Verifying Functional Correctness Properties At the Level of Java Bytecode*

Marco Paganoni[1] and Carlo A. Furia[1] [0000−0003−1040−3201]

Software Institute, USI Università della Svizzera italiana, Lugano, Switzerland
marco.paganoni@usi.ch bugcounting.net

Abstract. The breakneck evolution of modern programming languages aggravates the development of deductive verification tools, which struggle to timely and fully support all new language features. To address this challenge, we present ByteBack: a verification technique that works on Java bytecode. Compared to high-level languages, intermediate representations such as bytecode offer a much more limited and stable set of features; hence, they may help decouple the verification process from changes in the source-level language.
ByteBack offers a library to specify functional correctness properties at the level of the source code, so that the bytecode is only used as an intermediate representation that the end user does not need to work with. Then, ByteBack reconstructs some of the information about types and expressions that is erased during compilation into bytecode but is necessary to correctly perform verification. Our experiments with an implementation of ByteBack demonstrate that it can successfully verify bytecode compiled from different versions of Java, and including several modern language features that even state-of-the-art Java verifiers (such as KeY and OpenJML) do not directly support—thus revealing how ByteBack's approach can help keep up verification technology with language evolution.

1 Introduction

Modern programming languages are rich in expressive features and evolve regularly, extending their capabilities with each new version of the language. These characteristics make them easier to use and ever more powerful, to the ultimate benefit of programmers using them. On the contrary, they also complicate the development of verification tools: the more features to support, and the faster a programming language evolves, the harder it is to keep up-to-date a verification toolchain. Take Java as an example of a widely used modern language. As we discuss in Sec. 5, no state-of-the-art automated Java verifier fully supports all features of the language—even for older versions such as Java 8.

In this paper, we pursue the idea of performing formal verification not at the level of a language's source code but on an intermediate representation. Our ByteBack technique processes Java bytecode to verify functional (input/output) properties expressed as pre- and postconditions. By targeting bytecode instead of source code, ByteBack seamlessly supports a wide variety of Java features that are desugared when automatically translated to bytecode by the compiler. It can even verify some programs written in other programming languages, such as Scala, that also compile to Java bytecode.

* Work partially supported by SNF grant 200021-207919 (LastMile).

Performing functional verification of bytecode entails two main challenges. First, we need to provide convenient means of expressing the specification to be verified, as well as any other intermediate annotations. Requiring the user to directly annotate the bytecode is impractical, and at odds with the goal of expressing the behavior of the original Java program. ByteBack offers a Java library (called `BBlib`) with custom annotations and static methods. Users add specifications to the Java source code by writing Java expressions that call these library methods; ByteBack then recovers the specifications by analyzing `BBlib` calls in bytecode format. Supporting expressive contract specifications of the source code is a key novelty of ByteBack compared to other approaches, such as JayHorn [50] and SMACK [49], that also verify intermediate representations but mostly target implicit, low-level correctness properties (such as the absence of memory errors) and have only limited support for arbitrary functional specifications.

Reconstructing some of the information lost during the compilation from source code to bytecode is the second main challenge tackled by ByteBack. To this end, we define a bespoke static analysis working on Grimp—an alternative representation of bytecode (offered by the Soot static analysis framework [53]). ByteBack's analysis connects bytecode instructions to elements of `BBlib` specification, and generates verification conditions that encode program correctness. Concretely, ByteBack translates Grimp code and annotations to the Boogie intermediate verification language [5], which we then use as the interface to a backend SMT solver.

We implemented ByteBack in a tool with the same name, which verifies bytecode annotated with functional specifications expressed using the `BBlib` library. We verified 40 programs, including classic verification examples (such as sorting algorithms), using numerous Java features that state-of-the-art functional verification tools do not currently support. We also verified the implementation of some of the same algorithms in Scala, thus demonstrating that ByteBack can accommodate a variety of source-code level features by focusing on the verification of an intermediate representation.

A replication package including ByteBack's implementation, and the benchmarks and examples described in the paper, is available on Zenodo [46].

```
1  @Require(forall j: int • values[j] ≠ 1)
2  @Ensure(return ≥ 0)
3  static int summary1(int[] values) {
4    int result = 0;
5    for (int k = 0; k < values.length; k++) {
6      invariant(result ≥ 0);
7      if (values[k] = 0) result += 1;
8      else if (values[k] = 1) result += -1;
9      else if (values[k] > 0) result += values[k];
10   } return result; }
```

(a) Method `summary1` uses a regular **`for`** loop and **`if`**/**`else`** conditionals.

```
11 @Require(forall j: int • values[j] ≠ 1)
12 @Ensure(return ≥ 0)
13 static int summary2(int... values) {
14   var result = 0;
15   for (var v: values) result += switch(v) {
16   invariant(result ≥ 0);
17     case 0: yield(1);
18     case 1: yield(-1);
19     default: if (v > 0) yield(v); else yield(0);
20   } return result; }
```

(b) Method `summary2` uses varargs, a "foreach" loop, a **`switch`** expression, and **`var`** local types.

Fig. 1: Annotated Java methods that compute a numeric summary of **`int`** array `values`.

2 Motivating Examples

Fig. 1a shows the implementation of a simple Java method `summary1`, which scans its input integer array `values` and returns a numeric summary of its content: it ignores all negative values, adds 1 to the summary for each element 0, subtracts 1 for each element

1, and adds any bigger positive elements. The code also embeds some annotations that specify a precondition `@Require` ("input array `values` includes no element equal to 1"), a postcondition `@Ensure` ("the returned result is nonnegative"), and a loop invariant `invariant` ("local variable `result` stays nonnegative"). Clearly, `summary1` satisfies this specification; in fact, we can easily verify Fig. 1a's code against this specification using modern verifiers for Java (such as KeY, Krakatoa, or OpenJML)—after expressing the specification using the verifier's annotation language.

