finally, when `min`>`max`, method `m3` of $\texttt{C}_1$ will return 1. If $\texttt{C}_1$ and $\texttt{C}_2$ are contextually equivalent then the invariant `min` $\leq$ `max` is never violated. This is an *invariant definition* property.

From the high-level language perspective, context equivalence holds for the presented examples. However, when these examples are run on a physical machine, they are compiled to machine code. As the machine code is not strongly typed and it allows jumps to all addresses in memory, an attacker with machine code injection privileges can violate the secrecy properties of these examples [23].

B High-Level Language

This section formally presents the high-level language used herein: Java Jr. [11] alongside its syntax, static, dynamic and trace semantics. Most of the concepts are taken from the work of Jeffrey and Rathke [11], with some corrections as to fill the missing bits of the formalisation.

B.1 Syntax

Assume the presence of an infinite set of package names P_n ranged over by p, q, class and interfaces names C_n ranged over by c, i, t, u, object names O_n ranged over by o, field names F_n ranged over by f, g, h, variable names V_n ranged over by x, y, z and method names M_n ranged over by m. Names in a package are always pairwise distinct. Integers, unit and other ground types are considered to be implemented in a `System` package, alongside operations on them. The root of the class hierarchy is indicated with `Obj`. Values of the language,denoted with v, are object identifiers $p.o$ and ground-typed values such as `Unit` and natural numbers. The syntax of Java Jr. is presented in Fig. 4.

B.2 Static Semantics

The static semantics defines typing relations based on the judgments of Fig. 5. Adopt Γ as the standard type environment that binds variables to types, where variables cannot be repeated. Additionally, the component C is added to the typing environment to act as a reference to the standard class table. To preserve the encapsulation principle given by packages, types are annotated with package names; thus an expression does not simply have type t but type t in p.

Fig. 6 and Fig. 7 present the typing rules as from the original Java Jr. work [11]. They are mostly standard, except for few modifications. Modifications were made to rules Declaration-class, Declaration-object, Declaration-interface and Method-Types to ensure the "programming to an interface" paradigm is enforced.

$$
\begin{array}{rll}
C & ::= \overline{P} & \text{components} \\
P & ::= \{\texttt{package}\ p; \overline{D}\} & \text{packages} \\
D & ::= \texttt{class}\ c\ \texttt{extends}\ t & \text{declarations} \\
& \qquad \texttt{implements}\ \bar{t}\ \{K\ \overline{F_t}\ \overline{M}\} & \\
& \quad \mid \texttt{object}\ o : t\ \texttt{implements}\ \bar{t}\ \{\overline{F}\} & \\
& \quad \mid \texttt{interface}\ i\ \texttt{extends}\ \bar{t}\ \{\overline{M_t}\} & \\
& \quad \mid \texttt{extern}\ o : \bar{t}; & \\
K & ::= c(\overline{f} : \bar{t}, \overline{h} : \overline{u})\ \{ & \text{constructors} \\
& \quad \texttt{super}(\overline{f}); \texttt{this}.\overline{g} = \overline{h}\} & \\
F & ::= f = v; & \text{fields} \\
F_t & ::= f : t; & \text{field types} \\
M & ::= \texttt{public}\ m(\overline{x} : \bar{t}) : t\ \{\texttt{return}\ E;\} & \text{methods} \\
M_t & ::= m(\overline{x} : \bar{t}) : t; & \text{method types} \\
E & ::= v \mid x \mid E.f \mid E.f = E \mid E.m(\overline{E}) & \text{expressions} \\
& \quad \mid \texttt{new}\ t(\overline{E}) \mid (E == E ? E : E) \mid E; E & \\
& \quad \mid E\ \texttt{in}\ p & \\
t & ::= p.c \mid p.i \mid p.c\ \texttt{in}\ p \mid p.c\ \texttt{in}\ * \mid \texttt{Obj} & \text{types}
\end{array}
$$

Fig. 4. Syntax of Java Jr.

Rules Expr-concat and Scope-all have been devised by the authors as they were not presented in the original paper.

Type soundness The type system enjoys the progress and preservation properties as proven in the original Java Jr. work [11].

B.3 Dynamic Semantics

The dynamic semantics is given in terms of a relation $(C \vdash E) \rightarrow (C' \vdash E')$ that models the evolution of component C executing expression E to C' executing E'. The expression that is executed is immersed in an evaluation context $\mathbb{E}$, which models the environment in which the evaluation takes place. The syntax of an evaluation context is:

$$
\begin{aligned}
\mathbb{E} ::= & \ [\,] \mid \mathbb{E}.m(\overline{E}) \mid E.m(\overline{v}, \mathbb{E}, \overline{E}) \mid \mathbb{E}.f \mid \mathbb{E}.f = E \\
& \mid v.f = \mathbb{E} \mid \texttt{new}\ t(\overline{v}, \mathbb{E}, \overline{E}) \mid (\mathbb{E} == E ? E_T : E_F) \\
& \mid (v == \mathbb{E} ? E_T : E_F) \mid \mathbb{E}; E \mid \mathbb{E}\ \texttt{in}\ p
\end{aligned}
$$

Rules for reductions of the form $(C \vdash E) \rightarrow (C' \vdash E')$ are presented in Fig. 8.

$\vdash C : \mathsf{cmp}$	well-typed component C
$C \vdash P : \mathsf{pkg}$	well-typed package P
$C \vdash D : \mathsf{dec}\ \mathsf{in}\ p$	well-typed declaration D in package p
$C \vdash t : \mathsf{type}\ \mathsf{in}\ p$	t is a valid type in package p
$C \vdash c : \mathsf{cls}\ \mathsf{in}\ p$	c is a valid class in package p
$C \vdash i : \mathsf{itf}\ \mathsf{in}\ p$	i is a valid interface in package p
$C \vdash t <: t'\ \mathsf{in}\ p$	t is subtype of t' in package p
$C \vdash v : t\ \mathsf{in}\ *$	value v has type t in the whole component
$C \vdash K : \mathsf{cnstr}\ \mathsf{in}\ p$	well-typed constructor K in package p
$C \vdash M_t : \mathsf{hdr}\ \mathsf{in}\ p.i$	well-typed method header M_t of interface i in package p
$C \vdash M : \mathsf{mth}\ \mathsf{in}\ p.c$	well-typed method M of class c in package p
$C; \Gamma \vdash E : t\ \mathsf{in}\ p$	well-typed expression E in package p

Fig. 5. Typing judgments of Java Jr.

B.4 Trace Semantics

As Section A presented, full abstraction between two entities is achieved by showing contextual equivalence of the involved entities. In this case, contexts $\mathbb{C}$ can be thought as components, the expression $\mathbb{C}[C]$ denoting the merging of two components $\mathbb{C}, C$, more details can be found in [11].

The paper presenting Java Jr. [11] establishes a full abstraction result between contextual equivalence and trace semantics, denoted $\mathsf{Traces_H}(C)$, making the two notions identical. Two well-typed components that have the same trace semantics are thus also contextually equivalent. Formally: if $\vdash C_1 : \mathsf{cmp}$ and $\vdash C_2 : \mathsf{cmp}$ then $\mathsf{Traces_H}(C_1) = \mathsf{Traces_H}(C_2) \iff C_1 \simeq C_2$ Let us now define the details related to the trace semantics of Java Jr.

The trace semantics of a component C is a set of sequences of labels, actions that can be executed by C when interacting with another unknown piece of code. In this setting, labels are method calls and returns, possibly preceded by a number of binders that bind names of newly introduced objects.

Labels L that define actions are presented in Fig. 9 alongside the function for calculating free names in traces, τ is the internal silent action. Decorations ? and ! express the "direction" of the action: from the testing environment to the component under test or vice-versa.

Traces are sequences of a's: visible actions that are considered the same up to α-equivalence of newly defined names. Denote two α-equivalent traces $\overline{a_1}$, $\overline{a_2}$ as $\overline{a_1} \equiv_\alpha \overline{a_2}$. Labels of the form $\mathtt{new}(v).g$ act as binders for v in g.

Following are the downcasting rules for imported $(C + \mathtt{extern}\ v : t;)$ and exported $(C + \mathtt{object}\ v : t;)$ names. These downcasting rules define how to

$$\frac{C \equiv \overline{P} \quad C \vdash \overline{P} : \mathsf{pkg}}{\vdash C : \mathsf{cmp}}\ \text{(Programs)} \qquad \frac{C \vdash \overline{D} : \mathsf{dec\ in}\ p}{C \vdash \{\mathsf{package}\ p; \overline{D}\} : \mathsf{pkg}}\ \text{(Packages)}$$

$$\frac{\begin{array}{c} C \vdash t : \mathsf{cls\ in}\ p \quad \mathsf{name}(K) = c \quad C \vdash \overline{t} : \mathsf{itf\ in}\ * \\ C \vdash K : \mathsf{cnstr\ in}\ p \quad C \vdash \overline{M} : \mathsf{mth\ in}\ p.c \\ \forall u \in \{t, \overline{t}\}\ C.u.\mathtt{hdrs} \subseteq C.p.c.\mathtt{hdrs} \end{array}}{C \vdash \mathtt{class}\ c\ \mathtt{extends}\ t\ \mathtt{implements}\ \overline{t}\ \{K\ \overline{F_t}\ \overline{M}\} : \mathsf{dec\ in}\ p}\ \text{(Declaration-class)}$$

$$\frac{\begin{array}{c} C \vdash t : \mathsf{cls\ in}\ p \quad C \vdash t <: \overline{t}\ \mathsf{in}\ p \quad C \vdash \overline{t} : \mathsf{itf\ in}\ * \\ C.t.\mathtt{flds} = \{\overline{f} : \overline{t}'\} \quad C \vdash \overline{v} : \overline{t}'\ \mathsf{in}\ p \end{array}}{C \vdash \mathtt{object}\ o : t\ \mathtt{implements}\ \overline{t}\ \{\overline{f} = \overline{v}\} : \mathsf{dec\ in}\ p}\ \text{(Declaration-object)}$$

