Secure Compilation of Object-Oriented Components to Untyped Machine Code

Marco Patrignani *Dave Clarke*
Frank Piessens

Report CW 630, January 2013

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Secure Compilation of Object-Oriented Components to Untyped Machine Code

Marco Patrignani   *Dave Clarke*
Frank Piessens

Report CW 630, January 2013

Department of Computer Science, K.U.Leuven

Abstract

A fully abstract compilation scheme prevents high-level code security features from being bypassed by an attacker operating at a lower level. This paper presents a fully abstract compilation scheme from a realistic object-oriented language with dynamic memory allocation 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, guaranteeing that low- and high-level attackers have the same power.

Secure Compilation of Object-Oriented Components to Untyped Machine Code

Marco Patrignani [⋆], Dave Clarke, and Frank Piessens

iMinds-DistriNet, Dept. Computer Science, Katholieke Universiteit Leuven
`{first.last}@cs.kuleuven.be`

Abstract. A fully abstract compilation scheme prevents high-level code security features from being bypassed by an attacker operating at a lower level. This paper presents a fully abstract compilation scheme from a realistic object-oriented language with dynamic memory allocation 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, guaranteeing that low- and high-level attackers have the same power.

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 the low level 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 a fully abstract compilation scheme is well suited for expressing the preservation of security policies through compilation, as it preserves and reflects contextual equivalence. Contextual equivalence is a relation between two programs that cannot be distinguished by a third one. Security policies can be modelled by using contextual equivalence 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, it introduces no more vulnerabilities to 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 [13] are some of the

[⋆] Marco Patrignani holds a Ph.D. fellowship from the Research Foundation Flanders (FWO).

examples where compilation is not fully abstract. Recent techniques that achieve a fully abstract compilation rely on address space layout randomisation [2,10], type-based invariants [4,9], 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. That malicious code can violate the secrecy properties of the machine code generated by the compiler if the compiler generates unsafe code that does not preserve high-level language security features.

In order to withstand such a low-level attacker, high-level security features must be preserved in the code generated during compilation. To this end, this paper presents a fully abstract compilation scheme from a high-level object-oriented language with dynamic memory allocation to low-level untyped machine code. Full abstraction of the compilation scheme is achieved by enhancing the low-level machine model with a fine-grained program counter-based memory access control mechanism inspired by existing systems [15,16,19,20]. This paper extends the work of Agten *et al.* [3] in two ways: (i) it considers a high-level language with dynamic memory allocation and (ii) it adopts a more formal and more precise approach to the definition of the languages and in the proof of full abstraction of the compilation scheme. The closest result to that presented here is that of Jagadeesan *et al.* [10]. Instead of relying on address space layout randomisation, the compilation scheme presented here relies on a protection mechanism that can be implemented in hardware. This minimises software threats to attacks at the high-level, at hardware level and side-channels attacks. This paper also adopts a low-level model 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 real-world object-oriented language with dynamic memory allocation to low-level untyped machine code;
– the outline of a formal proof of full abstraction for this compilation scheme.

A limitation of the presented work is the lack of a prototype to evaluate the performance of the compilation scheme. However, as the current implementations of low-level mechanisms are rather costly, we conjecture that the additional overhead of the presented compiler is negligible. Another limitation is that the high-level language could adopt more powerful language constructs; this is discussed in Section 7.

The paper is organised as follows. Section 2 presents the high-level language. Section 3 presents the low-level language. Section 4 presents an informal overview of how to achieve a fully abstract compilation scheme and how to prove that result. Section 5 describes the compilation scheme. Section 6 presents the proof of full abstraction for the compilation scheme. Section 7 discusses extensions to the high-level language and to the behaviour of compiled components. Section 8 discusses related work and Section 9 concludes.

2 High-Level Language

After an informal introduction, this section formally presents the high-level language used herein: Java Jr. [12], alongside its syntax, static, dynamic and trace semantics. Most of the concepts are taken from [12], with some corrections as to fill the missing bits of the formalisation.

2.1 Informal Overview of Java Jr.

Java Jr. [12] 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 external objects of unknown implementation. Export packages define *classes* and *objects*; they provide an implementation of an import package. Listing 1.1 illustrates the package system of Java Jr. as well as its syntax.

```
package p_i;
  interface Foo {
    public createFoo() : Foo;
    public getCounter() : Int;
  }
  extern extFoo : Foo;

package p_e;
  class FooClass implements p_i.Foo {
    FooClass() { counter = 0; }
    public createFoo() : Foo { return new p_e.FooClass(); }
    public getCounter() : Int { return counter; }
    private counter : Int;
  }
  object extFoo : FooClass;
```

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

Listing 1.1 contains two package declarations: $\mathtt{p}_i$ is an import package and $\mathtt{p}_e$ is an export package implementing $\mathtt{p}_i$. Object `extObj` allocated in $\mathtt{p}_e$ implements the extern with the same name defined in $\mathtt{p}_i$.

In Java Jr., ground values, types and operations on them are assumed to be provided by a `System` package, whose name is omitted for the sake of brevity. Since the focus of this paper is security, we will 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, which can be used to denote secrets. 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 packaging system, a package can be compiled just by having only the import packages of any package it depends on, adopting the principle of separate compilation. As a result, formal parameters in methods have interface types, since classes that implement those interfaces are unknown. This leads to the discipline of "programming to an interface". An implication of this discipline is that cross-package object allocation is achieved through factory methods. For example, the name of class `FooClass` from Listing 1.1 is not visible from outside package $\mathtt{p}_e$, thus expressions of the form $\mathtt{new\ p}_e\mathtt{.FooClass()}$ cannot be written outside $\mathtt{p}_e$. A limitation of the language is that classes cannot extend other classes defined in different packages.

2.2 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 Figure 1.

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

Fig. 1: Syntax of Java Jr.

2.3 Notation

Before presenting the static semantics, we shall introduce some auxiliary functions. These are taken from Java Jr. [12].

Define $\overline{D}_1 \equiv \overline{D}_2$ whenever two sequences of declarations are equal up to reordering:

$$\overline{D}_1\overline{D}_2\overline{D}_3 \equiv \overline{D}_2\overline{D}_1\overline{D}_3$$

Define $\overline{P}_1 \equiv \overline{P}_2$ when two sequences of package definitions are equal up to reordering of packages and of declarations:

$$\begin{array}{ll} \overline{P}_1\overline{P}_2\overline{P}_3 \equiv \overline{P}_2\overline{P}_1\overline{P}_3 & \\ \{\texttt{package}\ p; \overline{D}_1\} \equiv \{\texttt{package}\ p; \overline{D}_2\} & \textit{if}\ \overline{D}_1 \equiv \overline{D}_2 \end{array}$$

Define $\texttt{dom}(\ )$ as:

$$\begin{aligned} \texttt{dom}(P_1 \ldots P_n) &= \texttt{dom}(P_1) \cup \ldots \cup \texttt{dom}(P_n) \\ \texttt{dom}(\{\texttt{package}\ p; \overline{D}\}) &= \{p.n \mid n \in \texttt{dom}(\overline{D})\} \cup \{p\} \\ \texttt{dom}(D) &= \{\texttt{name}(D)\} \end{aligned}$$

The domain of a list of elements is the set of elements obtained by applying the domain function to all elements of the list.

Define $\texttt{name}(\ )$ as:

$$\begin{array}{rr} \texttt{name}(\texttt{class}\ c \ldots) = c & \texttt{name}(\texttt{object}\ o \ldots) = o \\ \texttt{name}(\texttt{interface}\ i \ldots) = i & \texttt{name}(\texttt{extern}\ o : \overline{t};) = o \\ \texttt{name}(\texttt{public}\ m \ldots) = m & \texttt{name}(m(\overline{x} : \overline{t}) : t;) = m \\ \texttt{name}(f : t) = f & \texttt{name}(f = v) = f \end{array}$$

Use n to range over names. Define $\texttt{fn}(\ )$ as the free names of an entity, namely its name and the names of all other syntactic categories it contains.

Define $C.p$ as:

$$C.p = \{\texttt{package}\ p; \overline{D}\} \qquad \textit{if}\ \{\texttt{package}\ p; \overline{D}\} \in C$$

Define $C.p.n$ (or $C.t$ where $t \equiv p.n$) as:

$$C.p.n = \{\texttt{package}\ p; D\} \qquad \textit{if}\ \{\texttt{package}\ p; \overline{D}\} \in C, D \in \overline{D}, \texttt{name}(D) = n$$

Define $C.t.\texttt{superTypes}$ as:

$$\begin{aligned} &C.\texttt{Obj}.\texttt{superTypes} = \epsilon \\ &\quad C.t.\texttt{superTypes} = t', \overline{t} \\ &\qquad\qquad \textit{if}\ C.t = \{\texttt{package}\ p; \texttt{class}\ c\ \texttt{extends}\ t'\ \texttt{implements}\ \overline{t}\{K\ \overline{F_t}\ \overline{M}\}\} \\ &\quad C.t.\texttt{superTypes} = \overline{t} \\ &\qquad\qquad \textit{if}\ C.t = \{\texttt{package}\ p; \texttt{interface}\ i\ \texttt{extends}\ \overline{t}\{\overline{M_t}\}\} \end{aligned}$$

A component C is acyclic when for all types t defined in C, given the supertypes of t as $C.t.\mathtt{superTypes} = \overline{t}$, there is no $u \in \overline{t}$ that is a supertype of another supertype of t. For the remainder of the paper assume components to be acyclic.

Define the addition of components $C + C'$ as:

$$\begin{aligned} C + \epsilon &= C \\ C + (\{\mathtt{package}\ p; \overline{D}\}, C') &= (C, \{\mathtt{package}\ p; \overline{D}\}) + C' \quad \mathit{if}\ p \notin \mathtt{dom}(C) \\ (C_1, \{\mathtt{package}\ p; \overline{D}\}, C_2) + (\{\mathtt{package}\ p; \overline{D}'\}, C') &= (C_1, \{\mathtt{package}\ p; \overline{D} + \overline{D}'\}, C_2) + C' \end{aligned}$$

Define the addition of sequences of declarations $\overline{D} + \overline{D}'$ as:

$$\begin{aligned} \overline{D} + \epsilon &= \overline{D} \\ \overline{D} + (D, \overline{D'}) &= (\overline{D}, D) + \overline{D}' && \mathit{if}\ \mathtt{name}(D) \notin \mathtt{dom}(\overline{D}) \\ (\overline{D}_1, D, \overline{D}_2) + (D', \overline{D}') &= (\overline{D}_1, D', \overline{D}_2) + \overline{D}' && \mathit{if}\ \mathtt{name}(D) = \mathtt{name}(D') \end{aligned}$$

$M + M'$, $M_t + M_t'$, $F + F'$ and $F_t + F_t'$ are defined analgously.