Now consider method `summary2` in Fig. 1b. It's not hard to see that `summary2` implements essentially the same behavior as `summary1` but using different features of the Java language: `values` is a variadic argument (varargs) instead of a plain integer array; local variable `result` uses type inference (**`var`**) instead of declaring its type explicitly; the loop is an enhanced **`for`** loop ("foreach"); the loop's body uses a **`switch`** expression with **`yield`** instead of nested **`if`**/**`else`** conditionals. As shown in Tab. 5, these features have been added to Java only in recent versions of the language. As a result, none of the aforementioned Java verifiers that can check the correctness of `summary1` supports all the features used by `summary2`—even though the methods are essentially equivalent.

Our verification technique ByteBack, which we present in the rest of the paper, performs verification at the level of Java bytecode. One distinct advantage of this approach is that the Java compiler takes care of desugaring equivalent Java features into a simpler representation as bytecode instructions. Therefore, ByteBack verifies both variants `summary1` and `summary2` in Fig. 1 without having to explicitly add support for each new Java feature. This demonstrates that bytecode-level verification can help formal verification techniques keep up with rapidly evolving source-level languages.

3 How ByteBack Works

Fig. 2 overviews how ByteBack works, and the toolchain it implements. To verify a program, the user first annotates its *source code* with a specification using the functionalities of the `BBlib` library; Sec. 3.1 outlines this library and how it can be used. `BBlib`-annotated source code can be compiled with the Java *compiler* (or any other suitable compiler) into *bytecode*. ByteBack uses the Soot static analysis framework to transform the bytecode into the higher-level *Grimp* intermediate representation (an alternative bytecode representation that is syntactically closer to source code and retains higher-level typing information). As we explain in Sec. 3.2, ByteBack performs a static *analysis* of Grimp, with the goal of identifying the various program elements and linking them to their specification—embedded as calls to `BBlib` methods, and references to the annotations. With this information, ByteBack can *translate* program and annotations into *Boogie* code, which the Boogie verifier [5] processes to generate verification conditions, and finally determine whether the program verifies correctly.

3.1 Specifying Functional Properties

This section describes the main methods and annotations included in the `BBlib` library, and how we can use them to express the specification of a Java program.[1] Whereas

[1] `BBlib` is available as a JAR file, and hence any language that is bytecode-compatible with Java can use its features—as we'll demonstrate in some of Sec. 4's examples in Scala.

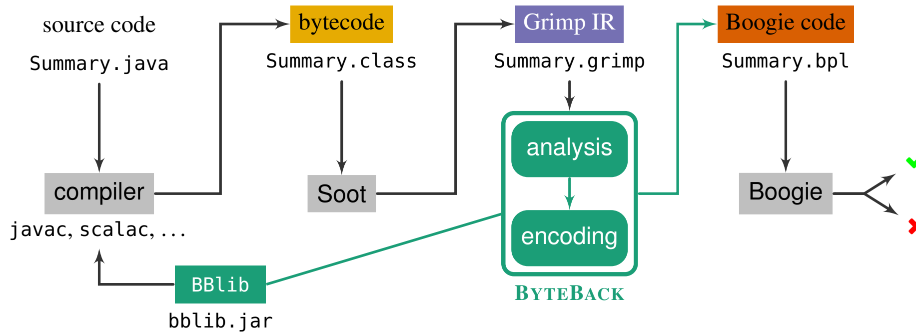

Fig. 2: An overview of how ByteBack's verification toolchain works.

Fig. 1's examples use a simplified idiomatic syntax, in this section we follow `BBlib`'s concrete syntax; Fig. 3 shows the same annotations with this concrete syntax.

```
21 @Require("no_ones")
22 @Ensure("nonnegative")
23 public static int summary(int... values);
24
25 @Predicate public static boolean no_ones(int[] values)
26 { return not(contains(values, 1, 0, values.length)); }
27
28 @Predicate public static boolean nonnegative(int[] values, int result)
29 { return gte(result, 0); }
30
31 @Pure public static boolean contains(int[] as, int e)
32 { int i = Binding.integer(); return exists(i, lte(0, i) & lt(i, as.length) & eq(as[i], e)); }
```

Fig. 3: Annotations for Fig. 1's methods using `BBlib`'s concrete syntax.

Pre- and postconditions. The main specification elements of a method m are its *precondition* and *postcondition*, encoded by adding annotations `@Require(String p)` and `@Ensure(String q)` just before m's declaration. Arguments `p` and `q` denote the name of *predicates*: methods returning **`boolean`** that encode the actual pre- and postconditions. We can annotate m with several `@Require`s and `@Ensure`s, which are implicitly conjoined. In Fig. 3's running example, we name the pre- and postcondition predicates `no_ones` and `nonnegative`.

Predicates. We mark any predicates p with annotation `@Predicate`, so that ByteBack can easily track them in the bytecode. For the same reason, a predicate p is defined in the same class as the method m it specifies. A predicate p is **`static`** iff m is, and its input signature types are the same as m's; this way, m's specification can refer to any program elements that are visible at m's interface. Since postconditions usually constrain a method's output, any predicate q used as a postcondition includes an extra input argument `result` of the same type as m's return type (if it is not **`void`**). In Fig. 3's running example, predicates `no_ones` and `nonnegative` are **`static`** methods like `summary`; the latter includes a second argument **`int result`**, which refers to the integer value returned by `summary`.

Pure expressions. A predicate's body encodes a Boolean expression that should be exactly expressible in logic. Therefore, it can only include *pure* (side-effect free) statements, and has to terminate with a single **`return`** statement that defines the overall predicate expression. In practice, this means that predicates can only *read* the global

program state but cannot modify it. However, pure methods may use local variables and may call methods that satisfy the same constraints and that we marked as `@Pure`; this includes recursive calls. For example, Fig. 3's predicate `no_ones` calls pure function `contains`.

