CIL to Java-bytecode Translation for Static Analysis Leveraging

Pietro Ferrara
JuliaSoft SRL
Verona, Italy
pietro.ferrara@juliasoft.com

Agostino Cortesi
Università Ca' Foscari di Venezia
Venice, Italy
cortesi@unive.it

Fausto Spoto
Università di Verona
Verona, Italy
fausto.spoto@univr.it

ABSTRACT

A formal translation of CIL (*i.e.*, .Net) bytecode into Java bytecode is introduced and proved sound with respect to the language semantics. The resulting code is then analyzed with Julia, an industrial static analyzer of Java bytecode. The overall process of translation and analysis is fast, scales up to industrial programs, and introduces a negligible number of false alarms. The main result of this work is to leverage existing, mature, and sound analyzers for Java bytecode by applying them to the (translated) CIL bytecode.

1 INTRODUCTION

Static analysis infers, at compile-time, properties about the runtime behavior of computer programs. It allows one to verify, for instance, the absence of runtime errors or security breaches. Static analysis applies also to compiled code in assembly or bytecode format. This is particularly interesting for applications distributed on the Internet, or downloaded from public (and possibly unsafe) application repositories (*e.g.*, the Google Play Store), when the source code is not available, but the user would like to statically check some safety or security properties.

The analysis of Java bytecode for the Java Virtual Machine (from now on, JB) has a long research tradition and many analyzers exist [26]. Some analyses build on formal mathematical roots, such as abstract interpretation [11, 14, 15, 22, 25]. Moreover, JB makes the design of static analysis easier by requiring bytecode to be typecheckable [21] and without unsafe operations such as free pointer operations. On the contrary, CIL bytecode, that is, the compiled bytecode used for the .Net platform (from now on, just CIL) , has not received much attention from the static analysis community yet. Moreover, CIL can be used in an *unsafe* way, that is, allowing free pointer operations, which makes its static analysis harder. However, these operations are very often used in very controlled contexts, hence, in most cases, a static analyzer could possibly capture their actual behavior anyway.

Despite clear differences, JB and CIL share strong similarities, being both low-level object-oriented languages where objects are stored and shared in the heap. Hence, it is tempting to leverage mature existing static analyses and tools for JB by translating CIL into *equivalent* JB and running the tools on the latter. Obviously, this introduces issues about the exact meaning of *equivalence* between CIL and its translation into JB. Moreover, the translation should not introduce code artifacts that confuse the analyzer and should work on industrial-size CIL applications, supporting as many unsafe pointer operations as possible.

1.1 Contribution

The main contribution of this work is the introduction of a translation of CIL to JB that is (i) theoretically sound, and (ii) effective in practice, so that an industrial static analyzer for JB can be applied to .Net (and in particular C#) programs. More languages compile into JB (*e.g.*, Scala) and CIL (*e.g.*, VB.Net and F#), with distinct features and code structures. Here, we focus on Java and C#, that have similar structure and compile into comparable bytecode.

We start by formalizing the concrete semantics of a representative subset of CIL and JB, and the translation of CIL into JB. Then, we prove this translation sound, that is, the concrete semantics of the initial CIL program is equivalent to that of the translated JB program. This guarantees that, if we prove a property of the JB program, then such property holds also for the original CIL program. Then we present a deep experimental evaluation over industrial-size open source popular programs, by applying the Julia static analyzer [25] to the translated JB.

Our experiments are focused on three main research hypotheses to prove the scalability, precision, and coverage of our approach. In particular, this article answers the following research questions:
Research Quest. 1 (Scalability). Does the CIL to JB translation scale up, that is, (i) can it deal with libraries of industrial size (100KLOCs) in a few minutes, and (ii) is its computational time comparable to that required by the static analysis phase?
Research Quest. 2 (Precision). Does the CIL to JB translation introduce less than 10% of the false alarms produced by the analyzer?
Research Quest. 3 (Libraries). Despite supporting only a subset of CIL, does the CIL to JB translation succeed on at least 95% of the system libraries?

About scalability, the overall goal is to obtain a tool that can be applied during the normal software development lifecycle, that is, it can perform the analysis at each build of the system. Therefore, we need a tool that analyzes hundreds of thousands of LOCs in few minutes. About precision, the 10% limit on false alarms (that is, alarms that are not real issues in the code, but they are due to some approximation performed by the analysis) is a standard bound to obtain a tool that does not produce too much *noise*, thus compromising its practical effectiveness when used by software developers. With respect to question 3, it is crucial that a static analyzer understands the behaviour of system libraries (*e.g.*, semantics of method calls and assumptions made on their parameters) or otherwise it could only rely on manual annotations or (possibly unsound) assumptions on their execution. However, system libraries need to

FormaliSE '18, June 2, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5718-0/18/06…$15.00
https://doi.org/10.1145/3193992.3193994

access memory through unsafe pointer. Java allows such behaviours through native methods (written in languages other than Java and bound through the Java Native Interface), while .Net allows unsafe pointers in its code. In these cases, our translation produces Java native methods. Through Research Question 3, we ensure that the effort of manually annotating .Net libraries is comparable to that needed for Java.

1.2 Related Work

Few attempts have been made in the past to translate CIL to JB. Grasshopper is probably the most popular one. However, it is not available any more[1]. As far as we can see, it was abandoned about a decade ago, and we cannot make any comparison with our translation. A similar tool was CLR2JVM [2]: it translates CIL to an intermediate XML_{CLR} representation, that can be then translated into XML_{JVM}, and finally to JB. As far as we can see[2], the tool should read .NET executables, but it failed parsing all the executable files of our experiments (see Sec. 5 for the complete list). This probably happened because CLR2JVM is not maintained any more (the last commit to the repository https://sourceforge.net/p/xmlvm/code/HEAD/tree/trunk/xmlvm/src/clr2jvm/ was more than six years ago), and it does not support the last CIL versions. Neither Grasshopper nor CLR2JVM has any documentation or discussion about how the translation is performed (in particular, how they handle instructions that are different from CIL to JB such as direct references). Therefore, as far as we can see our translation from CIL to JB is the only one that (i) works on recent releases of CIL and JB, and (ii) is formalized and proved sound.

Other translations between low-level languages exist, justified by the need of applying verification tools that work on a specific language only. For instance, [9] defines a translation from Boogie into WhyML and proves it sound, as we have done from CIL to JB. Similar translations work also at runtime, in particular inside a just-in-time compiler, as in [13]. However, we did not find any literature on the translation of CIL into JB for industrial-size software.

Many other static analysis tools for the .Net platform exist, in particular for C# code. There are tools that verify compliance to some guideline, such as Fxcop [24] and Coverity Prevent [17]. Other tools, such as NDepend [5] and CodeMetrics [6], provide metrics about the code under analysis. ReSharper [20] applies syntactical code inspections, finds code smells and guarantees compliance to coding standards. As far as we can see, there exist only two main fundamental tools with scientific base: (i) Spec# [12], an extension of C# with static checking of various kinds of manual specifications, and (ii) CodeContracts [23], an abstract interpretation-based static analyzer for CIL. In the Java world, the number of static analyzers based on syntactic reasoning, such as Checkstyle [1], FindBugs [4] and PMD [7], is comparable to that for .Net. However, Java attracted much more attention from the scientific community, and more semantic analyzers have been introduced during the last decade, such as CodeSonar [3], ThreadSafe [10] and Julia [25]. Few semantic analyzers, such as WALA [8], have been applied to various languages (*e.g.*, Java and JavaScript), but with ad-hoc translations of the source to the analyzed language.

Our approach lets us apply all the Java analyzers on .Net programs (almost) *for free*, that is, by translating CIL into JB and using the analyzers as they are (we expect that few manual annotations are needed to improve the precision of the analysis, in particular when dealing with library calls). We have also studied performance and results with Julia. As far as we know, our work is the first translation of CIL into JB for static analysis that is proven to be *sound* and comes with evidence that this translation applies to *industrial-size software* with results that are comparable in terms of precision and efficiency to those obtained on JB.

```
public class Wrap
{
  int  f ;
  Wrap(int f)  {  this . f  =  f ;  }
  static  ICollection <Wrap> WrapsCollection(int n)
  {
    ICollection <Wrap> result = new List<Wrap>();
    for  (int  i  = 0;  i  < n;  i++)
      result . Add(new Wrap(i));
    return  result ;
  }
}
```

(a) The C# source code of the running example.

```
1   static  ICollection <Wrap>
2     WrapsCollection(int n) {
3     newobj List<Wrap>::.ctor()
4     stloc.0
5     ldc.0
6     stloc.1
7     br #18
8
9     ldloc.0
10    ldloc.1
11    newobj Wrap::.ctor(int32)
12    call  ICollection <Wrap>::Add
13    ldloc.1
14    ldc.1
15    add
16    stloc.1
17
18    ldloc.1
19    ldarg.0
20    blt  #9
21
22    ldloc.0
23    ret
24  }
```

(b) The compilation into CIL of the code in (a).

```
1   static  WrapsCollection(I)LICollection  {
2     new <List>
3     dup
4     invokespecial <List.<init>>
5     astore_1
6     iconst_0
7     istore_2
8     goto 23
9
10    aload_1
11    iload_2
12    istore  3
13    new <Wrap>
14    dup
15    iload 3
16    invokespecial <Wrap.<init>>
17    invokevirtual <ICollection . Add>
18    iload_2
19    iconst_1
20    iadd
21    istore_2
22
23    iload_2
24    iload_0
25    if_icmplt 10
26
27    aload_1
28    areturn
29  }
```

(c) The translation into JB of the CIL in (b).

Figure 1: The C# code, CIL, and JB of the running example.