Define $C.t.\mathtt{hdrs}$ as:

$$\begin{aligned} &C.\mathtt{Obj}.\mathtt{hdrs} = \epsilon \\ &\quad C.t.\mathtt{hdrs} = C.t'.\mathtt{hdrs} + \overline{M}.\mathtt{hdrs} \\ &\qquad\qquad \mathit{if}\ C.t = \{\mathtt{package}\ p; \mathtt{class}\ c\ \mathtt{extends}\ t'\ \mathtt{implements}\ \overline{t}\{K\ \overline{F_t}\ \overline{M}\}\} \\ &\quad C.t.\mathtt{hdrs} = C.\overline{t}.\mathtt{hdrs} + \overline{M_t} \\ &\qquad\qquad \mathit{if}\ C.t = \{\mathtt{package}\ p; \mathtt{interface}\ i\ \mathtt{extends}\ \overline{t}\{\overline{M_t}\}\} \end{aligned}$$

Define $M.\mathtt{hdrs}$ as:

$$\mathtt{public}\ m(\overline{x} : \overline{t}) : t\{\mathtt{return}\ E;\}.\mathtt{hdrs} = m(\overline{x} : \overline{t}) : t;$$

Define $\overline{t}.\mathtt{hdrs}$ as:

$$t_1.\mathtt{hdrs} + \ldots + t_n.\mathtt{hdrs} \qquad \mathit{if}\ \overline{t} = t_1 \ldots t_n$$

$\overline{M}.\mathtt{hdrs}$ is defined analogously.

Define compatibility among headers $\overline{N}$ when: if a method name occurs more than once, it has the same signature. Formally:

$$m(\overline{x} : \overline{t}) : t; \in \overline{N}\ \mathit{and}\ m(\overline{y} : \overline{u}) : u; \in \overline{N} \Rightarrow t \equiv u\ \mathit{and}\ \overline{t} \equiv \overline{u}$$

Define $C.t.\mathtt{flds}$ as:

$$\begin{aligned} &C.\mathtt{Obj}.\mathtt{flds} = \epsilon \\ &\quad C.t.\mathtt{flds} = C.t'.\mathtt{flds} + \overline{F_t} \\ &\qquad\qquad \mathit{if}\ C.t = \{\mathtt{package}\ p; \mathtt{class}\ c\ \mathtt{extends}\ t'\ \mathtt{implements}\ \overline{t}\{K\ \overline{F_t}\ \overline{M}\}\} \end{aligned}$$

Define $C.t.\mathtt{mths}$ as:

$$\begin{aligned}
C.\mathtt{Obj}.\mathtt{mths} &= \epsilon \\
C.t.\mathtt{mths} &= C.t'.\mathtt{mths} + \overline{M} \\
&\qquad \textit{if } C.t = \{\mathtt{package}\ p; \mathtt{class}\ c\ \mathtt{extends}\ t'\ \mathtt{implements}\ \bar{t}\{K\ \overline{F_t}\ \overline{M}\}\}
\end{aligned}$$

Define $C.t.\mathtt{super}$ as the supertype of type t in component C. Formally:

$$\begin{aligned}
C.\mathtt{Obj}.\mathtt{super} &= \epsilon \\
C.t.\mathtt{super} &= t' \\
&\qquad \textit{if } C.t = \{\mathtt{package}\ p; \mathtt{class}\ c\ \mathtt{extends}\ t'\ \mathtt{implements}\ \bar{t}\{K\ \overline{F_t}\ \overline{M}\}\}
\end{aligned}$$

Define $C.p$ is an export (package) if there is a n such that $C.p.n = \{\mathtt{package}\, p; D\}$ where D is either a class or an object declaration. Define $C.p$ is an import (package) if it is not an export (package).

Define $C.p.\mathtt{exports}$ to be the component containing all the export packages for C, and analogously for $C.p.\mathtt{imports}$.

2.4 Static Semantics

The static semantics defines typing relations based on the judgments of Figure 2. Adopt Γ as the standard type environment that binds variables to types, where

$\vdash C : \mathsf{cmp}$	well-typed component C
$C \vdash P : \mathsf{pkg}$	well-typed package P
$C \vdash D : \mathsf{dec\ in}\ p$	well-typed declaration D in package p
$C \vdash t : \mathsf{type\ in}\ p$	t is a valid type in package p
$C \vdash c : \mathsf{cls\ in}\ p$	c is a valid class in package p
$C \vdash i : \mathsf{itf\ in}\ p$	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
$C \vdash K : \mathsf{cnstr\ in}\ p$	well-typed constructor K in package p
$C \vdash M_t : \mathsf{hdr\ in}\ p.i$	well-typed method header M_t of interface i in package p
$C \vdash M : \mathsf{mth\ in}\ p.c$	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. 2: Typing judgments of Java Jr.

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.

Figure 3 and Figure 4 present the typing rules as from the original Java Jr. work [12]. 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. Rules Expr-concat and Scope-all have been devised by the authors as they were not presented in the original paper.

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

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

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

$$\overset{\text{(Declaration-interface)}}{\dfrac{\begin{array}{c} C \vdash \overline{t} : \mathsf{itf\ in}\ * \qquad C \vdash \overline{M_t} : \mathsf{hdr\ in}\ p.i \\ \forall t \in \overline{t}\ C.t.\mathtt{hdrs} \subseteq C.p.i.\mathtt{hdrs} \end{array}}{C \vdash \mathtt{interface}\ i\ \mathtt{extends}\ \overline{t}\ \{\overline{M_t}\} : \mathsf{dec\ in}\ p}} \qquad \overset{\text{(Declaration-extern)}}{\dfrac{\begin{array}{c} C \vdash \overline{t} : \mathsf{type\ in}\ * \\ C.\overline{t}.\mathtt{hdrs}\ \text{are compatible} \\ C.p\ \text{is an import} \end{array}}{C \vdash \mathsf{extern}\ o : \overline{t}; : \mathsf{dec\ in}\ p}}$$

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

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

$$\overset{\text{(Types-interface)}}{\dfrac{C \vdash t : \mathsf{itf\ in}\ p}{C \vdash t : \mathsf{type\ in}\ p}} \qquad \overset{\text{(Subtype-refl)}}{\dfrac{C \vdash t : \mathsf{type\ in}\ p}{C \vdash t <: t\ \mathsf{in}\ p}} \qquad \overset{\text{(Subtype-trans)}}{\dfrac{\begin{array}{c} C \vdash t <: t''\ \mathsf{in}\ p \\ C \vdash t'' <: t'\ \mathsf{in}\ p \end{array}}{C \vdash t <: t'\ \mathsf{in}\ p}} \qquad \overset{\text{(Subtype-obj)}}{\dfrac{C \vdash t : \mathsf{type\ in}\ p}{C \vdash t <: \mathtt{Obj}\ \mathsf{in}\ p}}$$

$$\overset{\text{(Subtype-def)}}{\dfrac{\begin{array}{c} C \vdash t : \mathsf{type\ in}\ p \\ t' \in C.t.\mathtt{superTypes} \end{array}}{C \vdash t <: t'\ \mathsf{in}\ p}} \qquad \overset{\text{(Scope-all)}}{\dfrac{\begin{array}{c} C.p = \{\mathsf{package}\ p; \overline{D}\} \\ v \notin \mathsf{fn}(C \setminus p) \\ C \vdash v : t\ \mathsf{in}\ p \end{array}}{C \vdash v : t\ \mathsf{in}\ *}}$$

Fig. 3: Java Jr. typing rules, part one.

(Constructors)

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

(Method-Types)

$$\frac{\begin{array}{c} C \vdash t : \mathsf{itf\ in}\ * \\ C \vdash \overline{t} : \mathsf{itf\ in}\ * \end{array}}{C \vdash m(\overline{x} : \overline{t}) : t; : \mathsf{hdr\ in}\ p}$$

(Methods)

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

(Expr-val-obj)

$$\frac{C.v = \{\mathtt{package}\ p; \mathtt{object}\ o : t\ \mathtt{implements}\ \overline{t}\{\overline{F}\}\}}{C; \mathit{\Gamma} \vdash v : t\ \mathsf{in}\ p}$$

(Expr-val-obj-itf)

$$\frac{C.v = \{\mathtt{package}\ p; \mathtt{object}\ o : t\ \mathtt{implements}\ \overline{t}\{\overline{F}\}\} \qquad t' \in \overline{t}}{C; \mathit{\Gamma} \vdash v : t'\ \mathsf{in}\ p}$$

(Expr-val-extern)

$$\frac{C.v = \{\mathtt{package}\ p; \mathtt{extern}\ o : \overline{t};\} \qquad t \in \overline{t}}{C; \mathit{\Gamma} \vdash v : t\ \mathsf{in}\ p}$$

(Expr-var)

$$\frac{\begin{array}{c} x : t \in \mathit{\Gamma} \\ C \vdash t : \mathsf{type\ in}\ p \end{array}}{C; \mathit{\Gamma} \vdash x : t\ \mathsf{in}\ p}$$

(Expr-fld)

$$\frac{\begin{array}{c} C; \mathit{\Gamma} \vdash E : t'\ \mathsf{in}\ p \\ f : t \in C.t'.\mathtt{flds} \end{array}}{C; \mathit{\Gamma} \vdash E.f : t\ \mathsf{in}\ p}$$

(Expr-fldup)

$$\frac{\begin{array}{c} C; \mathit{\Gamma} \vdash E : u\ \mathsf{in}\ p \\ C; \mathit{\Gamma} \vdash E' : t\ \mathsf{in}\ p \\ f : t \in C.u.\mathtt{flds} \end{array}}{C; \mathit{\Gamma} \vdash E.f = E' : t\ \mathsf{in}\ p}$$

(Expr-meth)

$$\frac{\begin{array}{c} C; \mathit{\Gamma} \vdash E : u\ \mathsf{in}\ p \\ C; \mathit{\Gamma} \vdash \overline{E} : \overline{t}\ \mathsf{in}\ p \\ m(\overline{x} : \overline{t}) : t \in C.u.\mathsf{hdrs} \end{array}}{C; \mathit{\Gamma} \vdash E.m(\overline{E}) : t\ \mathsf{in}\ p}$$