	IN JAVA/LOGIC	IN `BBlib`
comparison	`x < y`, `x <= y`, `x == y` `x != y`, `x >= y`, `x > y`	**`lt`**`(x, y)`, **`lte`**`(x, y)`, **`eq`**`(x, y)` **`neq`**`(x, y)`, **`gte`**`(x, y)`, **`gt`**`(x, y)`
conditionals	`c ? t : e`	**`conditional`**`(c, t, e)`
propositional	`!a`, `a && b`, `a || b`, `a` $\Longrightarrow$ `b`	**`not`**`(a)`, `a & b`, `a | b`, **`implies`**`(a, b)`
quantifiers	$\forall$`x: T • P(x)` $\exists$`x: T • P(x)`	`T x = Binding.T(); `**`forall`**`(x, P(x))` `T x = Binding.T(); `**`exists`**`(x, P(x))`

Table 1: A list of `BBlib`'s aggregable operators, and the Java or logic operators that they replace.

Aggregable expressions. BYTEBACK has no access to the source code, but it should still be able to recover the pure logic expression encoded by a predicate's body after this is translated into bytecode by the compiler. When this is the case, we say that a source code expression is *aggregable*—informally, it translates into bytecode without information loss. Aggregability further constrains what we are allowed to use in a predicate's or pure function's body: *i*) Only pure expressions are allowed. *ii*) Branching statements (conditionals, loops) are not allowed, since they introduce jumps in the bytecode that may be cumbersome or impossible to render as a single logic expression. Instead, `BBlib` offers method **`conditional`**`(c, t, e)` to encode conditional *expressions*—similar to Java ternary expressions `c ? t : e` but translated to bytecode without introducing branching. *iii*) Java's usual Boolean operators (`!`, &&, ||) are not allowed because they are not aggregable: && and || are short-circuited, and hence they may introduce branching in the bytecode; expressions involving `!` may also introduce branching (e.g., `x = !y` translates to bytecode like **`if`** `(y) x = `**`false else`**` x = `**`true`**). Instead, `BBlib` offers replacement methods (**`not`**) or lets you use Java's eager Boolean operators (&, `|`, `^`) that are aggregable. *iv*) Similarly, comparison operators (<, >, ...) may introduce branching in the bytecode, and hence `BBlib` offers replacement methods (**`lt`**, **`gt`**, ...) that are aggregable. Tab. 1 summarizes the main aggregable operators provided by `BBlib` as static methods—used either instead of non-aggregable Java methods or to express common logic operators. Fig. 3 uses some of these operators to express the specification in the running example.

Frame specifications. A method's *frame* is the set of memory locations that the method may modify. BYTEBACK uses a simple approach to *infer* the frame of a method m. It performs a static analysis looking for any heap-modifying statement in m's Boogie translation. If it finds any, m's frame is the whole heap; otherwise, m's frame is empty. If this analysis determines that m's frame is non-empty but m is marked as `@Pure` or `@Predicate`, BYTEBACK reports a verification error. The analysis recursively follows any method called by m, and is set up so as to be *sound* but imprecise; for example, if m calls a method ℓ whose implementation is not available, we conservatively assume that ℓ may modify the heap. Users can still more finely specify a method's frame by adding postconditions that explicitly indicate heap locations that do *not* change. Supporting

more flexible framing methodologies [38,40,51,18,31,48] in ByteBack belongs to future work. In Fig. 3's example, ByteBack infers that `summary`'s frame is empty since its implementation only reads the content of array `values`.

Other specification elements. A method m's postcondition may include expressions **`old`**`(e)`—which denotes the value of `e` in m's pre-state. In addition, `BBlib` offers methods for common intra-method specification elements: *i*) **`invariant`**`(J)` declares a loop invariant `J`, and can be placed anywhere in the corresponding loop's body. Fig. 1 shows the loop invariant specification in the running example. *ii*) methods **`assertion`**`(E)` and **`assumption`**`(E)` introduce intermediate assertions (if `E` holds continue, otherwise fail) and assumptions (ignore states where `E` doesn't hold) that are useful to further guide the verification process of a method's implementation. As usual, the arguments `J` and `E` to these specification elements should be pure, aggregable expressions.

3.2 Translating Grimp into Boogie

This section outlines the translation from Grimp—a human-readable representation of bytecode produced by the Soot framework—to Boogie—a verification language that combines an expressive program logic with basic procedural constructs (variables, assignments, procedures). Grimp code represents *executable instructions* in a program's bytecode; in contrast, source-level *declarations* (such as class or variable declarations) are implicit in Grimp, but still accessible programmatically through Soot's API. Concretely, we present ByteBack's Boogie encoding as a translation $\mathcal{T}$ from Grimp (instructions) and Java (declarations) to Boogie code—even though this translation is actually implemented without access to Java source code. For clarity, we highlight Grimp/Java keywords (**`goto`** `l`) with a different color than Boogie keywords (**`goto`** `l`).

Heap model. ByteBack introduces a simple Boogie model of the heap adapted from Dafny's [36]—a state-of-the-art deductive verifier. The heap is a variable `#heap: Heap` that stores a polymorphic mapping of **`type`** `Heap = [Reference]`$\langle\alpha\rangle$`[Field` α`]`α from references to fields (of generic type α). To access the heap, ByteBack defines

```
function read⟨α⟩(h:Heap, r:Reference, f:Field α) returns (α)
```

that returns the value of field `f` in the object pointed to by reference `r`, and

```
function update⟨α⟩(h:Heap, r:Reference, f:Field α, v:α) returns (Heap)
```

that returns an updated heap after setting field `r.f` to `v`.