[1] We were unable to access the website http://dev.mainsoft.com, that seems to be the website of the tool from past forum discussions (http://stackoverflow.com/questions/95163/differences-between-msil-and-java-bytecode)

[2] http://xmlvm.org/documentation/

```
void init(ref int i)
{
  i++;
}

void run()
{
  int i=0;
  init(ref i);
}
```

(a) C# code

```
void init (WrapRef i) {
  i.value = i.value + 1;
}

int run() {
  int i = 0;
  WrapRef wrap = new WrapRef();
  wrap.value = i;
   init (wrap);
  i = wrap.value;
}
```

(b) Java code

```
void init(ref A& a)
{
  ldarg.0
  ldarg.0
  ldind.i4
  ldc.i4.1
  add
  stind.i4
}

int run()
{
  ldc.i4.0
  stloc.0
  ldloca.s 0
  call void Temporary.Foo
     ::' init '(int32&)
}
```

(c) CIL

Figure 2: CIL code using `ref` parameters.

JB	CIL	
`iadd` `ladd`	`add`	*(arith. op.)*

`iload i`
`lload i`
`aload i`
`istore i`
`lstore i`
`astore i`
	`ldloc i`
	`stloc i`
	`ldarg i`
		(local vars)

`invokevirtual` `invokestatic`	`call`	*(meth. call)*

`new T`
`getfield f`
`putfield f`
	`newobj T(···)`
	`ldfld f`
	`stfld f`
		(objects)

`if_icmpgt`	`bgt`	*(cond. branch)*
`dup` `dup2`	`dup`	*(stack)*

	`ldloca i`
	`stind`
	`ldind`
		(pointers)

Figure 3: JB and CIL minimal bytecode languages.

2 BACKGROUND

We provide here some background on CIL and JB, a running example, and a discussion on the architecture of our approach. For an exhaustive definition of JB and CIL, see [21] and [18], respectively.

2.1 CIL and JB

Bytecode is a machine-independent low-level programming language, used as target of the compilation of high-level languages, that hence becomes machine-independent. Bytecode languages are interpreted by their corresponding virtual machine, specific to each execution architecture. Both .Net and Java compile into bytecode. However, they use distinct instructions and virtual machines. .Net compiles into CIL, while Java compiles into JB. These have strong similarities: both use an operand stack for temporary values and an array of local variables standing for source code variables; both are object-oriented, with instructions for object creation, field access and virtual method dispatch. Despite these undeniable similarities, CIL and JB differ for the way of performing parameter passing (CIL uses a specific array of variables for the formal parameters, while JB merges them into the array of local variables); they handle object creation differently (CIL creates and initializes the object at the same time, while these are distinct operations in JB); they allocate memory slots differently (in CIL each value uses a slot, while JB uses 1 or 2 slots per 32- or 64- bit values, respectively); finally, CIL uses pointers explicitly, also in type-unsafe ways, while JB has no notion of pointer. We focus our formalization on a minimal representative subset of JB and CIL, as defined in Fig. 3. That figure presents bytecode instructions for:

arithmetic: JB has type-specific operations, such as `iadd` and `ladd` to add two integer or long values, respectively. Instead, CIL has generic operations, such as `add` to add two numerical values of the same type;

local variables access: JB has a single array of variables for both local variables and method arguments, and reads and writes values from this array through `xload` and `xstore`, where `x` is `i` for integer values, `l` for long values and `a` for references, respectively. In this array, 64 bits values use two subsequent slots. CIL, instead, uses two arrays: one for method's arguments (`ldarg i` loads the value of the i-th argument) and one for local variables (`ldloc i` and `stloc i` read and write the i-th local variable, respectively). In addition, it uses one slot both for 32- and 64-bit variables;

method call: JB has several kinds of method call instructions, such as `invokevirtual` and `invokestatic`. Instead, CIL has a unique `call` instruction;

object manipulation: in JB, instructions `new`, `getfield`, and `putfield` allocate a new object, read, and write its fields, respectively. CIL has similar instructions `newobj`, that also calls the constructor, `ldfld` and `stfld`;

conditional branch: JB has type-specific conditional branch instructions such as `if_icmpgt` (to branch if the greater than operator returns true on the topmost two integer values of the operand stack); CIL has generic instructions such as `bgt`;

stack: CIL duplicates the top value of the stack through the `dup` instruction. JB does the same with `dup` for 32 bits values and `dup2` for 64 bits values;

pointers: CIL contains some instructions to load the address of a local variable (`ldloca i`), and to store and load a value into the memory cell pointed by a reference (`stind` and `ldind`, respectively). Instead, JB has no direct pointer manipulation.

In the rest of this article, $\mathrm{St}_{\mathrm{CIL}}$ and $\mathrm{St}_{\mathrm{JB}}$ denote CIL and JB instructions or statements, respectively. A method (both in CIL and JB) is represented by (i) a sequence of (possibly conditional and branching) statements, and (ii) the number and static types of arguments and local variables.

2.2 Running Example

Fig. 1 shows the running example that Sec. 3 and 4 use to clarify the formalization. The C# code in Fig. 1a defines a class `Wrap` that wraps an integer value, and a static method `WrapsCollection` that, given an integer `n`, returns a collection of n wrappers containing values from 0 to $n-1$. Fig. 1b presents the (simplified) CIL obtained from its compilation: as usual with CIL, code is unstructured (*e.g.*, there are branches at lines 7 and 20), and each source code statement could be translated into many bytecode statements (*e.g.*, line 7 in Fig. 1a compiles into lines 3 and 4 in Fig. 1b). Fig. 1c presents the results of our translation of CIL into JB. The next sections explain the steps of the translation. First of all, notice some of the differences highlighted in Sec. 2.1. Namely, the type-generic CIL statement `stloc.1` at line 6 of Fig. 1b is translated into the type-specific `istore_2` at line 7 of Fig. 1c. Similarly, `newobj` (line 11) is translated into multiple JB statements (line 12-16). The running example contains few instructions that are not part of the minimal language defined in Sec. 2.1, and in particular (i) `ldc` and `i_const` that load constant (integer) values, (ii) `blt` and `if_icmplt` for conditional branching when an (integer) value is strictly less than another, (iii) `invokespecial` to invoke a specific method in JB, and (iv) `ret` and `areturn` to return a (reference) value.

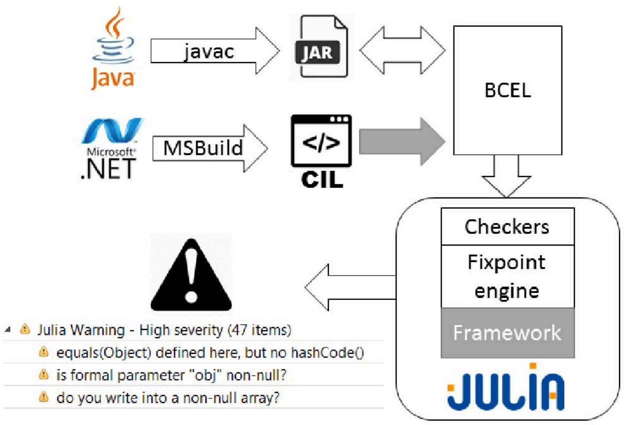
2.2.1 Example with Direct References. Safe C# code adopts direct pointers only for `out` and `ref` method parameters. These parameters can be assigned (and read as well in case of `ref`) inside the method, and “any change to the parameter in the called method is reflected in the calling method”[3]. In our translation, we build up wrapper objects to soundly represent their semantics.

Consider for instance the C# code in Fig. 2a. Method `init` receives a `ref` parameter and it increments it by one. This is compiled (Fig. 2c) into a method reading the value pointed by the direct reference (line 5), and writing it (line 8). Our goal is to translate this code into the Java code in Fig. 2b: we simulate the direct reference by constructing a wrapper object (line 7), assigning the value of the local variable to field `value` of the wrapper (line 8), and then propagating back the results of the call to `init` (line 9) by assigning the value of the local variable with the one stored in the wrapper (line 10). In addition, the `ref` parameter is replaced by the type of the wrapper object.

2.3 Julia

Julia [25] is a static analyzer for JB, based on abstract interpretation [16]. It transforms JB to basic blocks of code, that it analyzes through a fixpoint algorithm. Analyses are constraint-based or denotational. Currently, Julia features around 70 checkers, including nullness, termination, synchronization and taint analysis. Since Julia works on JB, in principle it analyzes any programming language that compiles into that bytecode. In particular, in this article we apply it to the analysis of the compilation of CIL into JB.

Fig. 4 is a high level view of our analysis, where the gray components have been implemented and modified to support .NET analyses. Java code is compiled by `javac` into a `jar` file, then parsed through the Byte Code Engineering Library [19] (BCEL). Julia receives this latter format, applies its analysis (by using many components such as the checkers, that define the analyses relying on various abstract domains, a fixpoint engine, and the framework specifying the semantics of some specific components of the programming language), and outputs a list of warnings. The added component in our approach is the translation of CIL into BCEL format (grey arrow in the upper part of Fig. 4). Since a program in BCEL format can be dumped into a jar file, we can dump a .dll file in this format. We added few annotations in a new framework of Julia to specify the main components of the .Net framework (*e.g.*, the signatures in the `Object` class).



Julia Warning - High severity (47 items)
equals(Object) defined here, but no hashCode()
is formal parameter "obj" non-null?
do you write into a non-null array?

Figure 4: Architecture of the analysis.

3 CONCRETE SEMANTICS

3.1 Notation

Let Ref and Num be the set of reference and numerical values, respectively, and let Val = Ref ∪ Num. A stack s of elements in T is a function in $\mathbb{N} \to T$ such that $\exists i \in \mathbb{N} : \forall i_1 \leq i : i_1 \in dom(s) \wedge \forall i_2 > i : i_2 \notin dom(s)$. We will refer to i as the height of s ($height(s)$). Given a stack s and an element e, $s :: e$ denotes a stack whose top element (that is, the one with the highest index) is e followed by the stack s.