$$\frac{\begin{array}{c} C \vdash \overline{t} : \mathsf{itf\ in}\ * \quad C \vdash \overline{M_t} : \mathsf{hdr\ in}\ p.i \\ \forall t \in \overline{t}\ C.t.\mathtt{hdrs} \subseteq C.p.i.\mathtt{hdrs} \end{array}}{C \vdash \mathtt{interface}\ i\ \mathtt{extends}\ \overline{t}\ \{\overline{M_t}\} : \mathsf{dec\ in}\ p}\ \text{(Declaration-interface)}$$

$$\frac{\begin{array}{c} C \vdash \overline{t} : \mathsf{type\ in}\ * \\ C.\overline{t}.\mathtt{hdrs}\ \text{are compatible} \\ C.p\ \text{is an import} \end{array}}{C \vdash \mathsf{extern}\ o : \overline{t}; : \mathsf{dec\ in}\ p}\ \text{(Declaration-extern)}$$

$$\frac{C.t = \{\mathsf{package}\ p; \mathtt{class}\ c\ \mathtt{extends}\ t'\ \mathtt{implements}\ \overline{t}\ \{K\ \overline{F_t}\ \overline{M}\}\}}{C \vdash t : \mathsf{cls\ in}\ p}\ \text{(Classes)} \qquad \frac{}{C \vdash \mathtt{Obj} : \mathsf{cls\ in}\ p}\ \text{(Classes-Obj)}$$

$$\frac{C.t = \{\mathsf{package}\ p; \mathtt{interface}\ i\ \mathtt{extends}\ \overline{t}\ \{\overline{M_t}\}\}}{C \vdash t : \mathsf{itf\ in}\ p}\ \text{(Interfaces)} \qquad \frac{C \vdash t : \mathsf{cls\ in}\ p}{C \vdash t : \mathsf{type\ in}\ p}\ \text{(Types-class)}$$

$$\frac{C \vdash t : \mathsf{itf\ in}\ p}{C \vdash t : \mathsf{type\ in}\ p}\ \text{(Types-interface)} \qquad \frac{C \vdash t : \mathsf{type\ in}\ p}{C \vdash t <: t\ \mathsf{in}\ p}\ \text{(Subtype-refl)} \qquad \frac{C \vdash t <: t''\ \mathsf{in}\ p \quad C \vdash t'' <: t'\ \mathsf{in}\ p}{C \vdash t <: t'\ \mathsf{in}\ p}\ \text{(Subtype-trans)} \qquad \frac{C \vdash t : \mathsf{type\ in}\ p}{C \vdash t <: \mathtt{Obj}\ \mathsf{in}\ p}\ \text{(Subtype-obj)}$$

$$\frac{C \vdash t : \mathsf{type\ in}\ p \quad t' \in C.t.\mathsf{superTypes}}{C \vdash t <: t'\ \mathsf{in}\ p}\ \text{(Subtype-def)} \qquad \frac{C.p = \{\mathsf{package}\ p; \overline{D}\} \quad v \notin \mathsf{fn}(C \setminus p) \quad C \vdash v : t\ \mathsf{in}\ p}{C \vdash v : t\ \mathsf{in}\ *}\ \text{(Scope-all)}$$

Fig. 6. Java Jr. typing rules, part one.

update objects and externs allocated inside components.

$$\begin{array}{l} C + \mathsf{extern}\ v : t; = C \qquad \textit{if}\ C \vdash v : t\ \mathsf{in}\ * \\ C + \mathsf{extern}\ v : t; = C + \{\mathsf{package}\ p; \mathsf{extern}\ o : \overline{t}, t;\} \\ \qquad\qquad \textit{if}\ C.v = \{\mathsf{package}\ p; \mathsf{extern}\ o : \overline{t};\} \\ \qquad\qquad \textit{and}\ C.\overline{t}.\mathtt{hdrs} \cup C.t.\mathtt{hdrs}\ \text{are compatible} \end{array}$$

$$\text{(Constructors)}\quad \frac{C \vdash \overline{u} : \mathsf{type\ in}\ p \qquad C.c.\mathtt{flds} = \{\overline{g} : \overline{u}\} \qquad C.p.c.\mathtt{super.flds} = \{\overline{f}' : \overline{t}\}}{C \vdash c(\overline{f} : \overline{t}, \overline{h} : \overline{u})\{\mathtt{super}(\overline{f}); \mathtt{this}.\overline{g} = \overline{h}\} : \mathsf{cnstr\ in}\ p}$$

$$\text{(Method-Types)}\quad \frac{C \vdash t : \mathsf{itf\ in}\ * \qquad C \vdash \overline{t} : \mathsf{itf\ in}\ *}{C \vdash m(\overline{x} : \overline{t}) : t; : \mathsf{hdr\ in}\ p}$$

$$\text{(Methods)}\quad \frac{C; \overline{x} : \overline{t}, \mathtt{this} : p.c \vdash E : t\ \mathsf{in}\ p \qquad C \vdash t : \mathsf{type\ in}\ p \qquad C \vdash \overline{t} : \mathsf{type\ in}\ p}{C \vdash \mathtt{public}\ m(\overline{x} : \overline{t}) : t\ \{\mathtt{return}\ E;\} : \mathsf{mth\ in}\ p.c}$$

$$\text{(Expr-val-obj)}\quad \frac{C.v = \{\mathtt{package}\ p; \mathtt{object}\ o : t\ \mathtt{implements}\ \overline{t}\{\overline{F}\}\}}{C; \Gamma \vdash v : t\ \mathsf{in}\ p}$$

$$\text{(Expr-val-obj-itf)}\quad \frac{C.v = \{\mathtt{package}\ p; \mathtt{object}\ o : t\ \mathtt{implements}\ \overline{t}\{\overline{F}\}\} \qquad t' \in \overline{t}}{C; \Gamma \vdash v : t'\ \mathsf{in}\ p}$$

$$\text{(Expr-val-extern)}\quad \frac{C.v = \{\mathtt{package}\ p; \mathtt{extern}\ o : \overline{t};\} \qquad t \in \overline{t}}{C; \Gamma \vdash v : t\ \mathsf{in}\ p}$$

$$\text{(Expr-var)}\quad \frac{x : t \in \Gamma \qquad C \vdash t : \mathsf{type\ in}\ p}{C; \Gamma \vdash x : t\ \mathsf{in}\ p}$$

$$\text{(Expr-fld)}\quad \frac{C; \Gamma \vdash E : t'\ \mathsf{in}\ p \qquad f : t \in C.t'.\mathtt{flds}}{C; \Gamma \vdash E.f : t\ \mathsf{in}\ p}$$

$$\text{(Expr-fldup)}\quad \frac{C; \Gamma \vdash E : u\ \mathsf{in}\ p \qquad C; \Gamma \vdash E' : t\ \mathsf{in}\ p \qquad f : t \in C.u.\mathtt{flds}}{C; \Gamma \vdash E.f = E' : t\ \mathsf{in}\ p}$$

$$\text{(Expr-meth)}\quad \frac{C; \Gamma \vdash E : u\ \mathsf{in}\ p \qquad C; \Gamma \vdash \overline{E} : \overline{t}\ \mathsf{in}\ p \qquad m(\overline{x} : \overline{t}) : t \in C.u.\mathtt{hdrs}}{C; \Gamma \vdash E.m(\overline{E}) : t\ \mathsf{in}\ p}$$

$$\text{(Expr-new)}\quad \frac{C \vdash c : \mathsf{cls\ in}\ p \qquad C \vdash \overline{E} : \overline{t} : \mathsf{in}\ p \qquad C \vdash C.p.c.\mathtt{flds} : \overline{t}}{C; \Gamma \vdash \mathsf{new}\ p.c(\overline{E}) : p.c\ \mathsf{in}\ p}$$

$$\text{(Expr-if)}\quad \frac{C; \Gamma \vdash E : u\ \mathsf{in}\ p \qquad C; \Gamma \vdash E' : u\ \mathsf{in}\ p \qquad C; \Gamma \vdash E_T : t\ \mathsf{in}\ p \qquad C; \Gamma \vdash E_F : t\ \mathsf{in}\ p}{C; \Gamma \vdash (E == E' ? E_T : E_F) : t\ \mathsf{in}\ p}$$

$$\text{(Expr-concat)}\quad \frac{C; \Gamma \vdash E : u\ \mathsf{in}\ p \qquad C; \Gamma, E : u\ \mathsf{in}\ p \vdash E' : t\ \mathsf{in}\ p}{C; \Gamma \vdash E; E' : t\ \mathsf{in}\ p}$$

$$\text{(Expr-coercion)}\quad \frac{C; \Gamma \vdash E : t\ \mathsf{in}\ p \qquad C \vdash t : \mathsf{type\ in}\ q}{C; \Gamma \vdash E\ \mathsf{in}\ p : t\ \mathsf{in}\ q}$$

$$\text{(Expr-subsumption)}\quad \frac{C; \Gamma \vdash E : t\ \mathsf{in}\ p \qquad C \vdash t <: u\ \mathsf{in}\ p}{C; \Gamma \vdash E : u\ \mathsf{in}\ p}$$

Fig. 7. Java Jr. typing rules, part two.

$$\begin{aligned}
C + \mathtt{object}\ v : t; &= C \qquad \textit{if}\ C \vdash v : t\ \mathsf{in}\ * \\
C + \mathtt{object}\ v : t; &= C + \{\mathtt{package}\ p; \mathtt{object}\ o : u \\
&\qquad \mathtt{implements}\ \overline{t}, t\{\overline{F}\}\} \\
&\quad \textit{if}\ C.v = \{\mathtt{package}\ p; \mathtt{object}\ o : u \\
&\qquad \mathtt{implements}\ \overline{t}\{\overline{F}\}\} \\
&\quad \textit{and}\ C \vdash u <: t\ \mathsf{in}\ p\ \textit{and}\ \overline{t} \neq \epsilon
\end{aligned}$$

Notice that the downcasting rules allow ground-typed variables to be treated as objects and externs simply via the first rule of each downcasting.