(Expr-new)

$$\frac{\begin{array}{c} C \vdash c : \mathsf{cls\ in}\ p \\ C \vdash \overline{E} : \overline{t} : \mathsf{in}\ p \\ C \vdash C.p.c.\mathtt{flds} : \overline{t} \end{array}}{C; \mathit{\Gamma} \vdash \mathsf{new}\ p.c(\overline{E}) : p.c\ \mathsf{in}\ p}$$

(Expr-if)

$$\frac{\begin{array}{c} C; \mathit{\Gamma} \vdash E : u\ \mathsf{in}\ p \qquad C; \mathit{\Gamma} \vdash E' : u\ \mathsf{in}\ p \\ C; \mathit{\Gamma} \vdash E_T : t\ \mathsf{in}\ p \qquad C; \mathit{\Gamma} \vdash E_F : t\ \mathsf{in}\ p \end{array}}{C; \mathit{\Gamma} \vdash (E == E' ? E_T : E_F) : t\ \mathsf{in}\ p}$$

(Expr-concat)

$$\frac{\begin{array}{c} C; \mathit{\Gamma} \vdash E : u\ \mathsf{in}\ p \\ C; \mathit{\Gamma}, E : u\ \mathsf{in}\ p \vdash E' : t\ \mathsf{in}\ p \end{array}}{C; \mathit{\Gamma} \vdash E; E' : t\ \mathsf{in}\ p}$$

(Expr-coercion)

$$\frac{\begin{array}{c} C; \mathit{\Gamma} \vdash E : t\ \mathsf{in}\ p \\ C \vdash t : \mathsf{type\ in}\ q \end{array}}{C; \mathit{\Gamma} \vdash E\ \mathsf{in}\ p : t\ \mathsf{in}\ q}$$

(Expr-subsumption)

$$\frac{\begin{array}{c} C; \mathit{\Gamma} \vdash E : t\ \mathsf{in}\ p \\ C \vdash t <: u\ \mathsf{in}\ p \end{array}}{C; \mathit{\Gamma} \vdash E : u\ \mathsf{in}\ p}$$

Fig. 4: Java Jr. typing rules, part two.

Type soundness The type system enjoys the progress and preservation properties as proven in the original Java Jr. work [12].

2.5 Dynamic Semantics

The dynamic semantics is given in terms of a relation $(C \vdash E) \to (C' \vdash E')$ 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 Figure 5.

$$\textsf{(Eval-method)}\quad \frac{\begin{array}{c} C.v = \{\texttt{package}\ p; \texttt{object}\ o : t\ \texttt{implements}\ \overline{t}\{\overline{F}\}\} \\ \texttt{public}\ m(\overline{x} : \overline{t}) : t\{\texttt{return}\ E;\} \in C.t.\texttt{mths} \end{array}}{(C \vdash \mathbb{E}[v.m(\overline{v})]) \rightarrow (C \vdash \mathbb{E}[E[v/\texttt{this}, \overline{v}/\overline{x}]\ \texttt{in}\ p])}$$

$$\textsf{(Eval-field)}\quad \frac{\begin{array}{c} C.v = \{\texttt{package}\ p; \texttt{object}\ o : t\ \texttt{implements}\ \overline{t}\{\overline{F}\}\} \\ f = w \in \overline{F} \end{array}}{(C \vdash \mathbb{E}[v.f]) \rightarrow (C \vdash \mathbb{E}[w])}$$

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

$$\textsf{(Eval-new)}\quad \frac{\begin{array}{c} C.p.c.\texttt{flds} = \overline{f} : \overline{t} \qquad p.o \notin \texttt{dom}(C) \\ C' = C + \{\texttt{package}\ p; \texttt{object}\ o : p.c\ \texttt{implements}\ \epsilon\{\overline{f} = \overline{v}\}\} \end{array}}{(C \vdash \mathbb{E}[\texttt{new}\ p.c(\overline{v})]) \rightarrow (C' \vdash \mathbb{E}[p.o])}$$

$$\textsf{(Eval-coercion)}\quad \frac{}{(C \vdash \mathbb{E}[v\ \texttt{in}\ p]) \rightarrow (C \vdash \mathbb{E}[v])} \qquad \textsf{(Eval-if-true)}\quad \frac{}{(C \vdash \mathbb{E}[(v == v ? E : E')]) \rightarrow (C \vdash \mathbb{E}[E])}$$

$$\textsf{(Eval-if-false)}\quad \frac{v \neq w}{(C \vdash \mathbb{E}[(v == w ? E : E')]) \rightarrow (C \vdash \mathbb{E}[E'])} \qquad \textsf{(Eval-concatenation)}\quad \frac{}{(C \vdash \mathbb{E}[v; E]) \rightarrow (C \vdash \mathbb{E}[E])}$$

Fig. 5: Dynamic semantics of Java Jr.

2.6 Trace Semantics

As Section 4.1 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 [12].

The paper presenting Java Jr. [12] 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:

$$\text{if } \vdash C_1 : \mathsf{cmp} \text{ and } \vdash C_2 : \mathsf{cmp} \text{ then } \mathsf{Traces_H}(C_1) = \mathsf{Traces_H}(C_2) \iff C_1 \simeq C_2$$

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 Figure 6 alongside the function for calculating free names in traces, τ is the internal silent action. Decorations ?

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

Fig. 6: Labels and free names in labels definition for the trace semantics of Java Jr.

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 update objects and externs allocated inside components.

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

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

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

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

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

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 $\Sigma \overset{\overline{a}}{\Longrightarrow} \Sigma'$ is defined in Figure 7, it describes the sequence of actions a well-typed component can engage in.

The trace semantics of a component C is:

$$\mathsf{Traces_H}(C) = \{\overline{a} \mid (C \vdash \mathsf{blk} \triangleright \epsilon : \epsilon) \overset{\overline{b}}{\Longrightarrow} \Sigma \textit{ and } \overline{a} \equiv_\alpha \overline{b}\}$$

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.

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

An example that shows the usage of the stack $\overline{\mathbb{E}}$ can be found in [12].

3 Low-Level Language

The following section presents the low-level machine model. Most of these concepts introduced here are taken from [3], some are more properly formalised and corrected.

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

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

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

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

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

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

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

$$\text{(Trace-refl)}\ \frac{}{\Sigma \xRightarrow{\epsilon} \Sigma} \qquad \text{(Trace-trans)}\ \frac{\Sigma \xRightarrow{\overline{a}} \Sigma'' \qquad \Sigma'' \xRightarrow{\overline{a}'} \Sigma'}{\Sigma \xRightarrow{\overline{a}\overline{a}'} \Sigma'} \qquad \text{(Trace-tau)}\ \frac{\Sigma \xrightarrow{\tau} \Sigma'}{\Sigma \xRightarrow{\epsilon} \Sigma'} \qquad \text{(Trace-action)}\ \frac{\Sigma \xrightarrow{a} \Sigma'}{\Sigma \xRightarrow{a} \Sigma'}$$

Fig. 7: Trace semantics for Java Jr.

3.1 Syntax

The low-level language adopted here is very similar to that presented in the work of Agten *et al.* [3]. It models a standard Von Neumann machine consisting of a program counter, a registers file, a flags register and a memory space. The registers file contains 12 general purpose registers R_0 to R_{11} and the stack pointer SP. The flag register contains the zero flag ZF and the sign flag SF. The memory space m is a function that maps addresses to words. Assume a finite number of addresses A, ranged over by a, p, they are sequences of bits. Low-level values v are sequences of bits, Figure 8 presents the syntax for instructions I and words are either values v or instructions I. Addresses, registers, instructions and values are 32 bits wide, memory is also addressed in multiples of 32 bits.

`movl` $\mathtt{r_d}$ $\mathtt{r_s}$	Load the word from memory address in register $\mathtt{r_s}$ into register $\mathtt{r_d}$.
`movs` $\mathtt{r_d}$ $\mathtt{r_s}$	Store the contents of register $\mathtt{r_s}$ in the address found in register $\mathtt{r_d}$.
`movi` $\mathtt{r_d}$ `i`	Load the constant value `i` into register $\mathtt{r_d}$. Notice that $\mathtt{i} < 32$.
`add` $\mathtt{r_d}$ $\mathtt{r_s}$	Write $\mathtt{r_d} + \mathtt{r_s} \bmod 2^{32}$ to register $\mathtt{r_d}$ and set the ZF flag accordingly.
`sub` $\mathtt{r_d}$ $\mathtt{r_s}$	Write $\mathtt{r_d} - \mathtt{r_s} \bmod 2^{32}$ to register $\mathtt{r_d}$ and set the ZF flag accordingly.
`cmp` $\mathtt{r_1}$ $\mathtt{r_2}$	Calculate $\mathtt{r_1} - \mathtt{r_2}$ and set both the ZF and the SF flags accordingly.
`jmp` $\mathtt{r_i}$	Jump to the address located in register $\mathtt{r_i}$.
`je` $\mathtt{r_i}$	If the ZF flag is set, jump to the address located in register $\mathtt{r_i}$.
`jl` $\mathtt{r_i}$	If the SF flag is set, jump to the address located in register $\mathtt{r_i}$.
`call` $\mathtt{r_i}$	Push the value of the program counter onto the stack and jump to the address located in register $\mathtt{r_i}$.
`ret`	Pop a value from the stack and jump to the popped location.
`halt`	Stop the execution with the result in register $\mathsf{R_0}$.

Fig. 8: Syntax of the low-level language.

Denote the protected memory as m_{sec} and the unprotected one as m_{ext}. A low-level program is a pair consisting of a memory space m_{sec} and a memory descriptor s associated with it: (m_{sec}, s). Define a memory descriptor s as a quadruple (b, s_c, s_d, n), where b is the base address of the protected memory, s_c is the size of the protected code section, s_d is the size of the protected data section and n is the number of entry points. Define $m(a)$ as the function that returns the word stored at address a in memory m. Whenever m is uniquely identifiable, shorten $m(a)$ with $*a$.

3.2 Dynamic Semantics

The dynamic semantics expresses the actions a low-level evaluation state Ψ can execute in order to become a new state Ψ'.

Definition 2 (Low-level state). *A low-level execution state, denoted Ψ, is defined as follows: $\Psi = (p, r, f, m, s)$, where p is the program counter, r is the register file, f is the flags register, m is the memory and s is the memory descriptor.*

Before introducing the semantics, Figure 9 defines a number of auxiliary functions that will be used by the semantics and the trace semantics.