Aggregates. As we explained in Sec. 3.1, a block of code that defines an *aggregable* expression consists of statements that: *i*) are *pure* (do not modify the heap); *ii*) are straight–line (no branches); *iii*) use `BBlib`'s propositional and comparison operators (Tab. 1), or other aggregable user-defined methods. Precisely, take a sequence s of Grimp instructions that satisfy these constraints. Then, s can be written in SSA form [4] as a sequence $s_1\, s_2\, \cdots\, s_{n+1}, n \geq 0$, of statements where each $s_k, k \leq n$, is an assignment $\mathsf{v}_k = e_k$ of an aggregable expression e_k to a fresh variable v_k; and the final s_{n+1} returns the last v_n. Given any such sequence s, ByteBack builds an overall expression $\mathcal{A}(s)$ by recursively replacing each usage of v_k with its unique definition in s. We call $\mathcal{A}(s)$ the *aggregate*

```
procedure summary(values: Reference) returns (@ret: int)
requires ¬contains(#heap, values, 1, 0, lengthof(values)); ensures @ret ≥ 0;

function contains(h: Heap, as: Reference, e: int) returns(bool)
{ ∃ i: int • 0 ≤ i ∧  i ≤ lengthof(as) ∧  (array.read(h, as, i): int) = e }
```

Fig. 4: ByteBack's Boogie encoding of `summary`'s signature and `contains` in Fig. 3.

of snippet s;[2] in a nutshell, $\mathcal{A}(s)$ is a pure expression equivalent to the one returned by s, which ByteBack can translate to a Boogie logic expression as we detail below. In Fig. 3's running example, `no_ones`'s body is already in aggregate form, and hence $\mathcal{A}$(`no_ones`) = `not(contains(values, 1, 0, values.length))`. For convenience, we extend the notation: $\mathcal{A}(e)$, where e is any aggregable expression built by a sequence of statements s, denotes *expression* e in *aggregate form*—defined as $\mathcal{A}(s$`;` **`return`** $e)$.

Types. ByteBack uses Boogie type **`int`** (corresponding to mathematical integers) for all bytecode integer types **`int`**, **`short`**, **`byte`**, **`long`**, and **`char`**; Boogie type **`real`** (corresponding to mathematical reals) for floating-point types **`float`** and **`double`**; Boogie type **`bool`** for type **`boolean`**;[3] and Boogie type `Reference` for all bytecode reference types. Thus, for example, $\mathcal{T}$(**`int`**) = **`int`**, $\mathcal{T}$(**`boolean`**) = **`bool`**, and $\mathcal{T}$(**`int`**`[]`) = `Reference`.

Declarations. ByteBack declares an uninterpreted Boogie type **`const`** `C: Type` for each **`class`** `C`; and it declares a **`const`** `C.f: Field` $\mathcal{T}$(`t`) for each each field `f` of `C`—where `t` is `f`'s static type.[4] Similarly, local variables (in implementations of non-pure methods) translate to Boogie local variables: $\mathcal{T}$(`t v`) = **`var`** `v:` $\mathcal{T}$(`t`).

Specification functions. ByteBack translates to Boogie functions any methods annotated with **`@Pure`**, which denotes logic functions used in `BBlib` specifications. Boogie functions that translate specification functions include an extra argument `h` of type `Heap` since they cannot directly read global variables. The body S of **`@Pure`** methods has to be aggregable; ByteBack first builds the *aggregate* $\mathcal{A}(S)$ expression as described above, and then translates that into Boogie. Fig. 4 shows the Boogie translation of `contains` in the running example.

$$\mathcal{T}\left(\begin{array}{l}\texttt{@Pure}\\ \texttt{t}_0\ \texttt{C.p}\ (\texttt{t}_1\ \texttt{d}_1, \ldots, \texttt{t}_m\ \texttt{d}_m)\\ \{\ S\ \}\end{array}\right) = \begin{array}{l}\textbf{function}\ \texttt{C.p}\\ \quad (\texttt{h}:\texttt{Heap}, \texttt{d}_1 : \mathcal{T}(\texttt{t}_1), \ldots, \texttt{d}_m : \mathcal{T}(\texttt{t}_m))\\ \quad \textbf{returns}\ \mathcal{T}(\texttt{t}_0))\\ \{\ \mathcal{T}(\mathcal{A}(S))\ \}\end{array}$$

Methods. ByteBack translates to Boogie procedures any other methods (that is, not annotated with **`@Pure`** or **`@Predicate`**). An additional extra argument `o` of type `Reference` matches the *target* of method calls; thus, it is absent in procedures translating static methods. Methods that return a value (whose return type is not **`void`**) include a return argument named `@ret` in Boogie, which is also passed to the postcondition predicate.

---

[2] Soot also performs a kind of aggregation of Grimp expressions; however, ByteBack's aggregates are different from Soot's in general.

[3] While pure bytecode uses `0`/`1` integers to encode Booleans, the Grimp intermediate representation includes a distinct Boolean type **`boolean`**.

[4] For simplicity, the presentation assumes that identifier names are unique and the same in bytecode as in Boogie; in practice, ByteBack also takes care of renaming to avoid clashes.

Frame specifications translate to Boogie `modifies` clauses; ByteBack infers them as described in Sec. 3.1, and hence they can only be empty (the `modifies` clause is omitted) or include the whole heap (`modifies #heap`). Preconditions and postconditions translate to Boogie `requires` and `ensures` clauses as follows. Given a `@Predicate` method `p`, ByteBack first builds its aggregate expression $\mathcal{A}(\texttt{p})$; then, it translates this Grimp expression to a Boogie expression $\mathcal{T}(\mathcal{A}(\texttt{p}))$; finally, it replaces `p`'s formal arguments with the corresponding Boogie formal arguments $\texttt{d}_1, \ldots, \texttt{d}_m$.

$$\mathcal{T}\left(\begin{array}{l}\texttt{@Require("p")}\\ \texttt{@Ensure("q")}\\ \texttt{t}_0\ \texttt{C.m}\ (\texttt{t}_1\ \texttt{d}_1, \ldots, \texttt{t}_m\ \texttt{d}_m)\\ \{\ B\ \}\end{array}\right) = \begin{array}{l}\textbf{procedure}\ \texttt{C.m}\\ \ (\texttt{o}:\texttt{Reference}, \texttt{d}_1 : \mathcal{T}(\texttt{t}_1), \ldots, \texttt{d}_m : \mathcal{T}(\texttt{t}_m))\\ \quad \textbf{returns}\ (\texttt{@ret} : \mathcal{T}(\texttt{t}_0))\\ \quad \textbf{requires}\ \mathcal{T}(\mathcal{A}(\texttt{p}))[\texttt{d}_1, \ldots, \texttt{d}_m]\\ \quad \textbf{ensures}\ \mathcal{T}(\mathcal{A}(\texttt{q}))[\texttt{d}_1, \ldots, \texttt{d}_m, \texttt{@ret}]\\ \quad \textbf{modifies}\ \mathcal{F}(B)\\ \{\ \mathcal{T}(B)\ \}\end{array}$$