(Eval-method)

$$\frac{C.v = \{\texttt{package}\ p; \texttt{object}\ o : t\ \texttt{implements}\ \bar{t}\{\overline{F}\}\} \quad \texttt{public}\ m(\overline{x} : \bar{t}) : t\{\texttt{return}\ E;\} \in C.t.\texttt{mths}}{(C \vdash \mathbb{E}[v.m(\overline{v})]) \to (C \vdash \mathbb{E}[E[v/\texttt{this}, \overline{v}/\overline{x}]\ \mathsf{in}\ p])}$$

(Eval-field)

$$\frac{C.v = \{\texttt{package}\ p; \texttt{object}\ o : t\ \texttt{implements}\ \bar{t}\{\overline{F}\}\} \quad f = w \in \overline{F}}{(C \vdash \mathbb{E}[v.f]) \to (C \vdash \mathbb{E}[w])}$$

(Eval-field-update)

$$\frac{\begin{array}{c} C.v = \{\texttt{package}\ p; \texttt{object}\ o : t\ \texttt{implements}\ \bar{t}\{\overline{F}\}\} \\ (f = u;) \in \overline{F} \\ C' = C + \{\texttt{package}\ p; \texttt{object}\ o : t\ \texttt{implements}\ \bar{t}\{\overline{F}'\}\} \\ \overline{F}' = \overline{F} + (f = w) \end{array}}{(C \vdash \mathbb{E}[v.f = w]) \to (C' \vdash \mathbb{E}[w])}$$

(Eval-new)

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

(Eval-coercion)

$$\frac{}{(C \vdash \mathbb{E}[v\ \mathsf{in}\ p]) \to (C \vdash \mathbb{E}[v])}$$

(Eval-if-true)

$$\frac{}{(C \vdash \mathbb{E}[(v == v?E : E')]) \to (C \vdash \mathbb{E}[E])}$$

(Eval-if-false)

$$\frac{v \neq w}{(C \vdash \mathbb{E}[(v == w?E : E')]) \to (C \vdash \mathbb{E}[E'])}$$

(Eval-concatenation)

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

Fig. 8. Dynamic semantics of Java Jr.

$$\begin{aligned} L &::= a \mid \tau \\ a &::= g? \mid g! \\ g &::= v.m(\overline{v}) \mid \texttt{return}\ v \mid \mathsf{new}(v).g \\ \mathsf{fn}(v.m(\overline{v})) &= \{v\} \cup \{v_i \mid v_i \in \overline{v}\} \\ \mathsf{fn}(\texttt{return}\ v) &= \{v\} \\ \mathsf{fn}(\mathsf{new}(v).g) &= \mathsf{fn}(g) \setminus \{v\} \end{aligned}$$

Fig. 9. Labels and free names in labels definition for the trace semantics of Java Jr.

Definition 1 (High-level state). *The state of a high level component C, denoted Σ, is defined as follows:*

$$\begin{aligned} \Sigma ::= &\ (C \vdash \mathsf{blk} \triangleright \overline{\mathbb{E}} : \bar{t} \to \bar{t}') \\ &\mid (C \vdash E : t \triangleright \overline{\mathbb{E}} : \bar{t} \to \bar{t}') \end{aligned}$$

where E is an expression of C, blk is a marker indicating that control is outside of C and $\overline{\mathbb{E}}$ models the evaluation stack, every entry $\mathbb{E}_i$ of $\overline{\mathbb{E}}$ having a hole of type t_i, yielding a result of type t'_i.

The state Σ models whether an external testing component is executing code (blk) or the component under test is executing (E). When blk is executing, it may call methods of the component which is being tested: C.

The relation $\Sigma \stackrel{\overline{a}}{\Longrightarrow} \Sigma'$ is defined in Fig. 10, it describes the sequence of actions a well-typed component can engage in.

(Trace-silent)
$$\frac{(C \vdash E) \rightarrow (C' \vdash E')}{(C \vdash E : t \rhd \overline{\mathbb{E}} : \overline{t} \rightarrow \overline{u}) \xrightarrow{\tau} (C' \vdash E' : t \rhd \overline{\mathbb{E}} : \overline{t} \rightarrow \overline{u})}$$

(Trace-method-call)
$$\frac{C.p.v = \{\texttt{package}\ p; \texttt{object}\ v : u\ \texttt{implements Obj}\ \{\overline{F}\}\} \qquad C \vdash v : u\ \mathsf{in}\ * \qquad m(\overline{x} : \overline{s}) : t'; \in C.u.\mathsf{hdrs} \qquad C' = C + \texttt{extern}\ \overline{v} : \overline{s};}{(C \vdash \mathsf{blk} \rhd \overline{\mathbb{E}} : \overline{t} \rightarrow \overline{u}) \xrightarrow{v.m(\overline{v})?} (C' \vdash v.m(\overline{v}) : t' \rhd \overline{\mathbb{E}} : \overline{t} \rightarrow \overline{u})}$$

(Trace-returnback)
$$\frac{C' = C + \texttt{extern}\ v : t;}{(C \vdash \mathsf{blk} \rhd \mathbb{E}, \overline{\mathbb{E}} : t \rightarrow u, \overline{t} \rightarrow \overline{u}) \xrightarrow{\mathsf{return}\ v?} (C' \vdash \mathbb{E}[v] : u \rhd \overline{\mathbb{E}} : \overline{t} \rightarrow \overline{u})}$$

(Trace-method-callback)
$$\frac{C.p.v = \{\texttt{extern}\ v : u, \texttt{Obj}\} \qquad C \vdash v : u\ \mathsf{in}\ * \qquad m(\overline{x} : \overline{s}) : s; \in C.u.\mathsf{hdrs} \qquad C' = C + \texttt{object}\ \overline{v} : \overline{s};}{(C \vdash \mathbb{E}[v.m(\overline{v})] : t \rhd \overline{\mathbb{E}} : \overline{t} \rightarrow \overline{u}) \xrightarrow{v.m(\overline{v})!} (C' \vdash \mathsf{blk} \rhd \mathbb{E}, \overline{\mathbb{E}} : s \rightarrow t, \overline{t} \rightarrow \overline{u})}$$

(Trace-return)
$$\frac{C' = C + \texttt{object}\ v : t;}{(C \vdash v : t \rhd \overline{\mathbb{E}} : \overline{t} \rightarrow \overline{u}) \xrightarrow{\mathsf{return}\ v!} (C' \vdash \mathsf{blk} \rhd \overline{\mathbb{E}} : \overline{t} \rightarrow \overline{u})}$$

(Trace-fresh-extern)
$$\frac{C.p \text{ is an import} \qquad p.o \notin \mathsf{dom}(C) \qquad p.o \in \mathsf{fn}(g?) \qquad C'' = C + \{\texttt{package}\ p; \texttt{extern}\ o : \texttt{Obj};\} \qquad (C'' \vdash \mathsf{blk} \rhd \overline{\mathbb{E}} : \overline{t} \rightarrow \overline{u}) \xrightarrow{g?} (C' \vdash E' : t' \rhd \overline{\mathbb{E}}' : \overline{t}' \rightarrow \overline{u}')}{(C \vdash \mathsf{blk} \rhd \overline{\mathbb{E}} : \overline{t} \rightarrow \overline{u}) \xrightarrow{\mathsf{new}(p.o).g?} (C' \vdash E' : t' \rhd \overline{\mathbb{E}}' : \overline{t}' \rightarrow \overline{u}')}$$

(Trace-fresh-object)
$$\frac{C.p.o = \{\texttt{package}\ p; \texttt{object}\ o : u\ \texttt{implements}\ \epsilon\{\overline{F}\}\} \qquad p.o \in \mathsf{fn}(g!) \qquad C'' = C + \{\texttt{package}\ p; \texttt{object}\ o : u\ \texttt{implements Obj}\{\overline{F}\}\} \qquad (C'' \vdash E : t \rhd \overline{\mathbb{E}} : \overline{t} \rightarrow \overline{u}) \xrightarrow{g!} (C' \vdash \mathsf{blk} \rhd \overline{\mathbb{E}}' : \overline{t}' \rightarrow \overline{u}')}{(C \vdash E : t \rhd \overline{\mathbb{E}} : \overline{t} \rightarrow \overline{u}) \xrightarrow{\mathsf{new}(p.o).g!} (C' \vdash \mathsf{blk} \rhd \overline{\mathbb{E}}' : \overline{t}' \rightarrow \overline{u}')}$$

(Trace-refl)
$$\frac{}{\Sigma \stackrel{\epsilon}{\Longrightarrow} \Sigma}$$

(Trace-trans)
$$\frac{\Sigma \stackrel{\overline{a}}{\Longrightarrow} \Sigma'' \qquad \Sigma'' \stackrel{\overline{a}'}{\Longrightarrow} \Sigma'}{\Sigma \stackrel{\overline{a}\overline{a}'}{\Longrightarrow} \Sigma'}$$

(Trace-tau)
$$\frac{\Sigma \xrightarrow{\tau} \Sigma'}{\Sigma \stackrel{\epsilon}{\Longrightarrow} \Sigma'}$$

(Trace-action)
$$\frac{\Sigma \xrightarrow{a} \Sigma'}{\Sigma \stackrel{a}{\Longrightarrow} \Sigma'}$$

Fig. 10. Trace semantics for Java Jr.

The trace semantics of a component C is: $\mathsf{Traces}_{\mathsf{H}}(C) = \{\overline{a} \mid (C \vdash \mathsf{blk} \rhd \epsilon : \epsilon) \overset{\overline{b}}{\Longrightarrow} \Sigma \text{ and } \overline{a} \equiv_\alpha \overline{b}\}$

Let us conclude this section with Example 1, which provides an example of the trace semantics.

Example 1 (Trace semantics & new labels). Consider the code presented in Listing 1.1, adopt C to refer to that code, the following is a trace it can generate.

$$
\begin{aligned}
&(C \vdash \mathsf{blk} \rhd \epsilon : \epsilon) \\
\xrightarrow{extFoo.createFoo()?}&(C \vdash extFoo.createFoo() \rhd \epsilon : \epsilon) \\
\xrightarrow{\mathtt{new}(o).\mathtt{return}\ o!}&(C \vdash \mathsf{blk} \rhd \epsilon : \epsilon) \\
\xrightarrow{o.getCounter()?}&(C \vdash o.getCounter() \rhd \epsilon : \epsilon) \\
\xrightarrow{\mathtt{return}\ 0!}&(C \vdash \mathsf{blk} \rhd \epsilon : \epsilon)
\end{aligned}
$$

An example that shows the usage of the stack $\overline{\mathbb{E}}$ can be found in [11].