As usual for object-oriented programming languages, an object is a map from field names to values, and a heap is a map from references to objects. Formally, Heap : Ref → Field → Val, where Field is the set containing all field names. $fresh(T, \mathsf{h}) = (r, \mathsf{h}')$ allocates an object of type T in heap h and returns (i) the reference r of the freshly allocated object, and (ii) the heap h′ resulting from the allocation of memory on h.

For simplicity, we consider only integer (Int) and long (Long) numerical types (NumTypes = {Int, Long}), and references (Ref). Given a value v, $typeOf(v)$ returns its type (Int, Long, or Ref). Since JB instructions often append a prefix to distinguish instructions dealing with different types (*e.g.*, `iadd` and `ladd`), we define a support function *JVMprefix* that, given a type t, returns the prefix of the given type (*i.e.*, `i` if t = Int, `l` if t = Long, and `a` if t = Ref). We define by WRef an object type with a unique field value.

Given a method signature `m` and a list of arguments L (with the receiver in the first argument if `m` is not static), $body(\mathtt{m}, L) : \mathbb{N} \to \mathrm{St}$ returns the body of the method resolving the call, that is, a

[3] https://msdn.microsoft.com/en-us/library/14akc2c7.aspx

sequence of statements (represented by a function mapping indexes to statements). Similarly, each statement belongs to a method; hence *getBody*(`st`) = b is the body of the method where `st` occurs. Finally, *isStatic*(`m`) means that `m` is static.

3.2 CIL

We define the concrete semantics of the CIL fragment of Sec. 2.1.

Concrete State A local state in CIL is composed by a stack of values or reference to local variables $\mathsf{Stack} : \mathbb{N} \rightarrow \mathsf{Val} \cup \mathsf{Ref}_{\mathsf{Loc}}$ (where $r_i \in \mathsf{Ref}_{\mathsf{Loc}}$ represents the cell's reference of the i-th local variable), an array of local variables $\mathsf{Loc} : \mathbb{N} \rightarrow \mathsf{Val}$, and an array of method arguments $\mathsf{Arg} : \mathbb{N} \rightarrow \mathsf{Val}$. A concrete CIL state consists of a local state and a heap, that is, $\Sigma_{\mathsf{CIL}} = \mathsf{Stack} \times \mathsf{Loc} \times \mathsf{Arg} \times \mathsf{Heap}$.

Concrete Semantics Fig. 5 shows the concrete CIL semantics $\langle \mathtt{st}, \sigma \rangle \rightarrow_{\mathsf{CIL}} \sigma'$. For a statement `st` and an entry state σ, it yields the state σ' resulting from the execution of `st` over σ; or a program label `l`, meaning that the next instruction to execute is that at `l`. Otherwise, the next instruction to execute is implicitly assumed to be the subsequent one, sequentially (if any).

For the most part, the concrete semantics is straightforward formalization of the runtime semantics defined by the CIL ECMA Standard [18]. For instance, rule `add` pops the two topmost values of the operand stack and replaces them with their addition. However, its semantics is defined iff the two values have the same type. Instead, `ldloc i` pushes to the operand stack the value of the i-th local variables, while `stloc i` stores the top of the operand stack into the i-th local variable. Statements working with objects, such as `ldfld` and `stfld`, read from and write into the heap, if their receiver is not `null`. `call` and `callstatic` create a frame (*i.e.*, an array of arguments, an empty array of local variables and an empty operand stack), execute the callee and leave its returned value on the stack, if any. For simplicity, the formalization assumes that there is no returned value. Finally, `ldloca i` loads the reference to the i-th local variable to the stack (represented by r_i), `stind` stores the given value to the given reference, and `ldind` loads the value pointed by the given reference.

Running example: Consider the running example in Fig. 1b and apply the concrete semantics when `n` is 1, that is, when the entry state consists of an empty operand stack and of an array of local variables, while the value of the arguments is $[0 \mapsto 1]$. Assume that `newobj` allocates the object at address #1. Then after the first block (lines 3-7) the address of the object is stored in local variable 0, local variable 1 (representing variable `i` of the source program) holds 0, and address #1 in the heap holds an object of type `List⟨Collection⟩`. Formally, the concrete state at line 7 is $(\emptyset, [0 \mapsto \#1, 1 \mapsto 0], [0 \mapsto 1], [\#1 \mapsto <>])$, where <> stands for the empty list. Then the body of the `for` loop (lines 9-16) is executed once and creates a new `Wrap` object (assume at address #2) wrapping 0, adds it to the list at address #1 and increments counter `i` (*i.e.*, local variable 0) by 1. Hence the execution of the concrete semantics of the body of the loop leads to the concrete state $\sigma = (\emptyset, [0 \mapsto \#1, 1 \mapsto 1], [0 \mapsto 1], [\#1 \mapsto < \#2 >, \#2 \mapsto \{f \mapsto 0\}])$ at line 16. The condition at line 20 will then route the program to line 22 (since argument 0 and local variable 1 hold 1). In conclusion, the concrete semantics will reach statement `ret` at line 23 with a state σ_{CIL} equal to σ, but where the operand stack contains reference #1.

3.3 JB

We define the concrete semantics of the JB fragment of Sec. 2.1.

Concrete State A local state in JB is composed by a stack of values $\mathsf{Stack} : \mathbb{N} \rightarrow \mathsf{Val}$ and an array of local variables $\mathsf{Loc} : \mathbb{N} \rightarrow \mathsf{Val}$. A concrete JB state consists of a local state and a heap, that is, $\Sigma_{\mathsf{JB}} = \mathsf{Stack} \times \mathsf{Loc} \times \mathsf{Heap}$.

Concrete Semantics Fig. 6 reports the concrete JB semantics $\rightarrow_{\mathsf{JB}}$. For a statement `st` and an entry state σ, it yields the state σ' resulting from the execution of `st` over σ. For the most part, the behavior of this semantics is identical to that for CIL. The main differences are that JB instructions work on specific types (*e.g.*, while CIL `add` statement adds two values of the same type, JB `iadd` and `ladd` statements add the values iff they are both `int` or `long`, respectively), and `new` only allocates a new object, while CIL `newobj` statements also calls a constructor.

Running example: The application of the JB concrete semantics is similar to that for CIL, but there are two minor differences: (i) there is only one array of local variables representing both CIL arguments and local variables (*e.g.*, CIL local variable 0 is represented by JB local variable 1, since the first local variable holds the argument of the method); and (ii) there is an *instrumentation* local variable at index 3. Therefore, after we apply the JB concrete semantics from the entry state that maps the argument to 1, we obtain the concrete state $([0 \mapsto \#1], [0 \mapsto 1, 1 \mapsto \#1, 2 \mapsto 1, 3 \mapsto 0], [\#1 \mapsto < \#2 >, \#2 \mapsto \{f \mapsto 0\}])$.

4 FROM CIL TO JB

4.1 Concrete States

Function $\mathbb{T}_\sigma[\![\ ]\!] : \Sigma_{\mathsf{CIL}} \rightarrow \Sigma_{\mathsf{JB}}$ translates CIL concrete states into JB concrete states (where l represents the number of elements in the stack of s): $\mathbb{T}_\sigma[\![(\mathsf{s}, \mathsf{l}, \mathsf{a}, \mathsf{h})]\!] = (\mathsf{s}', \mathit{cnvrtLoc}(\mathsf{l}, \mathsf{a}), \mathsf{h}'_l)$

This function (i) replaces direct reference with wrapper objects, and (ii) merges the array of local variables and arguments, adjusting variable indexes for 64 bits values. Formally:

$$
\begin{array}{l}
i \in \mathit{dom}(\mathsf{s}), l = \mathit{height}(\mathsf{s}) \\
\mathsf{s}' = \left[i \mapsto \begin{cases} \mathsf{s}(i) & \text{if } \mathsf{s}(i) \in \mathsf{Val} \\ r_i & \text{if } \mathsf{s}(i) \in \mathsf{Ref}_{\mathsf{Loc}} \end{cases} \right] \\
\text{where } \mathsf{h'}_{-1} = \mathsf{h}, \text{ and } (\mathsf{h'}_i, \mathsf{r}_i) = \mathit{allocWrp}(\mathsf{h}'_{i-1}, i) \\
\mathit{allocWrp}(\mathsf{h}, j) = \begin{cases} (\mathsf{h}, \mathtt{null}) \text{ if } \mathsf{s}(j) \in \mathsf{Val} & \\ \quad (\mathsf{h}'[r \mapsto \mathsf{h}(r)[\mathtt{value} \mapsto \mathsf{l}(j)]]) & \\ \qquad \text{where } (r, \mathsf{h}') = \mathit{fresh}(\mathsf{WRef}, \mathsf{h}) & \text{if } \mathsf{s}(i) \in \mathsf{Ref}_{\mathsf{Loc}} \end{cases}
\end{array}
$$

Intuitively, each direct reference in the operand stack (that is, $\mathsf{s}(i) \in \mathsf{Ref}_{\mathsf{Loc}}$) is replaced by another reference pointing to a wrapper object freshly allocated and containing in its field the value pointed by the original direct reference.