Fig. 4 shows the Boogie translation of `summary`'s signature and specification. Why does ByteBack translate postconditions in this way (inlining aggregate specification expressions), instead of just using the Boogie functions that translate postcondition predicates—such as `nonnegative(values, @result)` for `summary`'s postcondition? In general, postconditions may use `old` to refer to an expression's value in the pre-state; Boogie offers an `old` operator, but only accepts it explicitly in an `ensures`, not in user-defined functions. Therefore, a postcondition `@Ensure("inc")`, where predicate `inc` is declared as `@Predicate boolean inc(){return gt(x, old(x));}` can only be translated as `ensures read(#heap,this,C.x)` $>$ `old(read(#heap,this,C.x))`—not as `ensures inc(#heap)`, since `inc`'s body may not use `old`.

Constructors may also have a specification. ByteBack translates them like special methods that return a fresh (previously unallocated) reference in the heap to the created object—as specified by an automatically generated postcondition. To this end, ByteBack supplies Boogie procedures `new` and `array.new` to create new references, which translate bytecode instructions `new` and `newarray`. Then, actual constructor calls (`invokespecial` in bytecode) translate like normal procedure calls—as shown below.

Expected types. Expression types in Grimp mirror strictly the bytecode instructions they correspond to. This may lead to Soot attributing to a Grimp expression e an unnecessarily general type t when e is actually only used according to a more specific type t'. For example, the type of Grimp expression a `&` b is `int` according to Soot even if a and b are of type `boolean`. To have more specific types in these scenarios, ByteBack reconstructs the *expected type* $\mathbb{T}(e)$ of any Grimp expression e based on where e is used. Thus, if e is the right-hand side of an assignment $\texttt{v} = e$, $\mathbb{T}(e)$ is `v`'s type; if e is returned by a method m, $\mathbb{T}(e)$ is m's return type according to its signature; if e is the actual argument in a call to m, $\mathbb{T}(e)$ is m's formal argument type. Therefore, $\mathbb{T}(a\ \texttt{\&}\ b)$ is `boolean` as long as a `&` b is used as a Boolean.

Variable access. Tab. 2 summarizes the translation of reading and writing variables (local, instance, static, and array). Local variables are straightforward, as they also are local variables in Boogie. *Fields* of objects in the heap are read and written by calling predefined Boogie functions `read` and `heap.write` introduced earlier in this section.

GRIMP: e	BOOGIE: $\mathcal{T}(e)$	
`v`	`v`	local variable read
`o.f`	`read(#heap, o, f)`	instance field read
`C.f`	`read(#heap, type2ref(C), f)`	static field read
`a[`k`]`	`array.read(#heap, a,` $\mathcal{T}(k)$`)` : $\mathcal{T}(\mathbb{T}($`a[`k`]`$))$	array read
`v` = e	`v` $:= \mathcal{T}(e)$	local variable write
`o.f` = e	`#heap` := `update(#heap, o, f,` $\mathcal{T}(e)$`)`	instance field write
`C.f` = e	`#heap` := `update(#heap, type2ref(C), f,` $\mathcal{T}(e)$`)`	static field write
`a[`k`]` = e	`#heap` := `array.update(#heap, a,` $\mathcal{T}(k)$, $\mathcal{T}(e)$`)`	array write

Table 2: Boogie translation of read and write of variables in Grimp bytecode.

Unqualified field accesses `f` translate as qualified accesses `this.f` on `this`—which corresponds to some variable of type `Reference` in Boogie. The same functions `read` and `heap.write` also work for *static* field accesses: to this end, BYTEBACK supplies

```
function type2ref(class: Type) returns(Reference)
```

mapping each class type to a reference to a heap object that stores the static state. *Arrays* are also heap objects, but BYTEBACK provides custom functions `array.read` and `array.update` to access these objects by means of an index expression of type **`int`**.

```
function array.read⟨α⟩(h: Heap, a: Reference, k: int) returns (α)
```

As shown in Tab. 2, BYTEBACK casts (Boogie ‘:’ operator) the output of polymorphic `array.read` to array type $\mathbb{T}($`a[`k`]`$)$. This is not necessary for field accesses, since `read`’s output type parameter α is constrained by the input `f`; in contrast, `array.read` is only type-generic in the output, and hence usage context determines the concrete value of α.

Calls. Bytecode offers five call *instructions*: **`invokestatic`** (to call **`static`** methods), **`invokevirtual`** (instance methods), **`invokeinterface`** (abstract interface calls), **`invokespecial`** (constructors and **`super`** calls), and, since Java 7, **`invokedynamic`** (lambdas). BYTEBACK translates all such call *instructions* to Boogie procedure calls:[5]

$$\mathcal{T}(\texttt{invokevirtual o.m}(e_1,\ldots,e_n)) = \texttt{call C.m(o,}\ \mathcal{T}(e_1),\ldots,\mathcal{T}(e_n))$$

As usual, `C` is `m`’s class, and `o` is a reference to an instance of this class. The same translation works, with obvious adjustments, for the other kinds of call instructions—except **`invokedynamic`**, which BYTEBACK doesn’t currently support. Henceforth, **`invoke`** denotes any of the four supported bytecode call instructions.

Branching. BYTEBACK translates branching instructions (**`return`**, **`goto`**, and **`if`**) into the corresponding Boogie statements as shown in Tab. 3a. While Boogie also offers structured conditionals and loops, BYTEBACK does not use them since bytecode does not have structured programming constructs.

Literals. BYTEBACK translates any literal ℓ to a Boogie literal according to its expected type $\mathbb{T}(\ell)$. In particular, $\mathcal{T}(\texttt{0}) = \texttt{false}$ and $\mathcal{T}(\texttt{1}) = \texttt{true}$ when the expected type of integer literals `0` and `1` is **`boolean`**.