C Low-Level Language

For the sake of completeness, the following section presents the low-level machine model. The reader is referred to [18] for a more thorough treatment of the formalisation of the low-level language.

C.1 Syntax

The low-level language 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 R_0 to 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. 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.

Fig. 11 presents elements of the formalisation. Words w are either instructions i or sequences of bits 0 or 1 of length ℓ. Instructions i are elements of the set $\mathcal{I}$ and define the programming language executed on the architecture (Fig. 12). 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 in another memory. Formally, if

Words	$w ::= i \in \mathcal{I}$	*Memories*	$m ::= \emptyset$
	$\mid [0 \text{ or } 1]^\ell$		$\mid m; a \mapsto w$
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. 11. Elements of the assembly language formalisation.

$\texttt{movl r}_\texttt{d}\ \texttt{r}_\texttt{s}$	Load the word from the memory address in register $\texttt{r}_\texttt{s}$ into register $\texttt{r}_\texttt{d}$.
$\texttt{movs r}_\texttt{d}\ \texttt{r}_\texttt{s}$	Store the contents of register $\texttt{r}_\texttt{s}$ at the address found in register $\texttt{r}_\texttt{d}$.
$\texttt{movi r}_\texttt{d}\ \texttt{k}$	Load the constant value $\texttt{k}$ into register $\texttt{r}_\texttt{d}$. Note that $\texttt{k} < 2^\ell$.
$\texttt{add r}_\texttt{d}\ \texttt{r}_\texttt{s}$	Write $\texttt{r}_\texttt{d} + \texttt{r}_\texttt{s} \bmod 2^\ell$ into register $\texttt{r}_\texttt{d}$ and set the ZF flag accordingly.
$\texttt{sub r}_\texttt{d}\ \texttt{r}_\texttt{s}$	Write $\texttt{r}_\texttt{d} - \texttt{r}_\texttt{s} \bmod 2^\ell$ into register $\texttt{r}_\texttt{d}$ and set both the ZF and the SF flags accordingly.
$\texttt{cmp r}_\texttt{1}\ \texttt{r}_\texttt{2}$	Calculate $\texttt{r}_\texttt{1} - \texttt{r}_\texttt{2}$ and set both the ZF and the SF flags accordingly.
$\texttt{jmp r}_\texttt{i}$	Jump to the address located in register $\texttt{r}_\texttt{i}$.
$\texttt{je r}_\texttt{i}$	If the ZF flag is set, jump to the address in register $\texttt{r}_\texttt{i}$.
$\texttt{jl r}_\texttt{i}$	If the SF flag is set, jump to the address in register $\texttt{r}_\texttt{i}$.
$\texttt{call r}_\texttt{i}$	Push the value of the program counter onto the stack and jump to the address in register $\texttt{r}_\texttt{i}$.
`ret`	Pop a value from the stack and jump to the popped location.
`halt`	Stop the execution with the result in register R_0.

Fig. 12. Instruction set $\mathcal{I}$.

$\texttt{dom}(m) \cap \texttt{dom}(m') = \emptyset$, 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. 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.

C.2 Operational Semantics

Before introducing the semantics, a number of auxiliary notions are defined.

Fig. 13 defines the access control enforcement rules. Read judgments $s \vdash \texttt{predicate}(a, b, \cdots)$ as: "according to s, $\texttt{predicate}$ holds for addresses a, b, $\ldots$".

Define functions m_{sec}m,s and m_{ext}m,s, which return the protected and unprotected parts of a memory m according to the descriptor s, respectively as:

$$\frac{a_b \le p < (a_b + n_c + n_d)}{s \vdash \texttt{protected}(p)}\ \text{(Aux-protected)} \qquad \frac{p < a_b}{s \vdash \texttt{unprotected}(p)}\ \text{(Aux-unprotected1)} \qquad \frac{(a_b + n_c + n_d) \le p}{s \vdash \texttt{unprotected}(p)}\ \text{(Aux-unprotected2)}$$

$$\frac{p = a_b + (n-1) \cdot \mathcal{N}_e}{s \vdash \texttt{returnEntryPoint}(p)}\ \text{(Aux-returnEntry)} \qquad \frac{p = a_b + m \cdot \mathcal{N}_e \quad m \in \mathbb{N} \quad m < n}{s \vdash \texttt{entryPoint}(p)}\ \text{(Aux-entryPoint)} \qquad \frac{(a_b + n_c) \le p \quad p < (a_b + n_c + n_d)}{s \vdash \texttt{data}(p)}\ \text{(Aux-data)}$$

$$\frac{s \vdash \texttt{protected}(p) \quad s \vdash \texttt{protected}(a)}{s \vdash \texttt{readAllowed}(p, a)}\ \text{(Aux-read-1)} \qquad \frac{s \vdash \texttt{unprotected}(p) \quad s \vdash \texttt{unprotected}(a)}{s \vdash \texttt{readAllowed}(p, a)}\ \text{(Aux-read-2)} \qquad \frac{s \vdash \texttt{unprotected}(a)}{s \vdash \texttt{writeAllowed}(p, a)}\ \text{(Aux-write-1)}$$

$$\frac{s \vdash \texttt{protected}(p) \quad s \vdash \texttt{data}(a)}{s \vdash \texttt{writeAllowed}(p, a)}\ \text{(Aux-write-2)} \qquad \frac{s \vdash \texttt{unprotected}(p) \quad s \vdash \texttt{entryPoint}(p')}{s \vdash \texttt{entryJump}(p, p')}\ \text{(Aux-entry)} \qquad \frac{s \vdash \texttt{protected}(p) \quad s \vdash \texttt{unprotected}(p')}{s \vdash \texttt{exitJump}(p, p')}\ \text{(Aux-return)}$$

$$\frac{s \vdash \texttt{protected}(p) \quad s \vdash \texttt{protected}(p') \quad s \nvdash \texttt{data}(p')}{s \vdash \texttt{intJump}(p, p')}\ \text{(Aux-internal)} \qquad \frac{s \vdash \texttt{unprotected}(p) \quad s \vdash \texttt{unprotected}(p')}{s \vdash \texttt{extJump}(p, p')}\ \text{(Aux-external)}$$

Fig. 13. Access control enforcement rules. Assume $s \equiv (a_b, n_c, n_d, n)$

$m_{sec}m, s = \{a \mapsto w \mid a \mapsto w \in m, s \vdash \texttt{protected}(a)\}$ and $m_{ext}m, s = \{a \mapsto w \mid a \mapsto w \in m, s \vdash \texttt{unprotected}(a)\}$.

In the semantics there are two call stacks, one for the protected code, called the *secure stack*, and one for the unprotected code, called the *insecure stack*. Each stack is preceded by a word containing the location of the current head of the stack, let $\mathsf{SP}_{\mathsf{sec}}$ and $\mathsf{SP}_{\mathsf{ext}}$ indicate the address of the secure and insecure stack respectively. When the current stack is changed in the semantics, the stack pointer register SP is initialised to the right address using $\mathsf{SP}_{\mathsf{sec}}$ and $\mathsf{SP}_{\mathsf{ext}}$. Given a memory descriptor $s = (a_b, n_c, n_d, n)$, the secure stack starts at the beginning of the protected data section and the insecure stack starts at the end of the protected memory partition. Thus $\mathsf{SP}_{\mathsf{sec}} = (a_b + n_c)$ and, initially, $\mathsf{SP}_{\mathsf{sec}} \mapsto (a_b + n_c + 1)$; analogously, $\mathsf{SP}_{\mathsf{ext}} = (a_b + n_c + n_d)$ and, initially, $\mathsf{SP}_{\mathsf{ext}} \mapsto (a_b + n_c + n_d + 1)$. Call and return instructions are responsible of setting the SP register to the correct address when crossing boundaries between protected and unprotected memory. The value of the program counter is pushed onto the stack by a `call` instruction (the stack grows down), while a `ret` instruction pops one address from the top of the stack and jumps to that location. Updating the stack pointer SP is performed by the auxiliary function `setStack` (Fig. 14). 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. Given a jump between addresses p and

$$\frac{\begin{array}{c}\text{(Stack-out-to-in)}\\ s \vdash \texttt{entryJump}(p, p') \\ m' = m[\mathsf{SP}_{\mathsf{ext}} \mapsto \mathsf{SP}] \quad r' = r[\mathsf{SP} \mapsto m(\mathsf{SP}_{\mathsf{sec}})] \\ s \vdash \texttt{unprotected}\ r(\mathsf{SP}) \quad s \vdash \texttt{protected}\ r'(\mathsf{SP})\end{array}}{p, r, m, s \vdash \texttt{setStack} \searrow p', r', m'}$$

$$\frac{\begin{array}{c}\text{(Stack-in-to-out)}\\ s \vdash \texttt{exitJump}(p, p') \\ m' = m[\mathsf{SP}_{\mathsf{sec}} \mapsto \mathsf{SP}] \quad r' = r[\mathsf{SP} \mapsto m(\mathsf{SP}_{\mathsf{ext}})] \\ s \vdash \texttt{protected}\ r(\mathsf{SP}) \quad s \vdash \texttt{unprotected}\ r'(\mathsf{SP})\end{array}}{p, r, m, s \vdash \texttt{setStack} \searrow p', r', m'}$$

$$\frac{\begin{array}{c}\text{(Stack-no-change-i)}\\ s \vdash \texttt{intJump}(p, p') \quad s \vdash \texttt{protected}\ r(\mathsf{SP})\end{array}}{p, r, m, s \vdash \texttt{setStack} \searrow p', r, m}$$

$$\frac{\begin{array}{c}\text{(Stack-no-change-e)}\\ s \vdash \texttt{extJump}(p, p') \quad s \vdash \texttt{unprotected}\ r(\mathsf{SP})\end{array}}{p, r, m, s \vdash \texttt{setStack} \searrow p', r, m}$$

Fig. 14. Stack switch enforcement rules.

p', the stack switch rules produce a new register file r' and a new memory m' based on old ones r and m. The memory is updated to store the top of the current stack, located in SP, in the address storing the top of the current stack: $\mathsf{SP}_{\mathsf{sec}}$ or $\mathsf{SP}_{\mathsf{ext}}$. When the stack is changed, the register file is updated to initialise SP to the top of the right stack: the address stored at $\mathsf{SP}_{\mathsf{sec}}$ or $\mathsf{SP}_{\mathsf{ext}}$.