Then, for an array of values b and an index i, the following function counts the 64 bits types among the first i:

$$64^i_b = |\{j : 0 \leq j < i \text{ and } b[j] \text{ is a 64 bit value}\}|$$

Then the array $\mathit{cnvrtLoc}(\mathsf{l}, \mathsf{a})$ is defined as follows:

$$\forall 0 \leq i < |\mathsf{a}| : \mathit{cnvrtLoc}(\mathsf{l}, \mathsf{a})[i + 64^i_{\mathsf{a}}] = \mathsf{a}[i]$$

$$\forall 0 \leq i < |\mathsf{l}| : \mathit{cnvrtLoc}(\mathsf{l}, \mathsf{a})[|\mathsf{a}| + 64^{|\mathsf{a}|}_{\mathsf{a}} + i + 64^i_{\mathsf{l}}] = \mathsf{l}[i]$$

Running example: Consider the CIL exit state computed in Sec. 3.2, that is, $\sigma_{\mathsf{CIL}} = ([0 \mapsto \#1], [0 \mapsto \#1, 1 \mapsto 1], [0 \mapsto 1], [\#1 \mapsto < \#2 >, \#2 \mapsto \{f \mapsto 0\}])$. The CIL to JB translation computes $\mathbb{T}_\sigma[\![\sigma_{\mathsf{CIL}}]\!] = ([0 \mapsto \#1], [0 \mapsto 1, 1 \mapsto \#1, 2 \mapsto 1], [\#1 \mapsto < \#2 >$

$$\frac{\mathit{typeOf}(v_1) = \mathit{typeOf}(v_2)}{\langle \mathsf{add}, (s :: v_1 :: v_2, \mathsf{l}, \mathsf{a}, \mathsf{h})\rangle \rightarrow_{\mathsf{CIL}} (s :: (v_1 + v_2), \mathsf{l}, \mathsf{a}, \mathsf{h})}\ (\mathsf{add}) \qquad \frac{}{\langle \mathtt{ldloc\ i}, (s, \mathsf{l}, \mathsf{a}, \mathsf{h})\rangle \rightarrow_{\mathsf{CIL}} (s :: \mathsf{l}(\mathtt{i}), \mathsf{l}, \mathsf{a}, \mathsf{h})}\ (\mathtt{ldloc})$$

$$\frac{}{\langle \mathtt{stloc\ i}, (s :: \mathsf{v}, \mathsf{l}, \mathsf{a}, \mathsf{h})\rangle \rightarrow_{\mathsf{CIL}} (s, \mathsf{l}[\mathtt{i} \mapsto \mathsf{v}], \mathsf{a}, \mathsf{h})}\ (\mathtt{stloc}) \qquad \frac{}{\langle \mathtt{ldarg\ i}, (s, \mathsf{l}, \mathsf{a}, \mathsf{h})\rangle \rightarrow_{\mathsf{CIL}} (s :: \mathsf{a}(\mathtt{i}), \mathsf{l}, \mathsf{a}, \mathsf{h})}\ (\mathtt{ldarg})$$

$$\frac{\begin{array}{c}\mathit{isStatic}(\mathsf{m}(\mathtt{arg_0}, \cdots, \mathtt{arg_i})) = \mathsf{false} \wedge t \neq \mathtt{null} \wedge \\ \langle \mathit{body}(\mathsf{m}(\mathtt{arg_0}, \cdots, \mathtt{arg_i}), (t, v_1, \cdots, v_i)), ([], \emptyset, [0 \mapsto t, j \mapsto v_j : j \in [1..i]], \mathsf{h})\rangle \rightarrow_{\mathsf{CIL}} (\mathsf{s}', \mathsf{l}', \mathsf{a}', \mathsf{h}')\end{array}}{\langle \mathsf{call\ m}(\mathtt{arg_1}, \cdots, \mathtt{arg_i}), (s :: t :: v_1 :: \cdots :: v_i, \mathsf{l}, \mathsf{a}, \mathsf{h})\rangle \rightarrow_{\mathsf{CIL}} (s, \mathsf{l}, \mathsf{a}, \mathsf{h}')}\ (\mathtt{call})$$

$$\frac{\begin{array}{c}\mathit{isStatic}(\mathsf{m}(\mathtt{arg_0}, \cdots, \mathtt{arg_i})) = \mathsf{true} \wedge \\ \langle \mathit{body}(\mathsf{m}(\mathtt{arg_0}, \cdots, \mathtt{arg_i}), (v_1, \cdots, v_i)), ([], \emptyset, [j - 1 \mapsto v_j : j \in [1..i]], \mathsf{h})\rangle \rightarrow_{\mathsf{CIL}} (\mathsf{s}', \mathsf{l}', \mathsf{a}', \mathsf{h}')\end{array}}{\langle \mathsf{call\ m}(\mathtt{arg_1}, \cdots, \mathtt{arg_i}), (s :: v_1 :: \cdots :: v_i, \mathsf{l}, \mathsf{a}, \mathsf{h})\rangle \rightarrow_{\mathsf{CIL}} (s, \mathsf{l}, \mathsf{a}, \mathsf{h}')}\ (\mathtt{callstatic})$$

$$\frac{\mathit{fresh}(\mathsf{T}, \mathsf{h}) = (r, \mathsf{h}_1) \wedge \langle \mathit{body}(\mathtt{ctor}(\mathtt{arg_1}, \cdots, \mathtt{arg_i}), (v_1, \cdots, v_i)), ([], \emptyset, [0 \mapsto r, j \mapsto v_j : j \in [1..i]], \mathsf{h}_1)\rangle \rightarrow_{\mathsf{CIL}} (\mathsf{s}', \mathsf{l}', \mathsf{a}', \mathsf{h}')}{\langle \mathtt{newobj}\ \mathsf{T}(\mathsf{a_1}, \cdots, \mathsf{a_i}), (s :: v_1 :: \cdots :: v_i, \mathsf{l}, \mathsf{a}, \mathsf{h})\rangle \rightarrow_{\mathsf{CIL}} (s :: r, \mathsf{l}, \mathsf{a}, \mathsf{h}')}\ (\mathtt{newobj})$$

$$\frac{o \neq \mathtt{null}}{\langle \mathtt{ldfld}\ \mathsf{f}, (s :: o, \mathsf{l}, \mathsf{a}, \mathsf{h})\rangle \rightarrow_{\mathsf{CIL}} (s :: \mathsf{h}(o)(\mathsf{f}), \mathsf{l}, \mathsf{a}, \mathsf{h})}\ (\mathtt{ldfld}) \qquad \frac{o \neq \mathtt{null} \quad s' = \mathsf{h}(o)[\mathsf{f} \mapsto v]}{\langle \mathtt{stfld}\ \mathsf{f}, (s :: o :: v, \mathsf{l}, \mathsf{a}, \mathsf{h})\rangle \rightarrow_{\mathsf{CIL}} (s, \mathsf{l}, \mathsf{a}, \mathsf{h}[o \mapsto s'])}\ (\mathtt{stfld})$$

$$\frac{\mathit{typeOf}(v_1) = \mathit{typeOf}(v_2) \wedge v_1 > v_2}{\langle \mathtt{bgt\ 1}, (s :: v_1 :: v_2, \mathsf{l}, \mathsf{a}, \mathsf{h})\rangle \rightarrow_{\mathsf{CIL}} \langle \mathtt{1}, (s, \mathsf{l}, \mathsf{a}, \mathsf{h})\rangle}\ (\mathtt{bgt\ true}) \qquad \frac{\mathit{typeOf}(v_1) = \mathit{typeOf}(v_2) \wedge v_1 \leq v_2}{\langle \mathtt{bgt\ 1}, (s :: v_1 :: v_2, \mathsf{l}, \mathsf{a}, \mathsf{h})\rangle \rightarrow_{\mathsf{CIL}} (s, \mathsf{l}, \mathsf{a}, \mathsf{h})}\ (\mathtt{bgt\ false})$$

$$\frac{}{\langle \mathtt{ldloca\ i}, (s, \mathsf{l}, \mathsf{a}, \mathsf{h})\rangle \rightarrow_{\mathsf{CIL}} (s :: r_i, \mathsf{l}, \mathsf{a}, \mathsf{h})}\ (\mathtt{ldloca}) \qquad \frac{}{\langle \mathtt{stind}, (s :: r_i :: v, \mathsf{l}, \mathsf{a}, \mathsf{h})\rangle \rightarrow_{\mathsf{CIL}} (s, \mathsf{l}, \mathsf{a}, \mathsf{h}[r_i \mapsto v])}\ (\mathtt{stind})$$

$$\frac{}{\langle \mathsf{dup}, (s :: v, \mathsf{l}, \mathsf{a}, \mathsf{h})\rangle \rightarrow_{\mathsf{CIL}} (s :: v :: v, \mathsf{l}, \mathsf{a}, \mathsf{h})}\ (\mathsf{dup}) \qquad \frac{}{\langle \mathtt{ldind}, (s :: r_i, \mathsf{l}, \mathsf{a}, \mathsf{h})\rangle \rightarrow_{\mathsf{CIL}} (s :: \mathsf{h}(r_i), \mathsf{l}, \mathsf{a}, \mathsf{h})}\ (\mathtt{ldind})$$

Figure 5: Concrete CIL semantics.

, $\#2 \mapsto \{f \mapsto 0\}]$) (it merges CIL arguments and local variables). This state is almost identical to the exit state of the JB concrete semantics applied to the running example (Sec. 3.3) except for the instrumentation local variable at index 3.

4.2 Statements

Fig. 7 formalizes the translation $\mathbb{T}[\![\mathtt{st_{CIL}}, \mathsf{K}]\!] = \mathtt{st_{JB}}$ of a single CIL statement into a sequence of JB statements. K is the static type information about locals, arguments and stack elements, computed at $\mathtt{st_{CIL}}$ by a standard algorithm [18]. In particular, types and height of the stack are fixed and known for each bytecode. In addition, the forth component $\overline{\mathsf{w}}$ is a stack of elements in $\perp \cup (\mathbb{N} \times \mathbb{N})$ that, for each element in the stack, tells (i) $\perp$ if it is not a direct reference, or (ii) (i, j) where i is the index of the local variables pointed by the direct reference[4], and j the index of the local instrumentation variable containing a pointer to the wrapper simulating the reference.