[5] Thus, BYTEBACK relies on Boogie’s *modular* semantics of calls: the only effects of calling a method `m` are what `m`’s specification prescribes. This is a standard choice in deductive verification, since it supports modularity and is consistent with the Liskov substitution principle [42].

GRIMP: e	BOOGIE: $\mathcal{T}(e)$
`return` v	`@ret` $:= \mathcal{T}(v)$; `return`
`goto` ℓ	`goto` ℓ
`if` (c) B	`if` ($\mathcal{T}(c)$) {$\mathcal{T}(B)$}

(a) Boogie encoding of Grimp bytecode branching instructions.

GRIMP: e	BOOGIE: $\mathcal{T}(e)$	
`neg` a	$\neg\mathcal{T}(a)$	not
a `^` b	$\mathcal{T}(a) \neq \mathcal{T}(b)$	xor
a `&` b	$\mathcal{T}(a) \wedge \mathcal{T}(b)$	and
a `|` b	$\mathcal{T}(a) \parallel \mathcal{T}(b)$	or
`implies`(a,b)	$\mathcal{T}(a) \Longrightarrow \mathcal{T}(b)$	implies
a `==` b	$\mathcal{T}(a) \Longleftrightarrow \mathcal{T}(b)$	iff

(b) Boogie encoding of Grimp bytecode Boolean operators. Grimp expressions a and b have expected type `boolean` $= \mathbb{T}(a) = \mathbb{T}(b)$.

GRIMP: e	BOOGIE: $\mathcal{T}(e)$
`assertion`(b)	`assert` $\mathcal{T}(\mathcal{A}(b))$
`assumption`(b)	`assume` $\mathcal{T}(\mathcal{A}(b))$
`forall`(`v`, b)	$\forall$ `v`: $\mathcal{T}(\mathbb{T}(\mathtt{v})) \bullet \mathcal{T}(\mathcal{A}(b))$
`exists`(`v`, b)	$\exists$ `v`: $\mathcal{T}(\mathbb{T}(\mathtt{v})) \bullet \mathcal{T}(\mathcal{A}(b))$
`conditional`(b,T,E)	`if` $\mathcal{T}(\mathcal{A}(b))$ `then` $\mathcal{T}(\mathcal{A}(T))$ `else` $\mathcal{T}(\mathcal{A}(E))$

(c) Boogie encoding of `BBlib` specification methods and expressions. Variables `v` are created by methods of class `Binding`.

Table 3: BYTEBACK translation of branching, Boolean operators and specification elements.

Expressions. Most arithmetic and comparison operators `+`, `-`, `*`, `==`, `!=`, `<`, `<=`, `>=`, `>` translate to their Boogie counterparts $+, -, *, =, \neq, <, \leq, \geq, >$ as obvious: $\mathcal{T}(a \bowtie b) = \mathcal{T}(a)\ \mathcal{T}(\bowtie)\ \mathcal{T}(b)$. The division operator `/` translates to `div` or `/` in Boogie according to whether it represent integer or floating-point division: $\mathcal{T}(a$ `/` $b) = \mathcal{T}(a)$ `div` $\mathcal{T}(b)$ if $\mathbb{T}(a$ `/` $b) =$ `int`; otherwise $\mathcal{T}(a$ `/` $b) = \mathcal{T}(a)$ `/` $\mathcal{T}(b)$. BYTEBACK introduces and axiomatizes an overloaded Boogie function `cmp` to translate bytecode operator `cmp`: $\mathcal{T}(a$ `cmp` $b) =$ `cmp`$(\mathcal{T}(a),\mathcal{T}(b))$ returns `1` if $a > b$, `-1` if $a < b$, and `0` if $a = b$. Tab. 3b displays how BYTEBACK translates Grimp Boolean operators to Boogie. Java's short-circuited operators `&&` and `||` are not listed in the table, as the compiler desugars them into *conditional instructions* in bytecode; for example, `if (a && b) x = 1`... becomes `if (a == 0) goto end; if (b == 0) goto end; x = 1; end:`... in bytecode.

Since `boolean` is a subtype of `int` in Soot, the operands of Boolean operator expressions (e.g., a `==` b) may have different types (e.g., $\mathbb{T}(a) =$ `int` but $\mathbb{T}(b) =$ `boolean`—usually when a is used as an integer in other parts of the program). In these cases, BYTEBACK translates everything using the most general type `int`, so that all usages of the operands can be uniformly represented in Boogie (where the `bool` and `int` types are disjoint, as they are in Java).

Call expressions. Boogie does not allow procedure calls in expressions;[6] therefore, BYTEBACK saves the call value in a fresh variable, and replaces the call expression with a read of the variable: given a Grimp expression e, used in statement s, that includes a call `invoke o.m()` (virtual, static, or interface) to a method `m`, BYTEBACK first adds the statements `var #r`:$\mathcal{T}(\mathbb{T}($`invoke o.m`$))$; `call #r` $:= \mathcal{T}($`invoke o.m()`$)$ just before s, and then translates e into $\mathcal{T}(e[$`invoke o.m` $\mapsto$ `#r`$])$—replacing the call with `#r`.

[6] In contrast, calls to pure methods, translated to Boogie functions, can be directly transliterated to Boogie (pure) expressions.