Given $\Omega = (p, r, f, m, s)$, let $\lfloor \Omega \rfloor$ be the state $(p, r, f, m_{sec}m, s, s)$ and $\lceil \Omega \rceil$ be the state $(p, r, f, m_{ext}m, s, s)$. 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. Fig. 15 presents the rules for $\xrightarrow{i}$, rules for $\xrightarrow{e}$ are obtained by replacing an `intJump` assumption with an `extJump` one. Let $m(p) = \texttt{inst}$ denote that `inst` is the word allocated in $m(p)$, where $\texttt{inst} \in \mathcal{I}$. 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 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 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 3 (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.*

$$\frac{m(p) = (\mathtt{movl}\ \mathtt{r_d}\ \mathtt{r_s}) \quad s \vdash \mathtt{intJump}(p, p+1) \quad s \vdash \mathtt{readAllowed}(p, r(\mathtt{r_s})) \quad r' = r[\mathtt{r_d} \mapsto m(r(\mathtt{r_s}))]}{(p, r, f, m, s) \xrightarrow{i} (p+1, r', f, m, s)}\ (\textsf{Eval-movl})$$

$$\frac{m(p) = (\mathtt{movs}\ \mathtt{r_d}\ \mathtt{r_s}) \quad s \vdash \mathtt{intJump}(p, p+1) \quad s \vdash \mathtt{writeAllowed}(p, r(\mathtt{r_d})) \quad m' = m[r(\mathtt{r_d}) \mapsto r(\mathtt{r_s})]}{(p, r, f, m, s) \xrightarrow{i} (p+1, r, f, m', s)}\ (\textsf{Eval-movs})$$

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

$$\frac{m(p) = (\mathtt{cmp}\ \mathtt{r_1}\ \mathtt{r_2}) \quad s \vdash \mathtt{intJump}(p, p+1) \quad f' = f[\mathsf{ZF} \mapsto (\mathtt{r_1} == \mathtt{r_2}); \mathsf{SF} \mapsto (\mathtt{r_1} < \mathtt{r_2})]}{(p, r, f, m, s) \xrightarrow{i} (p+1, r, f', m, s)}\ (\textsf{Eval-compare})$$

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

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

$$\frac{m(p) = (\mathtt{call}\ \mathtt{r_d}) \quad p' = m(r(\mathtt{r_d})) \quad s \vdash \mathtt{intJump}(p, p') \quad p, r, m, s \vdash \mathsf{setStack} \searrow p', r', m' \quad r'' = r'[\mathsf{SP} \mapsto r(\mathsf{SP}) + 1] \quad m'' = m'[r''(\mathsf{SP}) \mapsto p+1]}{(p, r, f, m, s) \xrightarrow{i} (p', r'', f, m'', s)}\ (\textsf{Eval-function-call})$$

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

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

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

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

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

$$\frac{m(p) = (\mathtt{jmp}\ \mathtt{r_d}) \quad p' = r(\mathtt{r_d}) \quad s \vdash \mathtt{intJump}(p, p')}{(p, r, f, m, s) \xrightarrow{i} (p', r, f, m, s)}\ (\textsf{Eval-jump})$$

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

Fig. 15. Operational semantics of instructions in the low-level programs in the protected memory partition.

The operational semantics of the low-level language is a binary relation over states $\twoheadrightarrow \subseteq \Omega \times \Omega$ defined by the rules of Fig. 16. Rules $\mathsf{Eval-callback}$ and $\mathsf{Eval-returnback}$ ensure that the address to be followed after a callback is stored in the secure stack and that the address returnback entry point $\mathsf{A_{rb}}$ is pushed on the insecure stack. Thus the unprotected code always jumps to the returnback

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

(Eval-movs-out)

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

(Eval-callback)

$$\frac{\begin{array}{c} m(p) = (\texttt{call}\ \texttt{r}_\texttt{d}) \quad p' = m(r(\texttt{r}_\texttt{d})) \quad s \vdash \texttt{exitJump}(p, p') \\ r' = r[\mathsf{SP} \mapsto \mathsf{SP}+1] \quad m' = m[r(\mathsf{SP}) \mapsto p+1] \\ p, r', m', s \vdash \mathsf{setStack} \searrow p', r'', m'' \\ r''' = r''[\mathsf{SP} \mapsto \mathsf{SP}+1] \quad m''' = m''[r'''(\mathsf{SP}) \mapsto \mathsf{A_{rb}}] \end{array}}{(p, r, f, m, s) \twoheadrightarrow (p', r''', f, m''', s)}$$

(Eval-call)

$$\frac{\begin{array}{c} m(p) = (\texttt{call}\ \texttt{r}_\texttt{d}) \quad p' = m(r(\texttt{r}_\texttt{d})) \quad s \vdash \texttt{entryJump}(p, p') \\ p, r, m, s \vdash \mathsf{setStack} \searrow p', r', m' \\ r'' = r'[\mathsf{SP} \mapsto \mathsf{SP}+1] \quad m'' = m'[r''(\mathsf{SP}) \mapsto p+1] \end{array}}{(p, r, f, m, s) \twoheadrightarrow (p', r'', f, m'', s)}$$

(Eval-returnback)

$$\frac{\begin{array}{c} m(p) = (\texttt{ret}) \quad p' = m(r(\mathsf{SP})) = \mathsf{A_{rb}} \quad s \vdash \texttt{entryJump}(p, p') \\ r' = r[\mathsf{SP} \mapsto \mathsf{SP}-1] \quad p, r', m, s \vdash \mathsf{setStack} \searrow p', r'', m' \end{array}}{(p, r, f, m, s) \twoheadrightarrow (p', r'', f, m', s)}$$

(Eval-return)

$$\frac{\begin{array}{c} m(p) = (\texttt{ret}) \quad p' = m(r(\mathsf{SP})) \quad s \vdash \texttt{exitJump}(p, p') \\ r' = r[\mathsf{SP} \mapsto \mathsf{SP}-1] \quad p, r', m, s \vdash \mathsf{setStack} \searrow p', r'', m' \end{array}}{(p, r, f, m, s) \twoheadrightarrow (p', r'', f, m', s)}$$

Fig. 16. Operational semantics of whole low-level programs. $\mathsf{A_{rb}}$ is the address of the returnback entry point.

entry point when returning from a callback. Code located at the returnback entry point must contain a `ret` instruction in order to correctly resume the execution. The compiler ensures that rule $\mathsf{Eval-movs-out}$ will never be executed and that unused registers and flags are always reset to 0 when crossing the boundary.

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 (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* $s = (a_b, n_c, n_d, n)$*,* $p_0 = (a_b + n_c + n_d + 2)$*,* $r_0 = [\mathsf{SP} \mapsto m(\mathsf{SP_{ext}}); \mathsf{R_i} \mapsto 0\ _{\mathsf{i}=0..11}]$*, and* $f_0 = [\mathsf{ZF} \mapsto 0; \mathsf{SF} \mapsto 0]$*.*

The evaluation of P terminates if $\Omega_0(P) \twoheadrightarrow^* \Omega^{\perp}$; the result of the computation is stored in $\mathsf{R_0}$. 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 $\forall n \in \mathbb{N}, \exists \Omega'.\ \Omega_0(P) \twoheadrightarrow^n \Omega'$.

C.3 Trace Semantics

This section presents the trace semantics for protected, low-level programs.

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 [11].

Definition 5 (Initial state for traces). *The initial state for traces of a program* (m, s)*, denoted as* $\Theta_0(m, s)$*, is the state* $(\mathsf{unknown}, m, s)$*.*

Below are the labels exhibited by the traces semantics.

$$\Lambda ::= \alpha \mid \tau_i \qquad \alpha ::= \surd \mid \gamma? \mid \gamma! \qquad \gamma ::= \texttt{call}\ p(r) \mid \texttt{ret}\ p\ r(r_0)$$

A label Λ can be either an observable action α or a non-observable action τ_i. Action τ_i indicates the unobservable action occurred in protected memory. Observable actions include a tick $\surd$ indicating that the evaluation has terminated. Additionally, observable actions are function calls or returns to a certain address a, combined with the registers r and flags f. Registers and flags are in the labels as they convey information on the behaviour of programs.

Fig. 17 presents the rules defining the relation $\Theta \xRightarrow{\overline{\alpha}} \Theta'$, which describe when a state Θ generates trace $\overline{\alpha}$ and results in state Θ'. The traces of a low-level program P is defined as follows: $\mathsf{Traces_L}(P)\{\overline{\alpha} \mid \exists \Theta'.\Theta_0(P) \xRightarrow{\overline{\alpha}} \Theta'\}$.

D Algorithm

This paper does not present a transliteration of the pseudo-code of the algorithm mentioned in Section 4, which would be difficult to understand. Instead, this section presents several examples of the expected output of the algorithm in different cases. The examples illustrate crucial cases the algorithm needs to consider when creating the output component.

The algorithm takes as input two low-level traces $\overline{\alpha}_1$ and $\overline{\alpha}_2$ and two components 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. The algorithm outputs a high-level component C that differentiates C_1 and C_2, called the *output component*.