Few CIL statements (namely, `ldfld` and `stfld`) have a one-to-one translation into a JB statement (`getfield` and `putfield`). The statements reading and writing local variables and arguments (`ldarg`, `ldloc`, and `stloc`) are translated into their JB counterpart (*x*`load`, *x*`store`, respectively) taking into account the type of the value at the top of the stack, and adjusting the index of the variable taking into account arguments and 64 bit variables. Some CIL statements (`dup`) get translated into different JB statements on the basis of contextual information such as the type of values in the operand stack (`dup` and `dup2`). Other CIL statements can be translated only if the type of the values in the operand stack is numeric: (i) `add` can be translated into `ladd` and `iadd`, and (ii) `bgt` to `is_icmpgt`; if they are applied to references (as in generic CIL code), then the code is considered unsafe and not translated.

`call` requires to (i) translate the method call to the corresponding static or dynamic invocation statement in JB, and (ii) to propagate the side effects on direct pointers passed to the method as `out/ref` parameters to the local variables of the callee. The translation of `newobj` is tricky because of the different patterns used in CIL and JB for object creation[5]. While CIL creates and initializes the object (*i.e.*, calls its constructor) with a single instruction, JB splits these operations and requires the newly created object to occur below the arguments on the stack, before calling the constructor. Hence, the translation relies on a function *freshIdx* to store and load the values of the constructor arguments through instrumentation local variables. In particular, given a CIL method *m*, the number and types of arguments and local variables of the method are known (Sec. II.15.4 of [18]). Therefore, function *cnvrtLoc* tells which local variables the translated JB method already uses. Then, for each argument of each `newobj` statement in *m*, it is possible to allocate a fresh local variable to store and load its value. In this way, the translation allocates a new object and puts its address below the constructor arguments.

Instructions dealing with direct pointers (namely, `ldloca`, `stind`, and `ldind` in our minimal language) are translated through equivalent CIL instructions dealing with wrapper objects (and their

[4] Since the language we introduced in Fig. 3 supports only `ldloca` to get a direct pointer, we need to track only this information in the formalization.

[5] For sake of simplicity, we assume the constructor does not have `out/ref` parameters. In the implementation, they are treated as for `call` statements

$$\frac{\mathit{typeOf}(v) \neq \mathsf{Long}}{\langle \mathsf{dup}, (s :: v, \mathsf{l}, \mathsf{h})\rangle \rightarrow_{\mathsf{JB}} (s :: v :: v, \mathsf{l}, \mathsf{h})}\ (\mathsf{dup})$$

$$\frac{\mathit{typeOf}(v_1) \neq \mathsf{Long}}{\langle \mathsf{dup2}, (s :: v_1 :: v_2, \mathsf{l}, \mathsf{h})\rangle \rightarrow_{\mathsf{JB}} (s :: v_1 :: v_2 :: v_1 :: v_2, \mathsf{l}, \mathsf{h})}\ (\mathsf{dup2\ 32}) \qquad \frac{\mathit{typeOf}(v) = \mathsf{Long}}{\langle \mathsf{dup2}, (s :: v, \mathsf{l}, \mathsf{h})\rangle \rightarrow_{\mathsf{JB}} (s :: v :: v, \mathsf{l}, \mathsf{h})}\ (\mathsf{dup2\ 64})$$

$$\frac{\mathit{typeOf}(v_1) = \mathsf{Int} \wedge \mathit{typeOf}(v_2) = \mathsf{Int}}{\langle \texttt{iadd}, (s :: v_1 :: v_2, \mathsf{l}, \mathsf{h})\rangle \rightarrow_{\mathsf{JB}} (s :: (v_1 + v_2), \mathsf{l}, \mathsf{h})}\ (\texttt{iadd}) \qquad \frac{\mathit{typeOf}(v_1) = \mathsf{Long} \wedge \mathit{typeOf}(v_2) = \mathsf{Long}}{\langle \texttt{ladd}, (s :: v_1 :: v_2, \mathsf{l}, \mathsf{h})\rangle \rightarrow_{\mathsf{JB}} (s :: (v_1 + v_2), \mathsf{l}, \mathsf{h})}\ (\texttt{ladd})$$

$$\frac{x = \mathit{JVMprefix}(\mathit{typeOf}(\mathsf{l}(\texttt{i})))}{\langle x\texttt{load i}, (s, \mathsf{l}, \mathsf{h})\rangle \rightarrow_{\mathsf{JB}} (s :: \mathsf{l}(\texttt{i}), \mathsf{l}, \mathsf{h})}\ (x\texttt{load}) \qquad \frac{x = \mathit{JVMprefix}(\mathit{typeOf}(\mathsf{v}))}{\langle x\texttt{store i}, (s :: \mathsf{v}, \mathsf{l}, \mathsf{h})\rangle \rightarrow_{\mathsf{JB}} (s, \mathsf{l}[\texttt{i} \mapsto \mathsf{v}], \mathsf{h})}\ (x\texttt{store})$$

$$\frac{\begin{array}{c}\mathit{isStatic}(\mathsf{m}(\texttt{arg}_{\texttt{0}}, \cdots, \texttt{arg}_{\texttt{i}})) = \mathsf{false} \wedge t \neq \texttt{null} \wedge \\ \langle \mathit{body}(\mathsf{m}(\texttt{arg}_{\texttt{0}}, \cdots, \texttt{arg}_{\texttt{i}}), (t, v_1, \cdots, v_i)), ([], [0 \mapsto t, j \mapsto v_j : j \in [1..i]], \mathsf{h})\rangle \rightarrow_{\mathsf{JB}} (\mathsf{s}', \mathsf{l}', \mathsf{h}')\end{array}}{\langle \texttt{invokevirtual}\ \mathsf{m}(\texttt{arg}_1, \cdots, \texttt{arg}_{\texttt{i}}), (s :: t :: v_1 :: \cdots :: v_i, \mathsf{l}, \mathsf{h})\rangle \rightarrow_{\mathsf{JB}} (s, \mathsf{l}, \mathsf{h}')}\ (\texttt{invokevirtual})$$

$$\frac{\begin{array}{c}\mathit{isStatic}(\mathsf{m}(\texttt{arg}_{\texttt{0}}, \cdots, \texttt{arg}_{\texttt{i}})) = \mathsf{true} \wedge \\ \langle \mathit{body}(\mathsf{m}(\texttt{arg}_{\texttt{0}}, \cdots, \texttt{arg}_{\texttt{i}}), (v_1, \cdots, v_i)), ([], [j - 1 \mapsto v_j : j \in [1..i]], \mathsf{h})\rangle \rightarrow_{\mathsf{JB}} (\mathsf{s}', \mathsf{l}', \mathsf{h}')\end{array}}{\langle \texttt{invokestatic}\ \mathsf{m}(\texttt{arg}_1, \cdots, \texttt{arg}_{\texttt{i}}), (s :: v_1 :: \cdots :: v_i, \mathsf{l}, \mathsf{h})\rangle \rightarrow_{\mathsf{JB}} (s, \mathsf{l}, \mathsf{h}')}\ (\texttt{invokestatic})$$

$$\frac{\mathit{fresh}(\mathsf{T}, \mathsf{h}) = (r, \mathsf{h}')}{\langle \mathsf{new\ T}, (s, \mathsf{l}, \mathsf{h})\rangle \rightarrow_{\mathsf{JB}} (s :: r, \mathsf{l}, \mathsf{h}')}\ (\mathsf{new}) \qquad \frac{o \neq \texttt{null}}{\langle \texttt{getfield}\ \mathsf{f}, (s :: o, \mathsf{l}, \mathsf{h})\rangle \rightarrow_{\mathsf{JB}} (s :: \mathsf{h}(o)(\mathsf{f}), \mathsf{l}, \mathsf{h})}\ (\texttt{getfield})$$

$$\frac{o \neq \texttt{null} \quad s' = \mathsf{h}(o)[\mathsf{f} \mapsto v]}{\langle \texttt{putfield}\ \mathsf{f}, (s :: o :: v, \mathsf{l}, \mathsf{h})\rangle \rightarrow_{\mathsf{JB}} (s, \mathsf{l}, \mathsf{h}[o \mapsto s'])}\ (\texttt{putfield}) \qquad \frac{\mathit{typeOf}(v_1) = \mathsf{Int} \wedge \mathit{typeOf}(v_2) = \mathsf{Int} \wedge v_1 > v_2}{\langle \texttt{if\_icmpgt 1}, (s :: v_1 :: v_2, \mathsf{l}, \mathsf{h})\rangle \rightarrow_{\mathsf{JB}} \langle \texttt{1}, (s, \mathsf{l}, \mathsf{h})\rangle}\ \begin{pmatrix}\texttt{if\_icmpgt} \\ \texttt{true}\end{pmatrix}$$

$$\frac{\mathit{typeOf}(v_1) = \mathsf{Int} \wedge \mathit{typeOf}(v_2) = \mathsf{Int} \wedge v_1 \leq v_2}{\langle \texttt{if\_icmpgt 1}, (s :: v_1 :: v_2, \mathsf{l}, \mathsf{h})\rangle \rightarrow_{\mathsf{JB}} (s, \mathsf{l}, \mathsf{h})}\ \begin{pmatrix}\texttt{if\_icmpgt} \\ \texttt{false}\end{pmatrix}$$

Figure 6: Concrete JB semantics.

field value). Therefore, stind and ldind are simply translated through equivalent write and read of field value, respectively. ldloca instead requires to allocate a wrapper object newobj, stores a reference to the wrapper (stloc) in an instrumentation variable obtained through *freshIdx*, stores the value pointed by the direct reference in the local variable to its field value (ldloc and stfld), and leaves a reference to the wrapper in the stack (dup).