Specifications. ByteBack recognize `BBlib` operators and translates them to their counterparts in Boogie, as shown in Tab. 3c. Source-code **`while`** and **`for`** loops become conditional jumps in bytecode. Using Soot's static analysis capabilities, ByteBack identifies any loop in bytecode by its *head*, *backjump*, and *exit* locations. Thus, a source-code loop `while (!c)` L; R corresponds to bytecode structured as in Fig. 5a's left-hand side, where labels head, back, and exit mark the head, exit, and backjump locations; c is the loop's exit condition, B is the loop body, and R is the code that follows the loop. Any loop invariant J would be declared by a call `invariant(`J`)` to `BBlib` method **`invariant`** inside B. ByteBack encodes the semantics of loop invariants by means of suitable assumptions and assertions, as in Fig. 5a's right-hand side; then, it translates the annotated branching code to Boogie as usual. Fig. 5b shows a concrete example of how ByteBack encodes loops and invariants; note the aggregation (inlining) of the invariant predicate, which ensures that all its dependencies are replicated in each assertion and assumption in Boogie.

```
head: if (c) goto exit            assert(A(J)); head: if (c) goto exit
B[··· invariant(J) ···]           assume(A(J)); B[···]
back: goto head                   assert(A(J)); back: goto head
exit: R                           exit: assume(A(J)); R
```

(a) How ByteBack injects loop invariant checks (right) into Grimp bytecode loops (left). B denotes the loop body, which includes an invariant declaration (left).

```
for (int k = 0; k < 3; k++)        assert(0 ≤ k ∧ k ≤ 3);
{  boolean a = lte(0, k);          head : if (k ≥ 3) goto exit;
   boolean b = lte(k, 3);          assume(0 ≤ k ∧ k ≤ 3); k := k + 1;
   invariant(a & b); }             assert(0 ≤ k ∧ k ≤ 3); back: goto head;
                                   exit: assume(0 ≤ k ∧ k ≤ 3);
```

(b) An example of an annotated loop in Java (left), and its Boogie encoding produced by ByteBack (right).

Fig. 5: ByteBack's encoding of loops and loop invariants.

3.3 Implementation Details

We implemented ByteBack in a tool with the same name, written in about 11 thousand lines of Java code (plus another 52 kLOC of generated code). ByteBack's core uses the Soot static analysis framework [53] to process the bytecode to be verified, as we described in Sec. 3.2 at a high level. After analyzing the Grimp bytecode, ByteBack has collected all the information to generate Boogie code; to this end, a visitor pattern implementation creates a Boogie AST, and then dumps it into a Boogie file.

We developed the Boogie AST library using the metacompilation framework JastAdd [23], in combination with JFlex and Beaver[7] to parse Boogie source code. This capability is useful to: *i*) flexibly generate the heap model (Sec. 3.2) and other Boogie background definitions from a human-readable Boogie template file; *ii*) perform some analyses directly on the generated Boogie code (most notably, the frame inference briefly described in Sec. 3.1).

Features and limitations. ByteBack's current implementation supports most bytecode features but not exception handling and **`invokedynamic`** (which limits reasoning about lambdas); strings are supported as plain objects, which precludes precisely analyzing their semantics in Java; as we discussed previously, numeric types are encoded as infinite-precision numbers (integers and reals), which entails that ByteBack may

[7] JFlex: `https://jflex.de/`; Beaver: `http://beaver.sourceforge.net/`.

miss overflow and other numerical errors. Adding support for these features is possible in principle, and would require a combination of extending the Boogie encoding (for example, to include exceptional behavior), ByteBack's static analysis (for example, to identify the bootstrap methods that dynamically generate targets of `invokedynamic`), and `BBlib`'s features (for example, to support model-based specifications).

Sec. 3.1 described the features offered by ByteBack's `BBlib` specification library. Its current implementation is sufficient to specify a variety of examples (see Sec. 4) but lacks advanced features to express complex framing conditions and ghost code (specification code discarded during compilation), and to flexibly reuse specifications with inheritance and modularity. Supporting these features belongs to future work, also because it would require tackling challenges largely orthogonal to the focus of ByteBack.

As we demonstrate in Sec. 4, `BBlib`'s features can also specify programs written in Scala, leveraging its bytecode-level interoperability with Java. However, `BBlib` was developed with focus on Java, and hence its practical usability on Scala is more limited. In particular, the Scala compiler automatically generates features (such as setters and getters for fields) that are implicit in Scala source code; hence, users cannot directly annotate these features using `BBlib`. Addressing these limitations is possible, but would have to cater somewhat to the peculiarities of Scala (or other languages to be supported).

4 Experiments

In our experiments, we ran ByteBack on several examples in order to demonstrate that it can verify programs with different characteristics, which exercise various features of the Java programming language (including recent versions), as well as a few programs written in other languages built on top of Java bytecode.

4.1 Programs

Tab. 4 lists the 40 programs that we used for our experiments, and their size in non-empty lines of SOURCE code (LOC), as well as their size after compilation to BYTECODE (also in LOC of the representation returned by `javap -c`). The sizes include the annotations in `BBlib`, which specify the properties to be verified.

The majority of programs (32/40) use various features of Java 8; but we also included 4 programs using Java 17 features, and 4 Scala programs. The selection includes relatively simple programs that specifically target language features of Java (examples 1–16 and 33–35), classic algorithms and procedures (examples 17–27, 36, and 37–38), and object-oriented features (examples 28–32 and 39–40). Some examples implement the same algorithm for data structures with different types (e.g., **`double`** and **`int`** arrays).

We selected these examples to demonstrate that ByteBack can process a variety of modern Java features, including several that state-of-the-art Java deductive verifiers do not support (as we discuss in Sec. 5). It's important to stress that we are not comparing ByteBack's verification capabilities to those of much more mature tools such as KeY, Krakatoa, and OpenJML. We picked the features in Tab. 4's examples specifically to demonstrate that it's hard for source-level verifiers to keep up with the plethora of language features that are introduced over time—not to solve verification challenges. As we discuss in Sec. 3, ByteBack does not support all used features of Java (in particular, exceptions) and its specification capabilities (in particular, framing) are currently limited compared to source-level tools. The experiments only demonstrate our claim