Figure 10 presents the rules that define the small step semantics of the low-level language. Let $m(p) \cong \mathtt{inst}$ denote that `inst` is the word allocated in $m(p)$. Define an initial configuration Ψ_0 as follows:

(Aux-protected)
$$\frac{s \equiv (b, s_c, s_d, n) \qquad b \leq p < (b + s_c + s_d)}{s \vdash \texttt{protected}(p)}$$

(Aux-unprotected1)
$$\frac{s \equiv (b, s_c, s_d, n) \qquad p < b}{s \vdash \texttt{unprotected}(p)}$$

(Aux-unprotected2)
$$\frac{s \equiv (b, s_c, s_d, n) \qquad (b + s_c + s_d) < p}{s \vdash \texttt{unprotected}(p)}$$

(Aux-entryPoint)
$$\frac{s \equiv (b, s_c, s_d, n) \qquad p = b + m * 128 \qquad m \in \mathbb{N} \qquad m < n}{s \vdash \texttt{entryPoint}(p)}$$

(Aux-data)
$$\frac{s \equiv (b, s_c, s_d, n) \qquad (b + s_c) \leq p \qquad p < (b + s_c + s_d)}{s \vdash \texttt{data}(p)}$$

(Aux-returnEntry)
$$\frac{s \equiv (b, s_c, s_d, n) \qquad p = b + (n - 1) * 128}{s \vdash \texttt{returnEntryPoint}(p)}$$

(Aux-read-1)
$$\frac{s \vdash \texttt{protected}(p)}{s \vdash \texttt{readAllowed}(p, a)}$$

(Aux-read-2)
$$\frac{s \vdash \texttt{unprotected}(p) \qquad s \vdash \texttt{unprotected}(a)}{s \vdash \texttt{readAllowed}(p, a)}$$

(Aux-write-1)
$$\frac{s \vdash \texttt{unprotected}(a)}{s \vdash \texttt{writeAllowed}(p, a)}$$

(Aux-write-2)
$$\frac{s \vdash \texttt{protected}(p) \qquad s \vdash \texttt{data}(a)}{s \vdash \texttt{writeAllowed}(p, a)}$$

(Aux-entry)
$$\frac{s \vdash \texttt{unprotected}(p) \qquad s \vdash \texttt{entryPoint}(p')}{s \vdash \texttt{entryJump}(p, p')}$$

(Aux-internal-1)
$$\frac{s \vdash \texttt{unprotected}(p) \qquad s \vdash \texttt{unprotected}(p')}{s \vdash \texttt{externalJump}(p, p')}$$

(Aux-internal-2)
$$\frac{s \vdash \texttt{protected}(p) \qquad s \vdash \texttt{protected}(p')}{s \vdash \texttt{internalJump}(p, p')}$$

(Aux-return)
$$\frac{s \vdash \texttt{protected}(p) \qquad s \vdash \texttt{unprotected}(p')}{s \vdash \texttt{exitJump}(p, p')}$$

(Aux-vj-intern)
$$\frac{s \vdash \texttt{internalJump}(p, p')}{s \vdash \texttt{validJump}(p, p')}$$

(Aux-vj-extern)
$$\frac{s \vdash \texttt{externalJump}(p, p')}{s \vdash \texttt{validJump}(p, p')}$$

(Aux-vj-return)
$$\frac{s \vdash \texttt{exitJump}(p, p')}{s \vdash \texttt{validJump}(p, p')}$$

(Aux-vj-entry)
$$\frac{s \vdash \texttt{entryJump}(p, p')}{s \vdash \texttt{validJump}(p, p')}$$

Fig. 9: Auxiliary functions for the low-level language semantics.

$$\begin{aligned} \Psi_0 = (p_0, r_0, f_0, m, s) \qquad \textit{where} \qquad s &= (b, s_c, s_d, n) \\ p_0 &= (b + s_c + s_d)\%2^{32} \\ r_0 &= [\mathsf{SP} \mapsto p_0; \mathsf{R_i} \mapsto 0]\ i = 0..11 \\ f_0 &= [\mathsf{ZF} \mapsto 0; \mathsf{SF} \mapsto 0]. \end{aligned}$$

The evaluation of a low-level program is a sequence of steps that take an initial configuration to a final configuration: $(p_0, r_0, f_0, m, s) \to^* (-1, r, f, m', s)$, with the result of the computation stored in $\mathsf{R_0}$. Notice that in order to capture termination, the program counter is set to -1 whenever the `halt` instruction is encountered, in that way no rule for further progress can ever be applied as $m'(-1)$ would not return a valid operation.

3.3 Trace Semantics

A different syntactic category (Λ) is used to denote low-level traces in order to be able to differentiate immediately whether the focus is on high-level or low-level traces. Labels exhibited by the lo-level trace semantics are defined according to

(Eval-movl)
$$\dfrac{\begin{array}{c} m(p) \cong (\texttt{movl } \texttt{r}_\texttt{d}\ \texttt{r}_\texttt{s}) \\ s \vdash \texttt{validJump}(p, p+1) \\ s \vdash \texttt{readAllowed}(p, *r_s) \\ r' = r[r_d \mapsto *r_s] \end{array}}{(p, r, f, m, s) \twoheadrightarrow (p+1, r', f, m, s)}$$

(Eval-movs)
$$\dfrac{\begin{array}{c} m(p) \cong (\texttt{movs } \texttt{r}_\texttt{d}\ \texttt{r}_\texttt{s}) \\ s \vdash \texttt{validJump}(p, p+1) \\ s \vdash \texttt{writeAllowed}(p, *r_d) \\ m' = m[*r_d \mapsto r_s] \end{array}}{(p, r, f, m, s) \twoheadrightarrow (p+1, r, f, m', s)}$$

(Eval-movi)
$$\dfrac{\begin{array}{c} m(p) \cong (\texttt{movi } \texttt{r}_\texttt{d}\ \texttt{i}) \\ s \vdash \texttt{validJump}(p, p+1) \\ r' = r[r_d \mapsto i] \end{array}}{(p, r, f, m, s) \twoheadrightarrow (p+1, r', f, m, s)}$$

(Eval-call)
$$\dfrac{\begin{array}{c} m(p) \cong (\texttt{call } \texttt{r}_\texttt{d}) \qquad p' = *r_d \\ s \vdash \texttt{validJump}(p, p') \\ sp_n = \mathsf{SP} - 1 \qquad r' = r[\mathsf{SP} \mapsto sp_n] \\ p_r = p + 1 \qquad m' = m[sp_n \mapsto p_r] \end{array}}{(p, r, f, m, s) \twoheadrightarrow (p', r', f, m', s)}$$

(Eval-add)
$$\dfrac{\begin{array}{c} m(p) \cong (\texttt{add } \texttt{r}_\texttt{d}\ \texttt{r}_\texttt{s}) \\ s \vdash \texttt{validJump}(p, p+1) \\ v = (r_d + r_s)\%2^{32} \\ f' = f[\mathsf{ZF} \mapsto (v == 0)] \\ r' = r[r_d \mapsto v] \end{array}}{(p, r, f, m, s) \twoheadrightarrow (p+1, r', f', m, s)}$$

(Eval-sub)
$$\dfrac{\begin{array}{c} m(p) \cong (\texttt{sub } \texttt{r}_\texttt{d}\ \texttt{r}_\texttt{s}) \\ s \vdash \texttt{validJump}(p, p+1) \\ v = (r_d - r_s)\%2^{32} \\ f' = f[\mathsf{ZF} \mapsto (v == 0)] \\ r' = r[r_d \mapsto v] \end{array}}{(p, r, f, m, s) \twoheadrightarrow (p+1, r', f', m, s)}$$

(Eval-compare)
$$\dfrac{\begin{array}{c} m(p) \cong (\texttt{cmp } \texttt{r}_1\ \texttt{r}_2) \\ s \vdash \texttt{validJump}(p, p+1) \\ f' = f[\mathsf{ZF} \mapsto (r_1 == r_2); \mathsf{SF} \mapsto (r_1 < r_2)] \end{array}}{(p, r, f, m, s) \twoheadrightarrow (p+1, r, f', m, s)}$$

(Eval-jump)
$$\dfrac{\begin{array}{c} m(p) \cong (\texttt{jmp } \texttt{r}_\texttt{d}) \qquad p' = r_d \\ s \vdash \texttt{validJump}(p, p') \end{array}}{(p, r, f, m, s) \twoheadrightarrow (p', r, f, m, s)}$$

(Eval-je-true)
$$\dfrac{\begin{array}{c} m(p) \cong (\texttt{je } \texttt{r}_\texttt{i}) \qquad f(\mathsf{ZF}) == 1 \\ p' = r_i \qquad s \vdash \texttt{validJump}(p, p') \end{array}}{(p, r, f, m, s) \twoheadrightarrow (p', r, f, m, s)}$$

(Eval-je-false)
$$\dfrac{\begin{array}{c} m(p) \cong (\texttt{je } \texttt{r}_\texttt{i}) \qquad f(\mathsf{ZF}) \equiv 0 \\ s \vdash \texttt{validJump}(p, p+1) \end{array}}{(p, r, f, m, s) \twoheadrightarrow (p+1, r, f, m, s)}$$

(Eval-jl-true)
$$\dfrac{\begin{array}{c} m(p) \cong (\texttt{jl } \texttt{r}_\texttt{i}) \qquad f(\mathsf{SF}) \equiv 1 \\ p' = *r_i \qquad s \vdash \texttt{validJump}(p, p') \end{array}}{(p, r, f, m, s) \twoheadrightarrow (p', r, f, m, s)}$$

(Eval-jl-false)
$$\dfrac{\begin{array}{c} m(p) \cong (\texttt{jl } \texttt{r}_\texttt{i}) \qquad f(\mathsf{SF}) == 0 \\ s \vdash \texttt{validJump}(p, p+1) \end{array}}{(p, r, f, m, s) \twoheadrightarrow (p+1, r, f, m, s)}$$

(Eval-ret)
$$\dfrac{\begin{array}{c} m(p) \cong (\texttt{ret}) \qquad p' = m(\mathsf{SP}) \\ s \vdash \texttt{validJump}(p, p') \\ sp_n = \mathsf{SP} + 1 \qquad r' = r[\mathsf{SP} \mapsto sp_n] \end{array}}{(p, r, f, m, s) \twoheadrightarrow (p', r', f, m, s)}$$

(Eval-halt)
$$\dfrac{m(p) \cong (\texttt{halt})}{(p, r, f, m, s) \twoheadrightarrow (-1, r, f, m, s)}$$

Fig. 10: Dynamic semantics for the low-level language.

Figure 11. Low-level execution states for the trace semantics, still denoted with Ψ, do not deal with the whole memory but just with a subset of it.