Each CIL statement is translated into one or more JB statements, hence offsets are not preserved. Thus, function *statementIdx* : St → ℕ yields the JB offset of the first statement in the translation of the given CIL statement. In addition, since direct references are replaced by wrapper objects, when a method parameter has a direct reference type &T (and this happens when it is a ref or out parameter in safe C#), this is replaced by a wrapper object WRef.

Running example: Consider the running example in Fig. 1. Most CIL statements are translated into a single JB statement (*e.g.*, lines 18-20 and 22-23 of Fig. 1b are translated into lines 23-25 and 27-28 of Fig. 1c), with the noticeable exception of the CIL newobj statements at line 3 and 11, translated into lines 2-4 and 12-16, respectively. The former passes no argument to the constructor; the latter (that instantiates a Wrap) calls a constructor with an argument, hence requiring an instrumentation variable at index 3.
Direct references As sketched in Sec. 2.2.1, we model the semantics of pointers in safe C# code through wrapper objects. In particular, ldind (line 5 of Fig. 2c) is translated into the field access i.value (right side of the assignment at line 2 of Fig. 2b), while stind (line 8 of Fig. 2c) is translated into the assignment of i.value (left side of the assignment at line 2 of Fig. 2b). In addition, ldloca (line 15 of Figure 2c) leads to the construction and assignment of a wrapper object (lines 7 and 8 of Figure 2b), while after the method call the value contained in the wrapper object is written into the local variable (line 10 of Figure 2b).

4.3 Correctness

This section proves the translation from CIL to JB statements correct. Namely, given a concrete CIL state σ_{CIL} and applying the operational semantics for a statement st, one obtains a state that, when translated into JB, is exactly the state resulting from the translation of σ_{CIL} into JB and the application of the JB semantics to it:

$$\begin{array}{c}\forall \texttt{st} \in \mathsf{St}_{\mathsf{CIL}}, \sigma_{\mathsf{CIL}} \in \Sigma_{\mathsf{CIL}} : \\ \langle \texttt{st}, \sigma_{\mathsf{CIL}}\rangle \rightarrow_{\mathsf{CIL}} \sigma'_{\mathsf{CIL}} \text{ and } \langle \mathbb{T}[\![\texttt{st}, \mathsf{K}]\!], \mathbb{T}_{\sigma}[\![\sigma_{\mathsf{CIL}}]\!]\rangle \rightarrow_{\mathsf{JB}} \sigma'_{\mathsf{JB}} \\ \Downarrow \\ \mathbb{T}_{\sigma}[\![\sigma'_{\mathsf{CIL}}]\!] =_{\bullet} \sigma'_{\mathsf{JB}}\end{array}$$

where $\sigma_1 =_{\bullet} \sigma_2$ means that the two states are equal up to instrumentation variables introduced by the translation process. Formally, let $\sigma_1 = (\mathsf{s}_1, \texttt{l}^1_1 :: \ldots :: \texttt{l}^{\mathsf{n}}_1, \mathsf{h}_1)$ and $\sigma_2 = (\mathsf{s}_2, \texttt{l}^1_2 :: \ldots :: \texttt{l}^{\mathsf{n}}_2, :: \texttt{l}^{\mathsf{n}+1}_2 :: \ldots :: \texttt{l}^{\mathsf{n}+\mathsf{k}}_2, \mathsf{h}_2)$, then $\sigma_1 =_{\bullet} \sigma_2$ iff $\mathsf{s}_1 = \mathsf{s}_2$, and $\forall \texttt{i} \leq \mathsf{n} : \texttt{l}^{\texttt{i}}_1 = \texttt{l}^{\texttt{i}}_2$, and $\mathsf{h}_1 = \mathsf{h}_2$. Note that instrumentation variables are present only in the JB state, hence in the right hand-side of the equality.
Running example: Sec. 3.2 showed that, starting from $\sigma_{\mathsf{CIL}} = (\emptyset, \emptyset, [0 \mapsto 1], \emptyset)$, the concrete semantics on the program in Fig. 1b

$$\mathbb{T}\llbracket \mathsf{dup}, \bar{\mathsf{s}} :: \mathsf{t}, \bar{\mathsf{l}}, \bar{\mathsf{a}}, \overline{\mathsf{w}} \rrbracket = \begin{cases} \mathsf{dup} & \text{if } t \neq \mathsf{Long} \\ \mathsf{dup2} & \text{if } t = \mathsf{Long} \end{cases}$$

$$\mathbb{T}\llbracket \mathsf{add}, \bar{\mathsf{s}} :: \mathsf{t}_1 :: \mathsf{t}_2, \bar{\mathsf{l}}, \bar{\mathsf{a}}, \overline{\mathsf{w}} \rrbracket = \begin{cases} \texttt{iadd} & \text{if } t_1 = t_2 = \mathsf{Int} \\ \texttt{ladd} & \text{if } t_1 = t_2 = \mathsf{Long} \end{cases}$$

$$\mathbb{T}\llbracket \texttt{ldloc i}, \bar{\mathsf{s}}, \bar{\mathsf{l}}, \bar{\mathsf{a}}, \overline{\mathsf{w}} \rrbracket = x\texttt{load}\ j \text{ where } j = |\bar{\mathsf{a}}| + 64^{|\bar{\mathsf{a}}|}_{\bar{\mathsf{a}}} + i + 64^{i}_{\bar{\mathsf{l}}} \wedge x = \mathcal{JVMprefix}(\mathit{typeOf}(\bar{\mathsf{l}}(\mathtt{i})))$$

$$\mathbb{T}\llbracket \texttt{stloc i}, \bar{\mathsf{s}} :: \mathsf{t}, \bar{\mathsf{l}}, \bar{\mathsf{a}}, \overline{\mathsf{w}} \rrbracket = x\texttt{store}\ j \text{ where } j = |\bar{\mathsf{a}}| + 64^{|\bar{\mathsf{a}}}_{\bar{\mathsf{a}}} + i + 64^{i}_{\bar{\mathsf{l}}} \wedge x = \mathcal{JVMprefix}(\mathit{typeOf}(\bar{\mathsf{l}}(\mathtt{i})))$$

$$\mathbb{T}\llbracket \texttt{ldarg i}, \bar{\mathsf{s}}, \bar{\mathsf{l}}, \bar{\mathsf{a}}, \overline{\mathsf{w}} \rrbracket = x\texttt{load}\ j \text{ where } j = i + 64^{i}_{a} \wedge x = \mathcal{JVMprefix}(\mathit{typeOf}(\bar{\mathsf{a}}(\mathtt{i})))$$

$$\begin{aligned} &\mathbb{T}\llbracket \texttt{call}\ \mathsf{m}(\mathsf{arg}_1, \cdots, \mathsf{arg}_\mathtt{i}), \\ &\quad \bar{\mathsf{s}} :: t_1 :: \cdots :: t_i, \bar{\mathsf{l}}, \bar{\mathsf{a}}, \overline{\mathsf{w}} :: p_1 :: \cdots :: p_i \rrbracket \\ &= \texttt{invoke}\ ;\ \texttt{aload}\ p^1_{idx_1}\ ;\ \texttt{getfield value}\ ;\ x_{idx_1}\texttt{store}\ p^2_{idx_1}\ ;\ \cdots \\ &\qquad \cdots \texttt{aload}\ p^1_{idx_j}\ ;\ \texttt{getfield value}\ ;\ x_{idx_j}\texttt{store}\ p^2_{idx_j}\ ; \\ &\quad \text{where } \texttt{invoke} = \begin{cases} \texttt{invokestatic}\ \mathsf{m}(\mathsf{arg}_1, \cdots, \mathsf{arg}_\mathtt{i}) & \text{if } \mathit{isStatic}(\mathsf{m}(\mathsf{arg}_1, \cdots, \mathsf{arg}_\mathtt{i})) \\ \texttt{invokevirtual}\ \mathsf{m}(\mathsf{arg}_1, \cdots, \mathsf{arg}_\mathtt{i}) & \text{otherwise} \end{cases} \\ &\quad \{idx_1, \cdots, idx_j\} = \{k : \mathsf{arg}_\mathsf{k} \in \mathsf{Ref}_\mathsf{Loc}\} \\ &\quad \forall k \in [1..j] : x_{idx_k} = \mathcal{JVMprefix}(\mathit{typeOf}(\bar{\mathsf{l}}(p^2_{idx_k}))) \wedge \forall r \in [1..i] : p_i = (p^1_i, p^2_j) \end{aligned}$$

$$\begin{aligned} &\mathbb{T}\llbracket \texttt{newobj}\ \mathsf{T}(\mathsf{a}_1, \cdots, \mathsf{a}_\mathtt{i}), \\ &\quad \bar{\mathsf{s}} :: t_1 :: \cdots :: t_i, \bar{\mathsf{l}}, \bar{\mathsf{a}}, \overline{\mathsf{w}} \rrbracket \\ &= x_i\texttt{store}\ idx_i\ ;\ \cdots\ ;\ x_1\texttt{store}\ idx_1\ ;\ \mathsf{new\ T}\ ;\ \mathsf{dup}\ ; \\ &\qquad x_1\texttt{load}\ idx_1\ ;\ \cdots\ ;\ x_i\texttt{load}\ idx_i\ ;\ \texttt{invokevirtual}\ <\texttt{init}>(\mathsf{arg}_1, \cdots, \mathsf{arg}_\mathtt{i}) \\ &\quad \text{where } \forall j \in [1..i] : x_j = \mathcal{JVMprefix}(a_j) \wedge idx_j = \mathit{freshIdx}(\texttt{newobj}\ \mathsf{T}(\mathsf{a}_1, \cdots, \mathsf{a}_\mathtt{i}), j) \end{aligned}$$