For the algorithm to be correct, it should be able to detect when two different actions are encountered. Moreover, the code produced for high-level action a must be proven to generate low-level action α, where α is equivalent to a; this equivalence relation will be formalised in Definition 7 below. We do not provide a formal proof of the correctness of the algorithm, this fact can be seen from the following examples or by consulting the pseudo-code in the companion report.

$$\frac{(p,r,f,m,s) \xrightarrow{i} (p',r',f',m',s) \quad s \vdash \texttt{intJump}(p,p')}{(p,r,f,m,s) \xrightarrow{\tau_i}\!\!\!\!\rightarrow (p',r',f',m',s)} \text{ (Trace-internal)}$$

$$\frac{(p,r,f,m,s) \xrightarrow{i} (p',r',f',m',s) \quad s \vdash \texttt{protected}(p) \quad (p',r',f',m',s)^{\perp}}{(p,r,f,m,s) \xrightarrow{\surd}\!\!\!\!\rightarrow (p',r',f',m',s)} \text{ (Trace-internal-tick)}$$

$$\frac{s \vdash \texttt{entryPoint}(p)}{(\mathsf{unknown},m,s) \xrightarrow{\texttt{call}\ p(r)?}\!\!\!\!\rightarrow (p,r,f,m,s)} \text{ (Trace-call)}$$

$$\frac{s \vdash \texttt{returnEntryPoint}(p)}{(\mathsf{unknown},m,s) \xrightarrow{\texttt{ret}\ p\ r(r_0)?}\!\!\!\!\rightarrow (p,r,f,m,s)} \text{ (Trace-returnback)}$$

$$\frac{s \vdash \texttt{exitJump}(p,p') \quad m(p) = (\texttt{call}\ p') \quad r' = r[\mathsf{SP} \mapsto r(\mathsf{SP})+1] \quad m' = m[r(\mathsf{SP}) \mapsto p+1]}{(p,r,f,m,s) \xrightarrow{\texttt{call}\ p'(r)!}\!\!\!\!\rightarrow (\mathsf{unknown},m',s)} \text{ (Trace-callback)}$$

$$\frac{p' = m(\mathsf{SP}) \quad s \vdash \texttt{exitJump}(p,p') \quad m(p) = (\texttt{ret})}{(p,r,f,m,s) \xrightarrow{\texttt{ret}\ p'\ r(r_0)!}\!\!\!\!\rightarrow (\mathsf{unknown},m,s)} \text{ (Trace-return)}$$

$$\frac{}{\Theta \overset{\epsilon}{\Longrightarrow}\!\!\!\!\rightarrow \Theta} \text{ (Trace-refl)} \qquad \frac{\Theta \xrightarrow{\tau_i}\!\!\!\!\rightarrow \Theta'}{\Theta \overset{\epsilon}{\Longrightarrow}\!\!\!\!\rightarrow \Theta'} \text{ (Trace-tau-i)} \qquad \frac{\Theta \overset{\overline{\alpha}}{\Longrightarrow}\!\!\!\!\rightarrow \Theta'' \quad \Theta'' \overset{\overline{\alpha}'}{\Longrightarrow}\!\!\!\!\rightarrow \Theta'}{\Theta \overset{\overline{\alpha}\overline{\alpha}'}{\Longrightarrow}\!\!\!\!\rightarrow \Theta'} \text{ (Trace-trans)} \qquad \frac{\Theta \xrightarrow{\alpha}\!\!\!\!\rightarrow \Theta'}{\Theta \overset{\alpha}{\Longrightarrow}\!\!\!\!\rightarrow \Theta'} \text{ (Trace-action)}$$

Fig. 17. Rules of the trace semantics.

To further support this statement, the algorithm has been implemented in Scala, and it outputs Java components that adhere to the Java Jr. formalisation.[2]

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 of four types: call, return, callback, returnback. 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.

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, exits with value 1 or 2 based on which component it is interacting with. The output component is given the power to end the computation via the `exit` statement to avoid being trapped in infinite loops deriving from infinite traces.

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

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`, `unit` for type `Unit` and `null` otherwise. Since parameters cannot be accessed by external code, the value they are initialised to is not important. For the sake of simplicity, the following examples have constructors with no parameters.

The output component is extended with extra classes. Firstly, 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 1.1, whose DIP is defined in Listing 1.5.

```
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 extTrans1 : Transaction;
10 extern extTrans2 : Transaction;
```

Listing 1.5. Example of a distinguished import package.

Omitted code is the same in both C_1 and C_2 and can be found in Listing 1.1. 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.

The low-level traces will be massaged to aid understanding. For example, given that object `extAccount` is compiled to address 0x123 and that method `createAccount` is located at address 0x456, the low- level label `call` 0x456(0x123) is written as `extAccount.createAccount()`. Numbers in italic font, e.g. *1*, refer to indexes from $\mathcal{O}$, while identities of externally allocated objects are numbers in hexadecimal.

Example 2 (Different returned values). Consider the following implementations for C_1 and C_2.

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

```
1 private 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`! $\surd$, while trace $\overline{\alpha_2}$ for C_2 is `extAccount. getBalance()`? `ret 1`! `extAccount. getBalance()`? `ret 2`! $\surd$. 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, 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. As `oc` is static, its fields are global variables for the output component.

The algorithm outputs the following code in this example:

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

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 place only when the considered action is the first: e.g. `step` is 0. The second actions are the 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 3 (Different method of a callback).

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

```
1 public createAccount(): Account {
2   Transaction b =
3     extTrans.
4       createTransaction();
5 }
```

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

To achieve differentiation in this case, the algorithm needs to keep track of in the *current method*, since it indicates where the differentiating code will be placed. The current method is recorded in a stack which is initially set to method `main` in class `Tester`. Callbacks indicate that the current method is changed to a new entry, returnback 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 following code in this example:

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

Notice that the if-statements of lines 8 and 12, whose addition was discussed in Example 2, 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.

Example 4 (Different internally-typed returned object).

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

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

Trace $\overline{\alpha_1}$ is `extAccount.createAccount()`? `ret extAc- count`! $\surd$, while trace $\overline{\alpha_2}$ is `extAccount. createAccount()`? `ret` *1*! $\surd$. 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 `Object`, 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`. Elements of this list can be accessed via method `getInternByName( n )`, which returns the object with name `n`. Additionally, method `getNameByObject( o )` returns the `name` of object `o`. The algorithm has a table with the low-level identities and the type of all dynamically-allocated objects in order to generate correct code when retrieving `internals` as in line 6 in the code below.

The algorithm outputs the following code in this example:

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

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 correctness of the call to `getInternByName("`*1*`")` in line 6.

This example scales to different internally-typed parameters in a callback. In such cases, the lookup method used is `getNameByObject`, and the differentiation is made on the names bound to different internally allocated objects that are passed as parameters in a callback.

Example 5 (Different callee of a callback).

```
1 public createAccount(): Account {
2   Transaction b =
3     extTrans1.
4       createTransaction();
5 }
```

```
1 public createAccount(): Account {
2   Transaction b =
3     extTrans2.
4       createTransaction();
5 }
```

Trace $\overline{\alpha_1}$ is `extAccount.createAccount()`? `extTrans1. createTransaction()`! $\surd$, while trace $\overline{\alpha_2}$ is `extAccount. createAccount()`? `extTrans2.createTransaction()`! $\surd$. 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 4. 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 following code in this example:

```
1 // same main as in Example 3
2 public createTransaction() : Transaction {
3   if ( oc.getStep() == 1 ) {
4     if ( this.getName() == "extTrans1" ) { exit( 1 );}
5     else { exit( 2 ); }
6   }
7   return null;
8 }
```

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

Example 6 (Different externally-typed callback parameter).

```
1 public createAccount(): Account {
2   Transaction b =
3     extTrans.createTransaction();
4   b.callback( b );
5 }
```

```
1 public createAccount(): Account {
2   Transaction b =
3     extTrans.createTransaction();
4   b.callback( extTrans );
5 }
```

Trace $\overline{\alpha_1}$ is `extAccount.createAccount()`? `extTrans. createTransaction()`! `ret 0x6`? `0x6.callback( 0x6 )`! $\surd$, while trace $\overline{\alpha_2}$ is `extAccount. createAccount()`? `extTrans.createTransaction()`! `ret 0x6`? `0x6.callback( extTrans )`! $\surd$. 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, using the field `name` of external objects to achieve differentiation.

The algorithm outputs the following code in this example:

```
1 // same main and createTransaction from Example 5,
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 { exit(2); }
8   }
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()`.

Example 7 (Traces of different length).

```
1 public createAccount(): Account {
2   while ( 1 == 1 ) {
3     skip; };
4   return null;
5 }
```

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

Trace $\overline{\alpha_1}$ is `extAccount.createAccount()`?, while trace $\overline{\alpha_2}$ is `extAccount.createAccount()`? `ret` *1*! $\surd$. In this case, differentiating between C_1 and C_2 cannot be done by terminating with two different results since control is not returned to the output component when it interacts with C_1, since $\overline{\alpha_1}$ does not terminate with $\surd$. Here, differentiation is achieved in a classical sense, by diverging in a case and terminating in the other [19]. Previous examples did not adopt this approach since differentiation could be achieved in a simpler way.

The algorithm outputs the following code in this example:

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

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.

Example 8 (Different callee of a callback #2).

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

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

Trace $\overline{\alpha_1}$ is `extAccount.createAccount()`? `extTrans. createTransaction()`! `ret 0x6`? `0x6.createTransaction()`! $\surd$ while $\overline{\alpha_2}$ is `extAccount.createAccount()`? `extTrans. createTransaction()`! `ret 0x6`? `extTrans. createTran- saction()`! $\surd$. In this case the difference is the external object on which a callback is performed. Here, C_1 calls `createTransaction()` on `0x6`, while C_2 calls the same method on `extTrans`.

The following code is produced by the algorithm for this example:

```
1 // same main as in Example 3
2 public createTransaction() : Transaction {
3   if ( oc.getStep() == 1 ) {
4     oc.incrementStep();
```

```
5   }
6   if ( oc.getStep() == 2 ) {
7    oc.incrementStep();
8    Transaction h = 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" ) { exit( 1 ); } else { exit( 2 ); }
17  }
18  return null;
19 }
```

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.

E Algorithm Transliteration

This section describes in words and pseudo-code the algorithm of Section 4. Recall that the algorithm takes in input two low-level traces $\overline{\alpha}_1$ and $\overline{\alpha}_2$ and two components C_1 and C_2. Traces $\overline{\alpha}_1$ and $\overline{\alpha}_2$ are generated by $C_1^{\downarrow}$ and $C_2^{\downarrow}$ when interacting with the same external memory. The algorithm outputs a high-level component that differentiates between C_1 and C_2.

Notation. Write $< x >$ to indicate the current value of variable x for the algorithm and indicate syntactical equivalence with $\equiv$.

Starting point. The code skeleton is a single package definition p_t containing the following classes. No packages defined in C_1 and C_2 are named p_t.