Figure 12 presents the rules that define the relation $\Psi \xRightarrow{a} \Psi'$. Assume the following convention on the usage of registers: R_4 is used to identify the current object on a method call, R_5 to R_{11} are used for parameters. The trace semantics

$$\Lambda ::= \alpha \mid \tau_e \mid \tau_i \qquad \alpha ::= \gamma? \mid \gamma! \qquad \gamma ::= \texttt{call}\ a(\overline{v}) \mid \texttt{ret}\ v$$

Fig. 11: Labels for the trace semantics of the low-level language.

$$\text{(Trace-external)}\ \dfrac{s \vdash \texttt{externalJump}(p, p')}{(p, r, f, m, s) \overset{\tau_e}{\twoheadrightarrow} (p', r', f', m, s)} \qquad \text{(Trace-internal)}\ \dfrac{(p, r, f, m, s) \twoheadrightarrow (p', r', f', m', s) \quad s \vdash \texttt{internalJump}(p, p')}{(p, r, f, m, s) \overset{\tau_i}{\twoheadrightarrow} (p', r', f', m', s)}$$

$$\text{(Trace-call)}\ \dfrac{s \vdash \texttt{entryJump}(p, p') \quad v = \mathsf{R}_4 \quad \overline{v} = \mathsf{R}_5 :: \ldots :: \mathsf{R}_{11}}{(p, r, f, m, s) \overset{\texttt{call}\ p'(v,\overline{v})?}{\twoheadrightarrow} (p', r', f', m, s)} \qquad \text{(Trace-returnback)}\ \dfrac{s \vdash \texttt{entryJump}(p, p') \quad s \vdash \texttt{returnEntryPoint}(p') \quad v = \mathsf{R}_0}{(p, r, f, m, s) \overset{\texttt{ret}\ v?}{\twoheadrightarrow} (p', r', f', m, s)}$$

$$\text{(Trace-callback)}\ \dfrac{(p, r, f, m, s) \twoheadrightarrow (p', r', f', m', s) \quad s \vdash \texttt{exitJump}(p, p') \quad m(p) \cong (\texttt{jmp}) \quad v = \mathsf{R}_4 \quad \overline{v} = \mathsf{R}_5 :: \ldots :: \mathsf{R}_{11}}{(p, r, f, m, s) \overset{\texttt{call}\ p'(v,\overline{v})!}{\twoheadrightarrow} (p', r', f', m', s)}$$

$$\text{(Trace-return)}\ \dfrac{(p, r, f, m, s) \twoheadrightarrow (p', r', f', m', s) \quad s \vdash \texttt{exitJump}(p, p') \quad m(p) \cong (\texttt{ret}) \quad v = \mathsf{R}_0}{(p, r, f, m, s) \overset{\texttt{ret}\ v!}{\twoheadrightarrow} (p', r', f', m', s)}$$

$$\text{(Trace-refl)}\ \dfrac{}{\Psi \overset{\epsilon}{\Longrightarrow} \Psi} \qquad \text{(Trace-trans)}\ \dfrac{\Psi \overset{\overline{\alpha}}{\Longrightarrow} \Psi'' \quad \Psi'' \overset{\overline{\alpha}'}{\Longrightarrow} \Psi'}{\Psi \overset{\overline{\alpha\alpha}'}{\Longrightarrow} \Psi'} \qquad \text{(Trace-tau-e)}\ \dfrac{\Psi \overset{\tau_e}{\twoheadrightarrow} \Psi'}{\Psi \overset{\epsilon}{\Longrightarrow} \Psi'} \qquad \text{(Trace-tau-i)}\ \dfrac{\Psi \overset{\tau_i}{\twoheadrightarrow} \Psi'}{\Psi \overset{\epsilon}{\Longrightarrow} \Psi'} \qquad \text{(Trace-action)}\ \dfrac{\Psi \overset{\alpha}{\twoheadrightarrow} \Psi'}{\Psi \overset{\alpha}{\Longrightarrow} \Psi'}$$

Fig. 12: Trace semantics for the low-level language.

of a low-level program (m_{sec}, s) is defined as follows:

$$\mathsf{Traces_L}(m_{sec}, s) = \{\overline{\alpha} \mid (p_0, r_0, f_0, m_{sec}, s) \overset{\overline{\alpha}}{\Longrightarrow} \Psi'\}.$$

Since memory allocation is deterministic, low-level traces do not need α-equivalence. If two low-level traces are syntactically different, they are semantically different.

The trace semantics is compositional to the trace semantics that models execution from the unprotected program perspective. The trace semantics of unprotected programs can be obtained from the rules of Figure 12 by swapping decorations ! and ? on traces, eliminating assumptions of the form $\Psi \twoheadrightarrow \Psi'$ where present and adding assumptions of the form $\Psi \twoheadrightarrow \Psi'$ where missing.

3.4 A Fully Abstract Trace Semantics

A trace semantics that captures all communicated information on the labels of the traces is equivalent to the notion of contextual equivalence [18]. Such a trace semantics is called *fully abstract*, and it can always be used where contextual equivalence is required. A fully abstract trace semantics simplifies the proof of full abstraction of a compilation scheme, as Section 4.2 explains.

While the trace semantics of the low-level language captures the same actions of the trace semantic of the high-level language, it is not fully abstract. In fact, it does not capture all communicated information on the labels of the trace. For example, low-level programs can read and write outside the protected memory. Consider two low-level programs M_1 and M_2 that exhibit the same trace semantics. If M_1 performs a write in unprotected memory but M_2 does not, an external program is able to tell whether it is interacting with M_1 or M_2 by monitoring if unprotected memory changes. Similarly, flags and unused registers can be used to tell M_1 or M_2 apart.

In order to make the trace semantics fully abstract, two approaches exist: augmenting the expressivity of the labels or changing the semantics of the language [7]. Since the low-level language is used as the output language of a compilation scheme, we choose the second option.

The compilation scheme presented in Section 5 produces compiled components, denoted as $C^{\downarrow}$, whose trace semantics is fully abstract. Compiled components never read or write unprotected memory. Moreover, whenever a label is generated, flags and unused registers that are not used to convey information, such as the value of a parameter of a function call, are reset to a default value. Compiled components are also never supposed to execute the `halt` instruction. For the sake of simplicity, we omit this feature in the presentation of the main result and consider this trivial addition in Section 7.

Proposition 1 (Fully abstract low-level trace semantics modulo compilation). *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 compilation scheme of Section 5, we have that:* $\mathsf{Traces}_{\mathsf{L}}(C_1^{\downarrow}) = \mathsf{Traces}_{\mathsf{L}}(C_2^{\downarrow}) \iff C_1^{\downarrow} \simeq C_2^{\downarrow}$.

Proposition 1 drives the proof strategy presented in Section 6. This strategy was inspired by the work of Agten *et al.* [3] and is classical in the field [8,11]. Future work will present the result of Proposition 1 in more details. We now drop the terminology fully abstract trace semantics to avoid confusion with the main result: a fully abstract compilation scheme.

4 Informal Overview

This section briefly presents contextual equivalence as a good candidate for indicating the preservation of security properties. Then, this section provides an informal description of the proof strategy for the main result and an informal overview of the compilation scheme.

4.1 Contextual Equivalence: a Security Perspective

Contextual equivalence is widely accepted in the field of secure compilation, its definition is included in order to familiarise the reader with it [1,2,3,4,9,10,13].

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 [18]. Contexts in Java Jr. are components [12].

Contextual equivalence can be adopted to state security properties such as confidentiality, integrity and invariant definition, as in Figure 13 [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 Figure 13 differ in the value stored in `secret` after a

```
package p;
class C1 {
  secret, min, max:Int=0;

  public m1():Int {
    secret = 0;
    return 0;
  }
  public m2(arg:D):Int {
    secret = 0;
    arg.cb();
    if (secret==0) { return 0; }
    return 1;
  }
  public m3():Int {
    if (min ≤ max) { return 0; }
    return 1;
  }
}
```

```
package p;
class C2 {
  secret, min, max:Int=0;

  public m1():Int {
    secret = 1;
    return 0;
  }
  public m2(arg:D):Int {
    secret = 0;
    arg.cb();
    return 0;

  }
  public m3():Int {
    return 0;

  }
}
```

Fig. 13: 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, $\mathtt{C}_1$ checks whether changes to `secret` have been made during the call to `arg.cb( )` in method `m2`. If $\mathtt{C}_1$ and $\mathtt{C}_2$ 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 $\mathtt{C}_1$ will return 1. If $\mathtt{C}_1$ and $\mathtt{C}_2$ 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 [21].

4.2 Proving Full Abstraction of a Compilation Scheme, Informally

The main result proven in Section 6 is that $C_1 \simeq C_2 \iff C_1^{\downarrow} \simeq C_2^{\downarrow}$. 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, even certifying the result [5,14]; we are not interested in this direction. This is thus assumed, the implications of this assumptions are made explicit in Section 6.3. 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 thus they do not provide any inductive structure. To avoid working with contexts, we exploit Proposition 1 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 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 classical yet complex when the high-level language is strongly typed or it includes features such as dynamic memory allocation [3,8,11].

4.3 Achieving Full Abstraction of a Compilation Scheme, Informally

Let us now informally describe how a compilation scheme can be modified so that it becomes fully abstract. Find a detailed treatment of this topic in Section 5.

Assume that a correct compiler from Java Jr. to the presented machine code is given. Thus a component C and its compiled counterpart $C^{\downarrow}$ behave the same, from a high-level perspective. For the compilation scheme to be fully abstract, the information $C^{\downarrow}$ and external low-level code M exchange must be the same as that exchanged between C and external high-level components. But this is not always the case. In fact, external low-level code can perform method calls on objects of the wrong type and with ill-typed parameters. Consider a `Key` class defining a secret as its first field and another class `Pair` implementing pairs with methods `getFst()` and `getSnd()`. Assuming fields are accessed via offsets, external low-level code could invoke the `getFst()` method on an object of type `Key` and obtain the secret. Similarly, performing a method call with ill-typed parameters leads to the same exploit. Moreover, as low-level object identifiers are the addresses where objects are allocated, communicating them leaks too much information. An attacker could learn about the compiler allocation scheme or the size of an object, then it could call methods on objects just by guessing their address.

For the compilation scheme to be fully abstract, $C^{\downarrow}$ and M must communicate only via well-typed method calls, communicated values must be of the right type and object identifiers must be masked. In order to achieve this, $C^{\downarrow}$ is placed in the data section of the protected memory. For each method defined in an interface of C, a corresponding entry point is created in the code section of the protected memory. Code at each entry point acts as a proxy to the actual method implementation.