$$\mathbb{T}\llbracket \texttt{ldfld f}, \bar{\mathsf{s}} :: \mathsf{t}_\mathsf{o}, \bar{\mathsf{l}}, \bar{\mathsf{a}}, \overline{\mathsf{w}} \rrbracket = \texttt{getfield f}$$

$$\mathbb{T}\llbracket \texttt{stfld f}, \bar{\mathsf{s}} :: \mathsf{t}_\mathsf{o} :: \mathsf{t}_\mathsf{v}, \bar{\mathsf{l}}, \bar{\mathsf{a}}, \overline{\mathsf{w}} \rrbracket = \texttt{putfield f}$$

$$\mathbb{T}\llbracket \texttt{bgt k}, \bar{\mathsf{s}} :: \mathsf{t}_1 :: \mathsf{t}_2, \bar{\mathsf{l}}, \bar{\mathsf{a}}, \overline{\mathsf{w}} \rrbracket = \texttt{if\_icmpgt}\ \mathsf{k}' \text{ where } \mathsf{k}' = \mathit{statementIdx}(\mathit{getBody}(\mathsf{bgt\ k})(\mathsf{k})) \text{ if } t_1 = t_2 = \mathsf{Int}$$

$$\begin{aligned} \mathbb{T}\llbracket \texttt{ldloca i}, \bar{\mathsf{s}}, \bar{\mathsf{l}}, \bar{\mathsf{a}}, \overline{\mathsf{w}} \rrbracket &= \mathbb{T}\llbracket \texttt{newobj WrapRef()}\ ;\texttt{dup2}\ ;\texttt{stloc j};\texttt{ldloc i};\texttt{stfld value}, \bar{\mathsf{s}}, \bar{\mathsf{l}}, \bar{\mathsf{a}}, \overline{\mathsf{w}} \rrbracket \\ &\quad \text{where } j = \mathit{freshIdx}(\texttt{ldloca i}, 0) \end{aligned}$$

$$\mathbb{T}\llbracket \texttt{stind}, \bar{\mathsf{s}}, \bar{\mathsf{l}}, \bar{\mathsf{a}}, \overline{\mathsf{w}} \rrbracket = \mathbb{T}\llbracket \texttt{stfld value}, \bar{\mathsf{s}}, \bar{\mathsf{l}}, \bar{\mathsf{a}}, \overline{\mathsf{w}} \rrbracket$$

$$\mathbb{T}\llbracket \texttt{ldind}, \bar{\mathsf{s}}, \bar{\mathsf{l}}, \bar{\mathsf{a}}, \overline{\mathsf{w}} \rrbracket = \mathbb{T}\llbracket \texttt{ldfld value}, \bar{\mathsf{s}}, \bar{\mathsf{l}}, \bar{\mathsf{a}}, \overline{\mathsf{w}} \rrbracket$$

Figure 7: Translation of CIL statements into JB.

ends up in $\sigma'_{\text{CIL}} = ([0 \mapsto \#1], [0 \mapsto \#1, 1 \mapsto 1], [0 \mapsto 1], [\#1 \mapsto < \#2 >, \#2 \mapsto \{f \mapsto 0\}])$. Then Sec. 3.3 showed that, starting from the corresponding $\mathbb{T}_\sigma \llbracket \sigma_{\text{CIL}} \rrbracket$ state, the JB concrete semantics leads to $\sigma'_{\text{JB}} = ([0 \mapsto \#1], [0 \mapsto 1, 1 \mapsto \#1, 2 \mapsto 1, 3 \mapsto 0], [\#1 \mapsto < \#2 >, \#2 \mapsto \{f \mapsto 0\}]$. Hence, by definition of $\mathbb{T}_\sigma \llbracket\ \rrbracket$, we have $\mathbb{T}_\sigma \llbracket \sigma'_{\text{CIL}} \rrbracket = \bullet \sigma'_{\text{JB}}$ since the two stacks and the two heaps are equal, the values of three local variables of the JB state correspond to the values of the argument and the two local variables of the CIL state, respectively, and $=_\bullet$ projects away the fourth variable of the JB state σ'.

```
1  if_icmpeq 4
2  iconst_0
3  goto 5
4  iconst_1
5  nop
```

(a) On integers

```
1  lcmp
2  iconst_0
3  if_icmpeq 6
4  iconst_0
5  goto 7
6  iconst_1
7  nop
```

(b) On longs

Figure 8: Translation of `ceq`.

4.4 Other Instructions

In this section, we informally discuss how our approach deals with CIL instructions that are slightly different from other instructions in JB. We decided to handle these instructions informally since their translation is mostly straightforward. It is intended for readers that are expert of JB, CIL and more advanced C# features, such as generic type erasure in JB, or delegates in C#.

Numerical and Reference Comparison CIL compares numerical or reference values in two ways: through conditional branches (*e.g.*, `beq` branches when the topmost two values on the stack are equals) and comparisons (*e.g.*, `ceq` pushes 1 iff the topmost two values on the stack are equals, and 0 otherwise). As usual, these instructions are type independent and apply to numerical (int, float, long, ...) as well as reference values. JB uses a different approach, since its instructions are type dependent. If the topmost two values on the stack are integers, it uses a conditional branch instruction (`if_icmpeq`) similar to that of CIL (`beq`). However, JB has no comparison instruction on integers and we need to simulate it through a sequence of JB instructions relying on constants and branch. For instance, a `ceq` statement on integers is simulated as in Fig. 8a. Instead, if the topmost two values on the stack are long, JB uses a comparison statement `lcmp` that pushes to the stack 1, 0, or -1 iff the first value is less than, equal to, or greater than the second, respectively. Hence, we simulate CIL conditional branch and comparison instructions through `lcmp`, integer constants and a conditional branch on integers. For instance, `beq` is translated into the sequence `lcmp; ifne #i;` where `i` is the target JB instruction of `beq`. Instead, the treatment of comparisons over long is similar to int. Namely, `ceq` is translated into the code in Fig. 8b. Moreover, conditional branch and comparison work also on references. Equality and inequality statements are treated as for integers, since JB defines an `if_acmpeq` statement. Other CIL operators (*e.g.*, `bgt`) can be applied to arbitrary references, as long as one of them is `null`. For instance, it might branch if a reference is strictly greater than `null` (that is, if it is not `null`) and we translate these cases accordingly.

```
ldloc.0
ldc.i4.0
 callvirt  !0  List<A>::get(int32)
stloc.1
```

(a) In CIL

```
aload_0
iconst_0
invoke List.get:(I)LObject;
checkcast A
astore_1
```

(b) In JB

Figure 9: Getting an element from a list.

Generic Types CIL keeps information about generic types, while JB erases it into `Object`. For instance, imagine that we have a local variable `list` of type `List⟨A⟩`. At source code level, a method call like `A a = list.get(0)` in Java or `A a = list[0]` in C# is legal since the elements of the list have type `A` in both languages. At bytecode level, getting an element from the list effectively returns an object of type `A` in CIL (see Fig. 9a), while it returns an object whose static type is `Object` in JB and casts it dynamically to `A` through a `checkcast` (Fig. 9b). Hence, our translation of a CIL method call with generic return type `T` adds a `checkcast` instruction to `T` after the call.

Primitive types (*e.g.*, `int` and `long`) can be passed as generic types in CIL but not in JB. Hence, when using a primitive type for the generic parameter or return value of a CIL method call, we box and unbox the primitive value into a Java wrapper class such as `java.lang.Integer`.

Delegates Lambda expressions have only been introduced in Java 8, while C# has been using *delegates* since its very beginning. C# implements delegates through CIL instructions that load a pointer to a method (`ldftn`) and execute it, sometime by using inner classes. Namely, C# accesses a pointer to the method through `ldftn` and calls the `Invoke` method of the delegate class. Consider for instance Fig. 10. The C# code in Fig. 10a uses a delegate to call a method. In Fig. 10b, this is compiled into a `ldftn` statement at line 4 followed by a call to `Invoke` at line 7. We translate this by using reflection and string constants. Namely, the signature of the method pointed by `ldftn` is represented by a string, passed to an instrumentation library call in class `Reflection`, that calls this method by reflection (Fig. 10c). However, many static analyzers (including Julia) are unsound for reflection. Hence, our translation marks all signatures accessed in this way as entry points (that is, methods that might be directly called from outside the application and therefore are analyzed under the most generic assumptions). This might cause a loss of precision, since contextual information on delegates is lost, but preserves soundness.

Async and Await In C#, an `async` method returns a `Task` object that allows the caller to execute the code of the method asynchronously. On the other hand, statement `await` waits until the execution of the asynchronous method ends and extracts the results of the computation. This pattern is compiled into method pointers and reflection at CIL bytecode level, in the same way delegates are treated. Therefore, we apply the same solution for delegates we described in Sec. 4.4.

5 EXPERIMENTAL RESULTS

Our experiments aimed at answering the research questions in Sec. 1.1, hence assessing the practical interest of our translation for static analysis purposes.

Table 1: Experimental results on the 5 most starred Github C# projects.

Program	LOC	met.	fail	Tr. t.	An.t.	Al	F	Prec.
CodeHub	32,510	4,887	0	0’07"	0’43"	9	1	89%
SignalR	71,207	6,610	3	0’07"	0’50"	8	1	88%
Dapper	22,513	1,058	0	0’07"	0’29"	13	3	77%
ShareX	171,580	11,568	14	0’58"	2’08"	57	0	100%
Nancy	109,139	8,817	0	0’07"	1’25"	18	1	94%
Total	406,949	32,940	17	1’26"	4’35"	105	6	94%

Table 2: Experimental results on libraries.