Class `Listable` will be extended by all classes implementing interfaces of the import package. `Listable` has two fields: `name` and `type`, getters and setters for them and a method `setAndRegister(v,t)` that sets `name=v`, `type=t` and adds the current object to the list of external objects in `oc` (defined below).

Class `Intern` provides a wrapper for internal objects and the low-level identifier corresponding to them. `Intern` has two fields, `name` and `obj`; the second is of type `Object` since it stores references to internal objects of all types.

Class `Helper`, and object `oc` implementing it, provide access to global variables. `Helper` has a `step` variable, which is used to keep track of how many actions have been evaluated. `Helper` also has field `externals`, a list of `Listable` where all external objects are stored. Similarly, it has field `internals`, a list of `Intern` where all internal objects are stored. `Helper` provides the following methods: `addExtern` and `addIntern` to add elements to `internals` and `externals` respectively; `getInternByName` to retrieve an internal object given its name; `getNameByObj` to retrieve an internal object's name given the object and `getExtern` to retrieve an external object given its name and type.

Finally, a `Tester` class is created, it contains the `main` method.

The skeleton is then extended based on the import package defined by C_1 and C_2. Listing 1.6 presents the skeleton code for the import package of Listing 1.5.

```
1  package p_t;
2  ... // definition of Listable, Helper, Intern, Tester and object oc
3  class Bar_c extends Listable implements Bar {
4   public createBar() : Bar {
5     return null;
6   }
7   public callback( arg : Bar ) : Unit {
8     return;
9   }
10  public defaultCreate() : Bar {
11    if ( this == staticFor_Bar ){
12      return new Bar_c();
13    }
14    return staticFor_Bar.defautCreate();
15  }
16 }
17 class Baz_c extends Listable implements Baz {
18  public lock() : Int {
19    return 0;
20  }
21  public defaultCreate() : Baz {
22    if ( this == staticFor_Baz ){
23      return new Baz_c();
24    }
25    return staticFor_Baz.defautCreate();
26  }
27 }
28 // fields come from class Listable
29 object staticFor_Bar : Bar_c { name = "staticFor_Bar", type = "Bar" }
30 object staticFor_Baz : Baz_c { name = "staticFor_Baz", type = "Baz" }
31 object extBar : Bar_c { name = "extBar", type = "Bar" }
```

Listing 1.6. Example of a skeleton code.

For all interfaces i defined in the import package, a class named "i_c" is created. These classes extend `Listable` and implement i. These classes are accompanied by an object named "`staticFor_`i". Bodies of these classes are filled with dummy method implementation that return a value whose type matches the signature: `unit` and 0 for types `Unit` and `Int` respectively, or `null` otherwise. These method bodies will change during construction phase. A new method `defaultCreate()` is added, it is a factory method that returns new instances of the class when called from the static object related to the class. Objects are allocated with default values in the parameters of constructors, their state is not needed as it is mimicked by the code generated further on. Finally, for all externs, an object with the same name is created.

Algorithm variables and data structures. Global variables used by the algorithm are indicated using an *italic* font. The counter i is used to count execution steps: each time the algorithm switches phase it increments i by one. The flag *diff* is used to capture that the differentiation has occurred and the algorithm can produce the output. The stack $\overline{c.m}$ of method invocations is used to keep track of what external methods m of class c are called; no information about the current

package is needed since the testing component consists of a single package. The top of $\overline{c.m}$, denoted m_c, is referred to as the current method. The stack $\overline{n,t}$ is used to keep track of what variable named n of type t contains the returned value of a call to an internal method. The table $\mathcal{AO}$ records the correspondence between low-level names and types of objects that are dynamically allocated, either internally or externally. Entries of $\mathcal{AO}$ have form $(v, \bar{t})$, where v is the low-level identifier of a dynamically allocated object o, and $\bar{t}$ are the interfaces implemented by o. Entries that are added to $\mathcal{AO}$ in the construction phase are entries of externally allocated objects. Conversely, entries that are added to $\mathcal{AO}$ in the execution phase are entries of internally allocated objects. The table $\mathcal{MB}$ records what the method body of an externally defined method is. Entries of $\mathcal{MB}$ have form $(c.m, s)$, where $c.m$ denotes method m of class c and s is the body of such a method. The code generated in construction and execution phase is added to this table, which is then used to generate the output of the algorithm. All of these data structures are initially empty, save for $\overline{c.m}$ that is initialised to `Tester.main`; i is set to 0 and *diff* to false.

The algorithm employs other data structures to keep track of static information of C_1 and C_2. Table $\mathbb{II}$ contains all internal interfaces. Tables $\mathbb{EM}$ and $\mathbb{IM}$ contain all method signatures defined in external and internal interfaces respectively. Table $\mathbb{SO}$ contains the high- and low-level identifiers of all static objects.

The low-level value lifting function $\uparrow(v, t)$ takes a low-level value v, a type t and returns a corresponding high-level value v' of type t. This function is employed to obtain the high-level value corresponding to a low-level one. While the encoding for ground-typed values is straightforward, non ground-typed need to be retrieved from the lists that are kept in object `oc`.

if $t =$ `Unit`	then $\uparrow(v, t) =$ `unit`
if $t =$ `Int`	then $\uparrow(v, t) = v$
if $t = p.i$ and $(v_h, v) \in \mathbb{SO}$	then $\uparrow(v, t) = v_h$
if $t = p.i$ and $t \in \mathbb{II}$	then $\uparrow(v, t) =$ `((` t `) oc.` `getInternByName(` "v" `))`
	else $\uparrow(v, t) =$ `((` t `) oc.` `getExtern(` "v" `,` "t" `))`

Construction phase. This subroutine creates code to be added in $\mathcal{MB}$ for the entry related to m_c based on the type of the actions under consideration. All generated code is added in the body of the following if statement:

```
1 if (oc.getStep() == <i>) {
2   oc.incrementStep();
3   // code is added here
4 }
```

call `call` $a(v, \overline{v})?$. This is a call to a method m compiled at address a of object v with parameters $\overline{v} = v_0, \ldots, v_j$. Table $\mathbb{IM}$ tells that the method being called

has signature $m(x_0 : t_0, \ldots, x_j : t_j) : t$ and it is defined in interface t'. For all k from 0 to j the following code is added:

```
1 t_k x_k = ↑(v_k, t_k);
```

If t_k is not ground nor internally defined, the following code is added, and an entry $(v_k, \overline{t})$ is pushed in $\mathcal{AO}$, where $\overline{t}$ are all the interfaces t_k implements.

```
1 if ( x_k == null ) {
2   x_k = staticFor_t_k.defaultCreate();
3   ( (t_k_c) x_k).setAndRegister( v_k , t_k );
4 }
```

Given a fresh name n, the following code is added, and (n, t) is pushed on $\overline{n, t}$.

```
1 t n = ↑(v, t').m(x_0,...,x_j);
```

Then the execution phase subroutine is called on the following two actions of $\overline{\alpha_1}$ and $\overline{\alpha_2}$. It will possibly return some code that needs to be added after the one just generated. Finally, an empty string is returned.

returnback $\texttt{ret}(v)?$. For an external method returning value v of type t, given a fresh name n, the following code is added:

```
1 t n = ↑(v, t);
```

If t is not ground nor internally defined, the following code is added, and an entry $(v, \overline{t})$ is pushed in $\mathcal{AO}$, where $\overline{t}$ are all the interfaces t implements.

```
1 if ( n == null ) {
2   n = staticFor_t.defaultCreate();
3   ( ( Listable ) n).setAndRegister( v , t );
4 }
```

Then the following code is added:

```
1 return n;
```

Then m_c is popped from $\overline{c.m}$ and the execution phase subroutine is called on the following two actions of $\overline{\alpha_1}$ and $\overline{\alpha_2}$. What is returned from the execution phase is returned from this phase.

Execution phase. $\alpha_1(i)$ and $\alpha_2(i)$ can be different actions or the same one.

Different actions, assume wlog $\alpha_1 = \texttt{ret}(v_1)!$ *and* $\alpha_2 = \texttt{call}\ a_2(v_2, \overline{v}_2)!$

The subroutine adds the following code in $\mathcal{MB}$ for the entry related to m_c.

```
1 if ( oc.getStep() == <i> ) { exit(2); }
```

Then this subroutine returns the code:

```
1 exit(1);
```

returns $\texttt{ret}(v_1)!$ *and* $\texttt{ret}(v_2)!$. If $v_1 \equiv v_2$ then the following code is returned, given (n, t) to be popped from $\overline{n, t}$.

```
1 oc.incrementStep();
2 oc.addInternal( v1 , n );
```

The second line is added only if t is internally defined, in which case an entry of the form $(v_1, \bar{t})$ is added to $\mathcal{AO}$, where $\bar{t}$ are the interfaces implemented by t.

If $v_1 \not\equiv v_2$, the following code is returned and *diff* is set to `true`:

```
1 oc.addInternal( v1 , n );
2 if ( n == ↑(v,t) ) { exit(1); } else { exit(2); }
```

callbacks `call` $a_1(v_1, \overline{v}_1)!$ *and* `call` $a_2(v_2, \overline{v}_2)!$.

If $a_1 \equiv a_2$ and $v_1 \equiv v_2$ and $\overline{v}_1 \equiv \overline{v}_2$, assume that class c defines method m that corresponds to a_1 and a_2, $c.m$ is pushed on $\overline{c.m}$. The following code is added to $\mathcal{MB}$ for the entry related to $c.m$:

```
1 if ( oc.getStep() == <i> ) {
2   oc.incrementStep();
3   //code is added here
4 }
```

For all arguments $v_j \in \overline{v_1}$ with type t_j where t_j is internally defined, the following code is added after the comment in the previous code and an entry of the form $(v_j, \bar{t})$ is added to $\mathcal{AO}$, where $\bar{t}$ are the interfaces implemented by t_j.

```
1 oc.addInternal( vj , xj );
```

Then the subroutine invokes the construction subroutine on the following two actions of $\overline{\alpha_1}$ and $\overline{\alpha_2}$ returning what the construction phase returns.