For method calls to be well-typed, the code at entry points must check that a method is invoked on objects of the right type, with parameters of the right type. Similar checks need to be executed also when returning from a callback, so an entry point for callbacks is provided. These checks can be performed only on objects whose class is defined in C since they are allocated in protected memory; no control over externally allocated objects can be assumed.

To mask low-level object identifiers, a data structure $\mathcal{O}$ is created; it is a map between low-level object identifiers and natural numbers. Low-level object identifiers that are passed to external code are added to $\mathcal{O}$ right before the identifier is passed. The index in the data structure is then passed in place of the object identifier, the same index is passed whenever the same object should be. The code at entry points is responsible of retrieving object identifiers 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 identifiers. Moreover, indices in $\mathcal{O}$ are leaked in a deterministic order, based on the interaction between external and internal code. So the security of the compilation scheme scales to real-world scenarios where objects are allocated at different addresses in different runs. For the sake of simplicity, the main result (Theorem 1 in Section 6.3) does not treat allocation at different addresses directly.

5 Compilation Scheme

This section presents the assumptions made by the high-level language, followed by the compilation scheme from high- to low-level code. Here the word *component* refers to the Java Jr. component that is being compiled into low-level code.

Communication with external code. Assume the component being compiled provides one import package without a corresponding export one. Refer to this package as *the import package*. The import package contains interface and extern definitions that specify how the component interacts with external code. Callbacks from the component are invocations on methods defined in the import package. External code will provide an implementation of the import package for callbacks. Listing 1.2 provides an example of an import package.

```
package p_imp;
interface Bar extends Baz {
  public createBar() : Bar;
  public callback( arg : Bar ) : Unit;
}
interface Baz {
  public lock() : Int;
}
extern extBar : Bar;
```

Listing 1.2: Example of an import package.

In order to implement package p_{imp}, there needs to be at least one class implementing `Bar` and an object named `extBar` that is subtype of `Bar`.

Component code is assumed not to implement interfaces defined in the import package, while code in the import package can implement interfaces defined in the component. This affects the dynamic dispatch, details are discussed later.

5.1 Compilation Scheme

The compilation scheme extends that of Agten *et al.* [3], thus it shares some points with its predecessor. However, the compilation scheme needs to handle several subtleties that its predecessor does not. For example, it needs to mask object identifiers through $\mathcal{O}$, take care of the current object and perform the checks mentioned in Section 4.3. These additions stem from the fact that Java Jr. features dynamic memory allocation. Such additions force the compilation scheme to face the subtleties of additional attacks as exemplified in Section 4.3, while its predecessor did not. This makes the presented compilation scheme a novel contribution and a significant improvement over its predecessor.

General points [3]. Compiled components must be indistinguishable from the size point of view, thus a constant amount of space is reserved for each compiled package, independent of its implementation.

All packages, as well as interfaces declared in those packages and methods declared in those interfaces, are sorted alphabetically. This makes compiled components impossible to be distinguished based on the ordering of low-level method calls. Methods and fields are then given a unique index, starting from 0, based on their order of occurrence. Those indexes serve as the offset that is used to access methods and fields. Parameters and local variables are also given a method-local index to be used as above. Methods that are not inherited from implemented interfaces are treated classically [6].

The program counter is initialised to a given address in unprotected memory.

Code compilation. Figure 14 shows a graphical representation of the protected memory section which is generated by the compilation of the 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, which are described in the following paragraphs. The protected data section contains the v-tables, method body implementations, a procedure for object allocation, a secure stack and a secure heap. The v-tables are data structures used to perform the dynamic dispatch of methods. They return the address of the method to be executed on the current object based on its type and the method name. 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 by adding extra checks or by implementing object wrappers. For each method body, a prologue and an epilogue, responsible of allocating and deallocating activation records on the secure stack, are appended to it. This code is not located at the entry point. Expressions are compiled in the usual fashion [6].

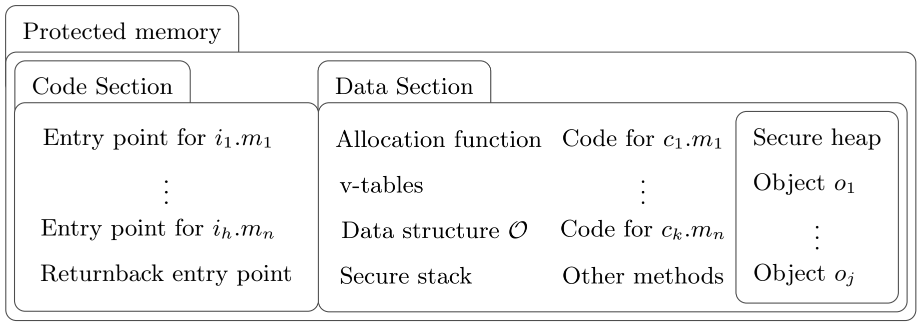

Fig. 14: Graphical representation of a compiled component.

Object identifiers. Assume that, for a given component, static objects are always compiled to the same address. Thus their low-level identifiers and their high-level ones are always the same across different runs. Since compilation is deterministic, dynamically allocated objects will always be placed at the same address, given the same external program, for the same component under test.

As motivated in Section 2.1, an object identifier is never passed in clear as a return value from a method call or as a parameter of a callback. Instead, it is added to the data structure $\mathcal{O}$ and its index in $\mathcal{O}$ is returned just before being sent to the external code. The order in which identifiers are added to $\mathcal{O}$ is the order of exposure to the outer world and not the allocation order.

Registers and stack conventions. Registers R_0 to R_3 are used as working registers for low-level instructions. Register R_4 is used to identify the current object in a method call. Before a callback, R_4 is stored in the secure stack on order to restore the current object to the right value once the callback returns. Registers R_5 to R_{11} are used for parameters (for the sake of simplicity the amount of parameters of a method has a maximum of seven).

The stack is split in a protected and an unprotected one. At each entry point, the protected stack is set as the active one. When leaving the protected section the unprotected stack is set to be the active one. In order to implement stack switches, a shadow stack pointer is introduced, which points to the base of the protected stack. To prevent tampering with the control flow, the base of the protected stack points to a procedure that writes 0 in R_0 and `halt`s. This prevents jumping to the returnback entry points when no callback was made [3].

Entry points. As presented in Section 2.1, Java Jr. enforces a "programming to an interface" style. For the compilation scheme, this means creating *method entry points* in protected memory for all interface-declared methods. A *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 checks and then jumps either to the code that performs the dynamic dispatch or to the callback. The second part returns control

Table 1: Pseudo code executed at entry points.

Method p entry point	Returnback entry point
Load current object $v = \mathcal{O}(\mathsf{R}_4)$	Push current object $v = \mathsf{R}_4$
Check that v has method p	Push return address a
Load parameters $\overline{v}$	Push return type m
Check types of $\overline{v}$ for p	Reset flags and unused registers
`Unit`-typed value checks	Replace object identifiers with index in $\mathcal{O}$
Call dynamic dispatch	`jmp` to callback address
Exit point	Re-entry point
Reset flags and unused registers	Pop return type m and check it
Replace object identifiers with index in $\mathcal{O}$	`Unit`-typed value checks
	Pop return address a, current object v and resume execution

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.

Let us explain the terminology of Table 1. Loading means that a value is retrieved from the memory, push and pop are operations on the secure stack. Resetting flags and registers means setting flags and registers besides R_0 and parameters to 0. `Unit`-typed value checks indicate that `Unit`-typed parameters must have value 0 [3]. Should any check fail, all registers and flags are cleared and the execution `halt`s.

Similar checks are needed in case Java Jr. is extended with other ground types. For example, to add `boolean`s, `bool`-typed parameters must have either value 1 or 0, which correspond to the high-level values `true` or `false`.

6 Full Abstraction of the Compilation Scheme

This section presents a proof of the main result: full abstraction of the compilation scheme. Section 6.1 presents the algorithm mentioned in Section 4.2 through a series of examples, Section 6.2 provides a transliteration of the algorithm and Section 6.3 contains the proofs.

6.1 Algorithm

This section describes the algorithm mentioned in Section 4.2. 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 external memory. The algorithm outputs a high-level component C that differentiates between C_1 and C_2, called the *output component*. Instead of presenting a transliteration of the pseudo-code of the algorithm, which would be difficult to understand, this section presents several examples of what the expected output of the algorithm is in different cases. The examples illustrate crucial cases the algorithm needs to consider when creating the output component. The algorithm has been implemented in Scala, and it interacts with Java components that adhere to the Java Jr. formalisation.[1]

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 produced is given the power to end the computation via the `exit` statement to avoid being trapped in infinite loops.

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 names 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 identifiers of static objects. Then, a code skeleton for the output component is created, based on the structure of the import package of C_1 and C_2.

For all interfaces i defined in the import package, 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:

```
public defaultCreate() : i_c { return new i_c (); }
```

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 will 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 and C_2, the low-level traces they generate and the output the algorithm

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

produces after being run. Components C_1 and C_2 are modifications of the code in Listing 1.1, whose import package is assumed to be defined in Listing 1.2. C_1 is presented on the left while C_2 stands on the right. Omitted code is the same in both components and can be found in Listing 1.1. The examples also present what the algorithm must do in order to create the correct output.

The reported low-level traces are massaged for better understanding. For example, given that object `extFoo` is compiled to address 0x123 and that method `createFoo` is associated to address 0x456, the label `call` 0x456(0x123) is written as `extFoo.createFoo()`. Numbers in italic font, e.g. *1*, refer to indexes from $\mathcal{O}$, while identifiers of externally allocated objects are numbers in hexadecimal.

Example 2 (Different returned values). Consider the following implementations for C_1 and C_2. Each code fragment is followed by the low-level trace it generates.

```
private extFoo : FooClass {
  counter = 1
}
public getCounter() : Int {
  return counter;
}
```

```
private extFoo : FooClass {
  counter = 0
}
public getCounter() : Int {
  return counter += 1;
}
```

$$\alpha_1 = \texttt{extFoo.getCounter()}?\ \texttt{ret 1}!\ \texttt{extFoo.getCounter()}?\ \texttt{ret 1}!$$

$$\alpha_2 = \texttt{extFoo.getCounter()}?\ \texttt{ret 1}!\ \texttt{extFoo.getCounter()}?\ \texttt{ret 2}!$$

In this example, the produced code needs to differentiate between C_1 and C_2 based on the type of expected returned values. These types can be either: ground, internal, external. With ground-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. Since `oc` is static, its fields are global variables for the output component.