Library	# met.	# fail	% fail	Tr. t.	Mem.
mscorlib	28,344	870	3.07%	23"	158
Sys.Core	6,988	47	0.68%	4"	96
Sys.Design	13,509	4	0.03%	20"	180
Sys	17,851	242	1.36%	21"	142
Sys.Runtime.Serial	5,624	74	1.32%	5"	86
Sys.ServiceModel	34,603	80	0.23%	34"	156
Sys.Web	28,249	38	0.13%	37"	216
Sys.Web.Extensions	4,245	0	0.00%	4"	109
Sys.Windows.Forms	28,319	53	0.19%	42"	189
Sys.XML	12,727	171	1.34%	23"	146
Total	180,460	1,579	0.87%	3’33"	

5.1 Experimental Setup

We implemented our translation from CIL to JB through (i) a C# program that translates a CIL program to an intermediate XML representation (representing Java bytecode), and (ii) a Java program that produces a `jar` file from an XML representation. We had to split the implementation in this way since the library to read CIL bytecode (`Mono.Cecil`) is written in .NET, while the library writing `jar` bytecode (`BCEL`) is written in Java. The first part of the translation (performing the real CIL to JB translation) runs in parallel on different classes through the System.Threading.Tasks library (part of the standard .Net framework). We ran our experiments on an Intel Core i5-6600 CPU at 3.30GHz machine with 16 GB of RAM, 64-bit Windows 7 Professional, and Java SE Runtime Environment version 1.8.0_111-b14.

As a first experiment to assess the efficiency and precision of our approach, we translated and analyzed the five most popular GitHub repositories (as on February 27th, 2017) written in C# and tagged as C# repositories[6]. Tab. 1 reports (i) the number of C# LOC of each projects (Column **LOC**)[7] (our benchmarks range between 22 and 120 KLOC, hence they are real world applications); (ii) the total number of methods (**# meth.**), and for how many of them the translation failed because of unsafe code (**fail**); (iii) the time (**Tr. t.**) consumed by the translation from CIL to JB; (iv) the time of Julia's analyses (**An. t.**), and (v) the number of alarms (**Al**), of false alarms because of loss of information introduced by the translation (**F**), and the precision (ratio of false alarms *w.r.t.* the total number of alarms, column **Prec.**) of Julia's analysis.

[6] We consider the number of watchers as measure of popularity of a repository. We discarded some projects *tagged* as C# that actually mostly contain native code (corefx, coreclr, mono), that did not compile in Visual Studio (roslyn, powershell), that have been dismissed (shadowsocks), or that are particularly small (wavefunction, below 1KLOC).

[7] LOC are computed with LocMetrics http://www.locmetrics.com/

```
delegate void Del(string message);
void DelegateMethod(string message) {...}
void go() {
        Del handler = DelegateMethod;
        handler("Hello World");}
```

(a) C# code

```
void go () {
        ldarg.0
        ldftn A::DelegateMethod(string)
        newobj A/Del::.ctor(object, int)
        ldstr "Hello World"
        call void A/Del::Invoke(string)}
```

(b) CIL

```
void go() {
        ldc "DelegateMethod(LString;)V"
        invokestatic
Reflection.GetMethod:(LString;)LMethod;
        ldc "Hello World"
        invokevirtual A/Del.Invoke:(LString ;)}
```

(c) JB

Figure 10: An example of CIL delegate.

In order to assess the efficiency and library coverage of our approach, we also analyzed the 10 largest (based on the size of the .dll files) system libraries of the Microsoft .Net framework version 4.0.30319. They contain unsafe code (such as cryptographic code in mscorlib.dll) and might not be compiled from C#, but possibly from VB.Net. Tab. 2 reports the number of methods of the library (**# met.**), the number and percentage of methods where the translation fails because of unsafe code (**# fail** and **% fail**), and time (**Tr. t.**) and memory (**Mem.**, in MB) for the translation.

Research Question 1: Efficiency
In 4 out of 5 top Github projects, our translation took 7" (Tab. 1); it took almost 1 minute for ShareX. These times are much shorter (overall, less than a third) than the analysis time, and the memory consumed ranged between 77MB (for Dapper) to 193MB (ShareX). The results for .Net framework libraries (Tab. 2) show a similar trend, translating 180K methods in 3'30" (that is, a bit more than 1 msec per method) consuming at most 189MB of memory.

This shows that our approach respects Research Question 1: it deals with industrial-size software in a few minutes and with a translation time comparable to the analysis time.

Research Question 2: Precision
We manually checked only the 105 high severity alarms issued by Julia on the top 5 Github projects, over a total of several thousands. Tab. 1 reports their number (column **Al**) and the number of false alarms (**F**) due to our translation. The static analysis might generate false alarms as well, for instance because of disjunctive constraints not tracked by Julia; we do not count these as false alarms, since we want to evaluate the imprecision due to the translation, and not that inherent to Julia. In particular, 6 alarms out of 105 (about 6%) are false because of imprecision introduced by the translation. This shows that our approach respects Research Question 2. The origins of this imprecision are (i) `async` and `await` statements (in particular in Dapper), and (ii) `try-catch-finally` blocks (*e.g.*, in SignalR). These would require to modify Julia to recognize these features more precisely (through automatic annotations produced by the translation).

Research Question 3: Libraries
We manually checked that all methods of the 5 top Github C# projects where our translation fails are actually unsafe. Column **# fail** in Tab. 1 shows that there is no failure for CodeHub, Dapper and Nancy. Instead, there are 3 failures for SignalR, due to unsafe methods in class `Infrastructure.SipHashBasedStringEqualityComparer`, and 14 failures for ShareX, due to unsafe methods in two classes: (i) `GreenshotPlugin.Core` has methods setting or getting colors in fast implementations of bitmaps (`UnsafeBitmap` and subclasses); (ii) `ShareX.ImageHelpers` uses unsafe classes (such as `UnsafeBitmap`). This shows that our approach fails only for unsafe code with unsafe pointer manipulation (storing pointers in fields, returning them from methods, performing pointer arithmetic). We expected to find more unsafe code in the .Net framework, but Tab. 2 shows that the translation succeeds for 99.13% of the methods, with a worst case of 96.93%. Hence, our approach respects Research Question 3.

REFERENCES

[1] Checkstyle. http://checkstyle.sourceforge.net. Accessed on March, 16th 2018.

[2] Clr to jvm. http://www.xmlvm.org/clr2jvm/. Accessed on March, 16th 2018.

[3] CodeSonar – Static Analysis SAST Software. https://www.grammatech.com/products/codesonar. Accessed on March, 16th 2018.

[4] Findbugs^{TM} – Find Bugs in Java Programs. http://findbugs.sourceforge.net/. Accessed on March, 16th 2018.

[5] Ndepend. http://www.ndepend.com. Accessed on March, 16th 2018.

[6] .Net Reflector Add-Ins. https://www.microsoft.com/en-us/research/project/spec.

[7] PMD. https://pmd.github.io/. Accessed on March, 16th 2018.

[8] WALA. http://wala.sourceforge.net/wiki/index.php/Main_Page. Accessed on March, 16th 2018.

[9] M. Ameri and C. A. Furia. Why Just Boogie? Translating between Intermediate VÂćerification Languages. In *Proceedings of IFM '16*, LNCS. Springer, 2016.

[10] R. Atkey and D. Sannella. ThreadSafe: Static Analysis for Java Concurrency. *Electronic Comm. of the European Association of Software Science and Technology*, 72, 2015.

[11] G. Barbon, A. Cortesi, P. Ferrara, and E. Steffinlongo. DAPA: degradation-aware privacy analysis of android apps. In *Proceedings of STM '16*, pages 32–46, 2016.

[12] M. Barnett, K. Leino, and W. Schulte. The Spec# Programming System: An Overview. In *Proceedings of CASSIS '04*, 2004.

[13] M. Bebenita, F. Brandner, M. Fähndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter. SPUR: a Trace-based JIT Compiler for CIL. In W. R. Cook, S. Clarke, and M. C. Rinard, editors, *Proceedings of OOPSLA '10*. ACM, 2010.

[14] G. Costantini, P. Ferrara, and A. Cortesi. A suite of abstract domains for static analysis of string values. *Softw., Pract. Exper.*, 45(2):245–287, 2015.

[15] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of POPL '77*. ACM Press, 1977.

[16] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of Principles of Programming Languages (POPL'77)*, pages 238–252, 1977.

[17] Coverity. Coverity Prevent^{TM}. http://www.coverity.com/library/pdf/coverity_prevent.pdf. Accessed on March, 16th 2018.

[18] ECMA. *Standard ECMA-335: Common Language Infrastructure (CLI)*. 2012.

[19] T. A. S. Foundation. Apache Commons BCEL. https://commons.apache.org/proper/commons-bcel. Accessed on March, 16th 2018.

[20] JetBrains. Resharper. https://www.jetbrains.com/resharper.

[21] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification, Java SE 7 Edition*. Addison-Wesley Professional, 1st edition, 2013.

[22] F. Logozzo. Cibai: An Abstract Interpretation-Based Static Analyzer for Modular Analysis and Verification of Java Classes. In *Proceedings of VMCAI '07*, LNCS. Springer, 2007.

[23] F. Logozzo and M. Fähndrich. Static Contract Checking with Abstract Interpretation. In *Proceedings of FoVeOOS '10*, LNCS. Springer, 2010.

[24] Microsoft. FxCop. https://msdn.microsoft.com/en-us/library/bb429476(v=vs.80).aspx. Accessed on March, 16th 2018.

[25] F. Spoto. The Julia Static Analyzer for Java. In *Proceedings of SAS '16*, LNCS. Springer, 2016.

[26] Wikipedia. List of Tools for Static Code Analysis. https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis#Java. Accessed on March, 16th 2018.