If the only difference is in the value of parameter x_j of type t_j, that has value v_j in the first trace, three cases arise. In all of them the generated code is added to $\mathcal{MB}$ for the entry related to $c.m_1$, where c is the class defining method m_1. The following code is added in case t_j is ground, internally or externally defined.

```
1 if ( xj == ↑(vj,tj) ) { exit(1); } else { exit(2); }
```

```
1 if ( oc.getNameByObject( xj ) == vj ) { exit(1); } else { exit(2); }
```

```
1 if ( ( ( Listable ) xj).getName() == vj ) { exit(1); } else { exit(2); }
```

If the two method names are different, and they are defined in class c_1 and c_2, then the following code is added to $\mathcal{MB}$ for the entry related to $c_1.m_1$:

```
1 if (oc.getStep() == <i>) { exit(1); }
```

The entry for $c_2.m_2$ is expanded with analogous code that exits with result 2. This also applies if two different classes define two methods with the same name.

If the two object names are different, the following code is added to the implementation of method m:

```
1 if ( oc.getStep() == <i>) {
2   if ( ( Listable ) this ).getName() == v1) { exit(1); } else { exit(2); }
3 }
```

If only one of the two low-level action $\alpha_1(i)$ or $\alpha_2(i)$ exists, then two cases arise (assume wlog that it is $\alpha_1(i)$). In this case the algorithm must only create the code that determines which component is under test, as it sets the *diff* flag to `true` so iteration over the traces will stop.

return $\texttt{ret}(v_1)!$. The subroutine returns the code:

```
1 exit(1);
```

callback $\texttt{call}\ a(v, \overline{v})!$. The subroutine adds the following code in $\mathcal{MB}$ for the entry related to $c.m$ where m is the method corresponding to a and c is the class defining it, and returns an empty string.

```
1 if ( oc.getStep() == <i> ) { exit(1); }
```

F Proofs

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

Definition 6 (Value equivalence). *A high-level value v_h and a low-level value v_l are equivalent, denoted $v_h \equiv_{\mathrm{v}} v_l$, given $\Sigma = (C \vdash \ldots)$ and $\Theta = (p, r, f, m, s)$ whenever:*

- *if $v_h = \texttt{unit}$ and $v_l = 0$;*
- *if $v_h = v : \texttt{Int}$ and $v_l = v$;*
- *if $v_h = \{\texttt{package}\ p; \texttt{object}\ o : t \ldots\}$, $v_l = i \in \mathbb{N}$, $v_h \in C$ and, given that $\mathcal{O}$ is found in m, $\mathcal{O}(i) = a$ and $s \vdash \texttt{protected}(a)$; or*
- *if $v_h = \{\texttt{package}\ p; \texttt{extern}\ o : t\}$, $v_l = a$, $v_h \in C$ and $s \vdash \texttt{unprotected}(a)$.*

Definition 7 (Action equivalence). *A high-level action a and a low-level action α are equivalent, denoted $a \equiv_{\mathrm{a}} \alpha$, given Σ and Θ whenever:*

- *if $a = \overline{\texttt{new}\ v.}\ v_h.w(\overline{v_h})?$ and $\alpha = \texttt{call}\ p(v_l, \overline{v_l})?$ and $v_h \equiv_{\mathrm{v}} v_l$, $\overline{v_h} \equiv_{\mathrm{v}} \overline{v_l}$ and, given that $\Theta = (\ldots, m', s)$, p is the entry point for w in m';*
- *if $a = \overline{\texttt{new}\ v.}\ v_h.w(\overline{v_h})!$ and $\alpha = \texttt{call}\ p(v_l, \overline{v_l})!$ and $v_h \equiv_{\mathrm{v}} v_l$, $\overline{v_h} \equiv_{\mathrm{v}} \overline{v_l}$ and, p is an address in external memory where, according to the calling convention, a call to method w must be compiled;*
- *if $a = \overline{\texttt{new}\ v.}\ \texttt{return}\ v_h?$ and $\alpha = \texttt{ret}\ p\ v_l?$ and $v_h \equiv_{\mathrm{v}} v_l$; or*
- *if $a = \overline{\texttt{new}\ v.}\ \texttt{return}\ v_h!$ and $\alpha = \texttt{ret}\ p\ v_l!$ and $v_h \equiv_{\mathrm{v}} v_l$.*

Fig. 18 contains a graphical representation of the proof scheme, where high- and low-level execution traces of two components C_1 and C_2 are related. High-level traces model component interacting with the output of the algorithm. Horizontal lines connect equivalent states and equivalent actions, the key shows what provides such an equivalence.

In Fig. 18 two low-level, even-numbered actions are the same by definition, as they are produced by the same external memory. The corresponding even-numbered, high-level actions are proven to be the same in Theorem 1 below. This

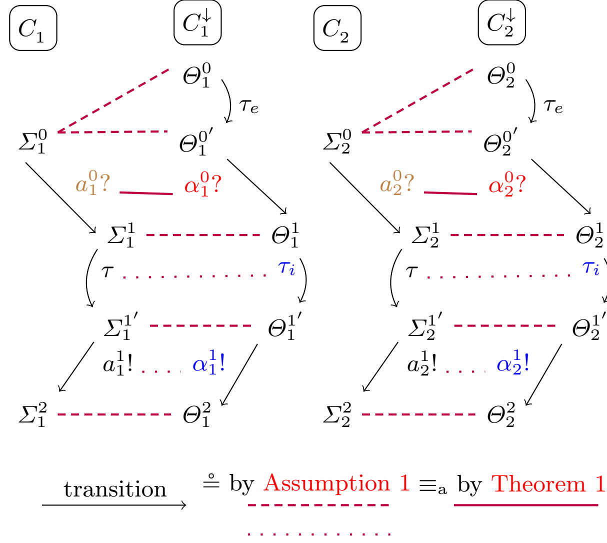

Fig. 18. Graphical representation of the proof scheme.

is because the algorithm outputs a component that replicates even numbered, low-level actions. Two low-level, odd-numbered actions may be different, in which case the corresponding high-level actions are different as stated in Assumption 1 below. What needs to be proven here is that the algorithm differentiates between the components generating those different traces.

In the proof, the algorithm receives two different low-level traces as input. This means that during the interaction between the compiled component and external memory, checks at entry points never terminate the execution.

Property 1 (Low-level traces numbering). Given a low-level trace $\overline{\alpha}$, if actions in the trace are numbered starting from 0, then every even-numbered action is a call or returnback and every odd-numbered action is a return or a callback. $\forall i \in \mathbb{N}, \alpha^{(2i)} \in \overline{\alpha} \Rightarrow \alpha^{(2i)} = \gamma?$ and $\alpha^{(2i+1)} \in \overline{\alpha} \Rightarrow \alpha^{(2i+1)} = \gamma!$.

Proof. Straightforward induction on i.

Proof of Proposition 1.

Proof. Can be found in [18].

Proof of Theorem 1.

Proof. Since m is the same while interacting with $C_1^{\downarrow}$ and $C_2^{\downarrow}$, the different action in $\overline{\alpha_1}$ and $\overline{\alpha_2}$ is found at an odd-numbered position. Moreover, by analysing the code generated in the construction phase of the algorithm, one can see that the generated component will perform high-level actions that are equal to the

low-level ones. This means that there is at least one high-level trace in the trace semantics of C_1 and C_2 whose even-numbered actions are equivalent to the even-numbered actions of $\overline{\alpha_1}$ and $\overline{\alpha_2}$. Call these traces $\overline{a_1}$ and $\overline{a_2}$ respectively. Thus $\forall i \in \mathbb{N}, a_1^{(2i)} \equiv_{\text{a}} \alpha_1^{(2i)}$; similarly for $\overline{a_2}$.

As odd-numbered actions are generated by the compiled components, apply Assumption 1 to state that the corresponding high-level action will be equivalent to it. Thus: $\forall i \in \mathbb{N}, a_1^{(2i+1)} \equiv_{\text{a}} \alpha_1^{(1i+1)}$; similarly for $\overline{a_2}$.

The execution phase of the algorithm generates code that distinguishes between C_1 and C_2 if they perform two different odd-numbered actions, a simple analysis of the execution phase code presented in Section E shows that.

Proof of Theorem 2.

Proof. The if and only if is split in two subpoints. The direction $\Leftarrow$ holds due to Assumption 1. The direction $\Rightarrow$ is reversed to the equivalent statement: $C_1^{\downarrow} \not\simeq C_2^{\downarrow} \Rightarrow C_1 \not\simeq C_2$. Apply Proposition 1 to restate the statement as $\mathsf{Traces}_{\mathsf{L}}(C_1^{\downarrow}) \neq \mathsf{Traces}_{\mathsf{L}}(C_2^{\downarrow}) \Rightarrow C_1 \not\simeq C_2$. Apply Theorem 1 to prove the statement.

G Benchmarking measurements

We have then taken a simple program and, using a hardware high-frequency timestamp, timed its performance in without any protection, in Fides and in Fides extended with the runtime checks provided by the security runtime. Table 2 below presents the overhead introduced by the secure compiler. The "Instruction" column indicates which operation have been tested. The security runtime adds checks to calls, callbacks, returns and returnbacks, so they are the only instructions that are considered. The "Normal" column indicates the cost of each operation without using the Fides architecture. The "Fides" column indicates the cost of each operation while using the Fides architecture without the secure compilation scheme. The "Prototype" indicates the cost of each operation when the secure compilation scheme is used in addition to Fides. The number following calls and callbacks indicates the number of arguments used and which trigger runtime checks. Each operation was performed 1000 times on a MacBook Pro with a 2.3 GHz Intel Core i5 processor and 4GB 1333MHz DDR3 RAM. The difference between the first two columns shows the already high overhead of adopting the Fides architecture. The difference between the third and fourth column is the overhead of the security checks introduced by the secure compiler. 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 amount of boundaries crossing.

Instruction	Normal	Fides	Prototype	Prototype overhead
call 1	0.03 μs	11.39 μs	11.52 μs	1.22 %
call 8	0.03 μs	11.39 μs	11.58 μs	1.66 %
callback 1	0.02 μs	4.65 μs	5.01 μs	7.89 %
callback 8	0.02 μs	4.65 μs	4.99 μs	7.09 %
return	0.03 μs	4.42 μs	4.49 μs	1.58 %
returnback	0.02 μs	11.91 μs	11.92 μs	0.09 %

Table 2. Cost of different instructions and overhead of the secure compilation scheme.