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

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

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 ground-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).

```
public createFoo() : Foo {
 extBar.lock();
}
```

```
public createFoo() : Foo {
 Bar b = extBar.createBar();
}
```

$$\alpha_1 = \texttt{extFoo.createFoo()}?\ \texttt{extBar.lock()}!$$
$$\alpha_2 = \texttt{extFoo.createFoo()}?\ \texttt{extBar.createBar()}!$$

In this example, C_1 performs a callback on method `lock`, while C_2 performs it on method `createBar`.

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 following code is produced by the algorithm for this example:

```
 1 public main ( args : String[] ) : Unit {
 2  if ( oc.getStep() == 0 ) {
 3    oc.incrementStep();
 4    Foo f = extFoo.createFoo();
 5  }
 6 }
 7 public createBar() : Bar {
 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 3 and 7, whose addition was discussed in Example 2, help the produced code achieve differentiation in this case as well. Should methods `createBar()` 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).

```
public createFoo() : Foo {
  return this;
}
```

```
public createFoo() : Foo {
  return new FooClass();
}
```

$\alpha_1 = \texttt{extFoo.createFoo()}?\ \texttt{ret extFoo}!$ $\alpha_2 = \texttt{extFoo.createFoo()}?\ \texttt{ret}\ \mathit{1}!$

In this case the produced code needs to 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 static object: `extFoo`, while C_2 returns a new object whose index in $\mathcal{O}$ is *1*.

To achieve differentiation in this case, the produced code needs to keep track of internally allocated objects. For this it relies on a list `internals` provided by `oc`. In order for internal objects to be accessible, they are wrapped with a new class: `Internal` that has two fields. The first, of type `Obj`, contains a reference to an internal object. The second, `name`, can be used to filter the search for objects. No two objects with the same `name` can be added to `internals`. 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 identifier 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 following code is produced by the algorithm for this example:

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

Line 5 has no effect, since `internals` already has an entry for `extFoo`. 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).

```
public createFoo() : Foo {
  Bar b = extBar.createBar();
  b = b.createBar();
}
```

```
public createFoo() : Foo {
  Bar b = extBar.createBar();
  b = extBar.createBar();
}
```

$\alpha_1 = \texttt{extFoo.createFoo()}?\ \texttt{extBar.createBar()}!\ \texttt{ret 0x6}?\ \texttt{0x6.createBar()}!$

$\alpha_2 = \texttt{extFoo.createFoo()}?\ \texttt{extBar.createBar()}!\ \texttt{ret 0x6}?\ \texttt{extBar.createBar()}!$

In this case the difference is the external object on which a callback is performed. Here, C_1 calls `createBar()` on `0x6`, while C_2 calls the same method on `extBar`.

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 following code is produced by the algorithm for this example:

```
 1 // same main as in Example 3
 2 public createBar() : Bar {
 3   if ( oc.getStep() == 1 ) {
 4     oc.incrementStep();
 5   }
 6   if ( oc.getStep() == 2 ) {
 7     oc.incrementStep();
 8     Bar h = oc.getExternal( "0x6", "Bar" );
 9     if ( h == null ) {
10       h = staticForBar.defaultCreate();
11       ( ( Listable ) h ).setAndRegister( "0x6", "Bar" );
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.

Example 6 (Different externally-typed parameter of a callback).

```
public createFoo() : Foo {
  Bar b = extBar.createBar();
  b.callback( b );
}
```

```
public createFoo() : Foo {
  Bar b = extBar.createBar();
  b.callback( extBar );
}
```

$\alpha_1 =$ `extFoo.createFoo()`? `extBar.createBar()`! `ret 0x6`? `0x6.callback( 0x6 )`!

$\alpha_2 =$ `extFoo.createFoo()`? `extBar.createBar()`! `ret 0x6`? `0x6.callback( extBar )`!

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 following code is produced by the algorithm for this example:

```
// same main and createBar from Example 5,
// except that lines 20 - 22 are removed
public callback( arg : Bar ) : Unit {
  if ( oc.getStep() == 3 ) {
    if ( ( ( Listable ) arg ).getName() == "0x6" ) { exit(1); }
    else { exit(2); }
  }
}
```

Casting `arg` to `Listable` is needed in order to make sure the call to `getName()` succeeds. In fact, `arg` is known to implement interface `Bar`, which has no connection with class `Listable` that defines method `getName()`.

Example 7 (Traces of different length). As mentioned in Section 3.4, compiled components cannot execute the `halt` instruction. Additionally, compiled components are the result of compiling well-typed, high-level components; well-typedness ensures that they cannot get stuck [17]. This means that there is only one way that $C_1^{\downarrow}$ and $C_2^{\downarrow}$ can create traces of different length: C_1 diverges and does not return control to external code while C_2 returns control to external code, or vice-versa. An example of such a behaviour is presented below.

```
public createFoo() : Foo {
  while ( 1 == 1 ) { skip; };
  return null;
}
```

```
public createFoo() : Foo {
  return new FooClass();
}
```

$\alpha_1 = \texttt{extFoo.createFoo()}?$ $\alpha_2 = \texttt{extFoo.createFoo()}?\ \texttt{ret}\ \mathit{1}!$

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. Here, differentiation is achieved in a classical sense, by diverging in any case and terminating in the other [18]. Previous examples did not adopt this approach since differentiation could be achieved in a simpler way.

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

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

When control is returned to `main` after a call to `createFoo()`, 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.

These examples highlighted the most peculiar difficulties the algorithm needs to face, which arise from the presence of dynamic memory allocation feature in Java Jr. For the algorithm to be correct, the code produced for action a must be proven to generate action α, and $\alpha \equiv_{\mathrm{a}} a$ in the sense of Definition 4 below. Moreover, the algorithm should be able to differentiate when two different actions are encountered. We do not provide a formal proof of the correctness of the algorithm, this fact can be seen from these examples or by consulting Section 6.2.

6.2 Algorithm Transliteration

This section describes in words and pseudo-code the algorithm of Section 6.1.

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 1.3 presents the skeleton code for the import package of Listing 1.2.

```
package p_t;
... // definition of Listable, Helper, Intern, Tester and object oc
class Bar_c extends Listable implements Bar {
  public createBar() : Bar {
    return null;
  }
  public callback( arg : Bar ) : Unit {
    return;
  }
  public defaultCreate() : Bar {
    if ( this == staticFor_Bar ){
      return new Bar_c();
    }
    return staticFor_Bar.defautCreate();
  }
}
class Baz_c extends Listable implements Baz {
  public lock() : Int {
    return 0;
  }
  public defaultCreate() : Baz {
    if ( this == staticFor_Baz ){
      return new Baz_c();
    }
    return staticFor_Baz.defautCreate();
  }
}
// fields come from class Listable
object staticFor_Bar : Bar_c { name = "staticFor_Bar", type = "Bar" }
object staticFor_Baz : Baz_c { name = "staticFor_Baz", type = "Baz" }
object extBar : Bar_c { name = "extBar", type = "Bar" }
```

Listing 1.3: 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:

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

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:

```
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.

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

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

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

```
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.

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

Then the following code is added:

```
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 $\alpha_1 = \mathtt{ret}(v_1)!$ *and* $\alpha_2 = \mathtt{call}\ a_2(v_2, \overline{v}_2)!$

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

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

Then this subroutine returns the code:

```
exit(1);
```

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

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

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

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

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

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

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 $c.m$:

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

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 $(v_j, \overline{t})$ is added to $\mathcal{AO}$, where $\overline{t}$ are the interfaces implemented by t_j.

```
oc.addInternal( $v_j$ , $x_j$ );
```

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.

```
if ( $x_j$ == $\uparrow(v_j, t_j)$ ) { exit(1); } else { exit(2); }
```

```
if ( oc.getNameByObject( $x_j$ ) == $v_j$ ) { exit(1); } else { exit(2); }
```

```
if ( ( ( Listable ) $x_j$).getName() == $v_j$ ) { 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 $c_1.m_1$:

```
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:

```
if ( oc.getStep() == $<i>$) {
  if ( ( Listable ) this ).getName() == $v_1$) { exit(1); } else { exit(2); }
}
```

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 `ret`$(v_1)!$. The subroutine returns the code:

```
exit(1);
```

callback `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.

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

6.3 Full Abstraction of the Compilation Scheme

This section introduces additional definitions and it provides an overview of the proof strategy. Finally, it states the main result: full abstraction of the compilation scheme (Theorem 2 below).

Definition 3 (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 $\Psi = (p, r, f, m, s)$ 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 4 (Action equivalence). *A high-level action a and a low-level action α are equivalent, denoted $a \equiv_{\mathrm{a}} \alpha$, given Σ and Ψ 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 $\Psi = (\ldots, m', s)$, 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 $\alpha = \texttt{ret}\ v_l?$ and $v_h \equiv_{\mathrm{v}} v_l$; or*
- *if $a = \overline{\texttt{new}\ v.}\ \texttt{return}\ v_h!$ and $\alpha = \texttt{ret}\ v_l!$ and $v_h \equiv_{\mathrm{v}} v_l$.*

Figure 15 contains a graphical representation of the proof scheme, where high- and low-level execution traces of two components are related. High-level traces model the interaction of a component with the output of the algorithm. Horizontal lines connect equivalent states and equivalent actions, the key shows what provides such an equivalence.

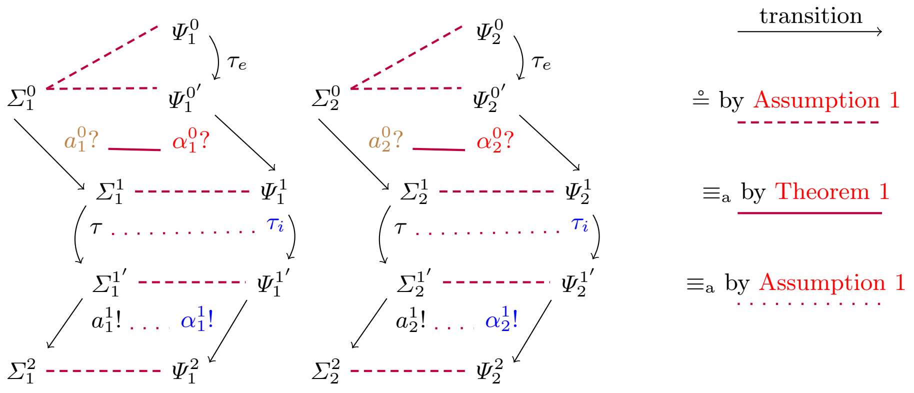

Fig. 15: Graphical representation of the proof scheme.

In Figure 15 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 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 is assumed to receive 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.

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 Σ and low-level states Ψ, denoted $\Sigma \doteq \Psi$, such that:*
  - *The initial high- and low-level states are equivalent. Formally: if $C^{\downarrow} = (m, s)$ then $(C \vdash \mathsf{blk} \rhd \epsilon : \epsilon) \doteq (p_0, r_0, f_0, m, s)$, and*
  - *Given two equivalent states and two corresponding internal transitions, the states these transitions lead to are equivalent. Moreover, given two equivalent states, given two equivalent actions, the states these transitions lead to are equivalent. Formally: if $\Sigma \doteq \Psi, \Sigma \xrightarrow{\tau} \Sigma' \xrightarrow{a} \Sigma'', \Psi \xrightarrow{\tau_i}\!\!\twoheadrightarrow^* \Psi' \xrightarrow{\alpha}\!\!\twoheadrightarrow \Psi''$, then $\Sigma' \doteq \Psi'$. Moreover, if $a \equiv_{\mathsf{a}} \alpha$ then $\Sigma'' \doteq \Psi''$.*

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

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.

$$\forall i \in \mathbb{N}, \alpha^{(2i)} \in \overline{\alpha} \Rightarrow \alpha^{(2i)} = \gamma? \text{ and } \alpha^{(2i+1)} \in \overline{\alpha} \Rightarrow \alpha^{(2i+1)} = \gamma!.$$

Proof. Straightforward induction on i. □

Theorem 1 (Algorithm correctness). *For any two high-level components C_1 and C_2 that exhibit two different low-level traces $\overline{\alpha_1}$ and $\overline{\alpha_2}$ when interacting with the same external code m, the algorithm of Section 6.1 outputs a component C that differentiates between C_1 and C_2.* $\mathsf{Traces_L}(C_1^{\downarrow}) \neq \mathsf{Traces_L}(C_2^{\downarrow}) \Rightarrow C_1 \not\simeq C_2$.

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 $\forall i \in \mathbb{N}, a_1^{(2i)} \equiv_{\mathsf{a}} \alpha_1^{(2i)}$; 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: $\forall i \in \mathbb{N}, a_1^{(2i+1)} \equiv_{\mathsf{a}} \alpha_1^{(1i+1)}$; 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 6.1 shows that. $\square$

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 and of the secure heap).* $C_1 \simeq C_2 \iff C_1^{\downarrow} \simeq C_2^{\downarrow}$.

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 $\mathsf{Traces_L}(C_1^{\downarrow}) \neq \mathsf{Traces_L}(C_2^{\downarrow}) \Rightarrow C_1 \not\simeq C_2$. Apply Theorem 1 to prove the statement. $\square$

7 Extensions

The following section informally describes how to include more powerful language constructs into the high level language and how to compile them. Then it describes how to allow compiled components to execute `halt` instructions.

Inner classes. An instance of an inner class C could access `private` fields of its enclosing instance D. To implement this, D is extended with static methods to access its private fields and code in C replaces fields access with calls to those methods. Since the newly created static methods are unavailable at the high level, their presence breaks full abstraction of a standard compilation scheme [1].

However, the presented compilation scheme is fully abstract even if Java Jr. is extended with inner classes. In fact, those additional static methods do not occur in the entry points, so external code does not have access to them. So, what was a breach in full abstraction for previous works [1] is not a breach here.

Cross-Package Class Inheritance. Consider class C extending class D when C is protected and D is not and vice-versa. In both cases, low-level instances of the unprotected class must not be able to access the state of the protected one. Moreover, `super` calls must be well-typed whenever such calls cross memory boundaries.

When instantiating class C, two objects are allocated, one for C and one for D, each object in the memory region where its class is defined. A field is added to C to refer to the superclass instance. Additional code is needed to ensure that the current object identifier is always the correct one. If C is protected and D is not, `super` calls are compiled as callbacks; otherwise, `super` calls are compiled as method calls. In the latter case, the compiled component must keep another data structure with entries of the form: (a_{sub}, a_{sup}), where a_{sub} is the address of the externally allocated sub-object and a_{sup} is the address of the corresponding internally allocated super-object. This prevents `super` calls to be invoked on objects of the wrong type. External code could supply any address as the current object in a `super` call, replicating the security exploits presented in Section 4.3.

Executing `halt` from compiled components. In order to allow compiled components to call to `halt`, the trace semantics presented in Section 3.3 needs to be expanded [7]. This extension merely influences the formalisation of the low-level trace semantics and the proof of full abstraction of the compilation scheme. The compilation scheme does not need to be changed.

Firstly, a stuck state needs to be defined. A state Ψ is stuck, denoted as $\Psi^{\perp}$, if the computation cannot proceed further; the state reached after the execution of `halt` is a stuck state. Then, traces need to highlight if the execution reached a stuck state. For this, an additional label is needed, thus the syntax of α is expanded as follows: $\alpha ::= \ldots \mid \surd$. Figure 16 presents rules for generating $\surd$ on labels.

$$\frac{s \vdash \mathtt{unprotected}(p) \qquad (p', r', f', m, s)^{\perp}}{(p, r, f, m, s) \overset{\surd}{\twoheadrightarrow} (p', r', f', m, s)}\ \text{(Trace-external-tick)}$$

$$\frac{(p, r, f, m, s) \twoheadrightarrow (p', r', f', m', s) \qquad s \vdash \mathtt{protected}(p) \qquad (p', r', f', m', s)^{\perp}}{(p, r, f, m, s) \overset{\surd}{\twoheadrightarrow} (p', r', f', m', s)}\ \text{(Trace-internal-tick)}$$

Fig. 16: Additional rules for the generation of $\surd$ on traces.

Since compiled components can call to `halt`, compiled components can evaluate to stuck states. Otherwise, due to Assumption 1, this could not be possible since high-level well-typed components cannot get stuck [17]. Example 8 below discusses how the algorithm can achieve differentiation in this case as well.

Example 8 (Compiled components executing `halt`*).*

```
public createFoo() : Foo {
  exit( 0 );
}
```

```
public createFoo() : Foo {
  return new FooClass();
}
```

$\alpha_1 =$ `extFoo.createFoo()`$?\ \surd$ $\qquad \alpha_2 =$`extFoo.createFoo()`$?$ `ret` $1!$

Should either C_1 or C_2 call `exit()`, control is not returned to the produced component since `exit()` is compiled to a `halt` instruction. In this case, differentiating between C_1 and C_2 cannot be done by terminating with two different results, it has to be achieved by termination in one case and divergence in the other as in Example 7. The component whose trace presents a $\surd$ sooner than the other trace will cause termination of the execution. The algorithm is thus responsible of diverging whenever an interaction with the other component is detected.

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

```
1 public main ( args : String[] ) : Unit {
2  if ( oc.getStep() == 0 ) {
3   oc.incrementStep();
4   Foo f = extFoo.createFoo();
5   while ( 1 == 1 ) { skip; }
6  }
7 }
```

The algorithm knows that C_1 will call `exit()` since α_1 has a $\surd$ at the end. When control is returned to `main` after a call to `createFoo()`, it means that the output component is interacting with C_2. In this case the produced code diverges via the expression in line 5.

Extension to the algorithm. If compiled components can call to `halt` the following case needs to be considered by the algorithm.

If only one of the two low-level action $\alpha_1(i)$ or $\alpha_2(i)$ is a $\surd$, then two cases arise (assume wlog that $\alpha_2(i) = \surd$ and $\alpha_1 \neq \surd$). 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:

```
while ( 1 == 1 ) { skip; }
```

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.

```
while ( 1 == 1 ) { skip; }
```

8 Related Work

This paper builds on the work of Agten *et al.* [3], where the same result is achieved for a simpler high-level language. The presented work adopts an object-oriented language with dynamic object allocation, which makes the result significantly harder to achieve. Moreover, the main result is proved to a higher degree of formality and precision. A detailed assessment of additional differences between the two works can be found throughout the paper.

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 [13], 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.* [10] 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.* [9] 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 protection mechanism is the type system.

A large amount of work on secure compilation involved the compilation of unsafe languages such as C. An extensive survey can be found in Younan *et al.* [21]. Instead of focussing on a fully abstract compilation, that research has been devoted to strengthening the security properties C offers.

A different area of research is devoted to providing security architectures with fine-grained low-level protection mechanisms. Different security architectures with access control mechanisms comparable to ours have been developed in the last years: TrustVisor [15], Flicker [16], Nizza [19] and SPMs [20]. The existence of working prototypes underlines the feasibility of this kind of approach to bring efficient and secure low-level memory access control in commodity hardware. No results comparable to ours has been proven for these systems.

9 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 to untyped machine code. Full abstraction of the compilation scheme is proven correct, guaranteeing preservation 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 concurrency and distribution primitives. Additionally, we plan to provide a fully abstract trace semantics for the low-level language.

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. Iain D. Craig. *The interpretation of object-oriented programming languages - updated to include C# (2. ed.)*. Springer, 2002.
7. Pierre-Louis Curien. Definability and full abstraction. *Electron. Notes Theor. Comput. Sci.*, 172:301–310, April 2007.
8. Frank S. de Boer, Marcello M. Bonsangue, Martin Steffen, and Erika Ábrahám. A fully abstract semantics for UML components. In *FMCO'04*, volume 3657 of *Lecture Notes in Computer Science*, pages 49–69. Springer-Verlag, 2005.
9. Cedric Fournet, Nikhil Swamy, Juan Chen, Pierre-Evariste Dagand, Pierre-Yves Strub, and Ben Livshits. Fully abstract compilation to JavaScript. Technical report, MSR, 2012.
10. Radha Jagadeesan, Corin Pitcher, Julian Rathke, and James Riely. Local memory via layout randomization. In *CSF '11*, pages 161–174. IEEE, 2011.
11. Alan Jeffrey and Julian Rathke. A fully abstract may testing semantics for concurrent objects. *Theor. Comput. Sci.*, 338(1-3):17–63, June 2005.
12. 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.
13. Andrew Kennedy. Securing the .NET programming model. *Theor. Comput. Sci.*, 364(3):311–317, November 2006.
14. Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4):363–446, December 2009.
15. 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.
16. 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.
17. Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of computer and system sciences*, 375:348–375, 1978.
18. Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.
19. 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.
20. Raoul Strackx, Frank Piessens, and Bart Preneel. Efficient isolation of trusted subsystems in embedded systems. In *SecureComm*, pages 344–361, 2010.
21. 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, June 2012.