#	EXPERIMENT	LANGUAGE	ENCODING		VERIFICATION	SOURCE	BYTECODE	BOOGIE
			TIME [s]	BYTEBACK	TIME [s]	SIZE [LOC]		
1	Array Operations	Java 8	2.8	10%	1.16	36	103	148
2	Boolean Operations	Java 8	3.6	9%	1.34	57	85	157
3	Control Flow	Java 8	2.8	8%	1.33	74	123	219
4	Enhanced For	Java 8 F	2.9	8%	1.25	25	52	107
5	Field Access	Java 8	2.8	6%	1.18	29	32	96
6	Floating-Point Operations	Java 8	2.9	8%	1.21	37	52	110
7	Instance Field	Java 8	3.0	6%	1.15	18	16	98
8	Integer Operations	Java 8	3.1	12%	1.45	202	332	250
9	Multiclass	Java 8	3.0	8%	1.19	14	14	113
10	Quantifiers	Java 8	3.1	6%	1.15	25	28	92
11	Static Field	Java 8	4.4	9%	1.82	32	66	146
12	Static Initializer	Java 8	3.7	6%	1.63	14	14	91
13	Static Method Calls	Java 8	3.0	8%	1.19	32	40	112
14	Switch	Java 8	3.1	6%	1.23	23	25	109
15	Unit	Java 8	2.9	6%	1.15	13	12	97
16	Virtual Method Calls	Java 8	3.0	7%	1.21	31	40	122
17	GCD	Java 8	3.0	9%	1.15	41	88	127
18	Insertion Sort `double`	Java 8	2.9	11%	2.53	49	132	147
19	Insertion Sort `int`	Java 8	3.1	11%	1.70	49	131	147
20	Linear Search	Java 8 T	2.9	10%	1.14	60	126	164
21	Max `double`	Java 8	3.0	9%	1.15	45	92	92
22	Max `int`	Java 8	2.9	10%	1.16	45	90	126
23	Selection Sort `double`	Java 8	3.1	13%	4.56	87	231	172
24	Selection Sort `int`	Java 8	3.0	12%	3.56	87	230	172
25	Square Sorted Array	Java 8	2.9	10%	1.14	54	123	140
26	Sum `double`	Java 8	3.0	8%	1.16	35	70	124
27	Sum `int`	Java 8	2.8	8%	1.14	35	70	124
28	Generic List	Java 8 G, T	3.1	9%	1.17	46	68	134
29	Binary Search	Java 8	3.1	10%	1.13	51	124	131
30	Comparator	Java 8	3.0	10%	1.24	51	30	188
31	Dice	Java 8 D	3.0	10%	1.17	41	25	129
32	Counter	Java 8	2.9	8%	1.20	33	62	150
33	Pattern matching	Java 17 P	3.0	7%	1.14	18	26	105
34	Switch Expressions	Java 17 S, Y	2.9	8%	1.16	23	56	135
35	Type Inference	Java 17 L	3.0	7%	1.15	29	59	116
36	Summary	Java 17 S, Y, F, L, A	3.1	10%	1.23	47	88	137
37	GCD	Scala 3	3.5	8%	1.31	46	93	130
38	Linear Search	Scala 3	3.4	10%	1.26	69	126	168
39	Comparator	Scala 3	3.4	9%	1.42	49	35	237
40	Dice	Scala 3	4.5	11%	1.85	38	25	223
	total		133.1		60.89	1 790	3 234	5 585
	average		3.1	9%	1.42	45	81	140

Table 4: Verification experiments performed with BYTEBACK. In each row: the EXPERIMENT's name; its source LANGUAGE (and any of Tab. 5's features it uses); the ENCODING TIME (seconds) and its percentage directly attributable to BYTEBACK (excluding Soot's initialization time); the VERIFICATION TIME (seconds) of running Boogie on the encoding generated by BYTEBACK; the size (in non-blank lines of code) of the SOURCE code, of the BYTECODE (as printed by `javap -c`), and of the BOOGIE code.

that verification at the level of bytecode has some distinctive advantages for supporting language evolution, and hence it can complement source-level verification.

4.2 Results

All the experiments ran BYTEBACK on a Fedora 36 GNU/Linux machine with an Intel i7-7600U CPU (2.8GHz), running Boogie 2.15.7.0, Z3 4.11.1.0, and Soot 4.3.0. To account for possible measurement noise, we repeated each experiment 5 times and report the mean of the wall-clock running times in the 95th percentile.

We ran Boogie with default options—except for programs 4 and 36, where we enabled option `/infer:j`, which can infer simple loop invariants. This is useful to handle these programs' enhanced **`for`** loops: translated to bytecode, a loop such as `for(var v: values)` in Fig. 1b introduces an index variable `int k` to iterate over array `values`; however, `k` does not exist in the source code, and hence one cannot annotate the loop with a suitable invariant for `k` and must rely on inferring it.

All the experiments in Tab. 4 verified successfully without errors. The running time of BYTEBACK (column ENCODING TIME) is generally short and predictable: 3.1 seconds per example on average. This time measures BYTEBACK's analysis of bytecode and translation to Boogie; it excludes the compilation time (from Java/Scala to bytecode) and the running time of Boogie (reported separately in column VERIFICATION TIME). Column ENCODING BYTEBACK reports the percentage of encoding time after we deduct Soot's fixed context initialization time: BYTEBACK's net average analysis time is a small fraction of the total (just 0.28 seconds per example).

The running time of Boogie (column VERIFICATION TIME) on BYTEBACK's output is also moderate: 1.4 seconds per example on average. There are a few outliers: the two variants of Selection Sort take up to 5 seconds to verify. This is because Selection Sort's implementation calls another method to compute the minimum value in an array range; this introduces more modular verification work. In contrast, Insertion Sort's implementation uses two nested loops, which results in a simpler Boogie program.

If we compare Tab. 4's two rightmost columns, we notice that the size of the Boogie code is roughly proportional to the size of the bytecode (Kendall's $\tau = 0.46$). Boogie code is about 1.8 times larger, as BYTEBACK's aggregation process reconstructs complex higher-level expressions. The size difference is especially pronounced for programs focusing on object-oriented features (examples 28–32 and 39–40): such features are desugared in bytecode, but "resurface" in the form of Boogie axioms and functions.

These experiments demonstrate BYTEBACK's current capabilities. Its Boogie encoding is fairly standard (as mentioned in Sec. 3.2, its heap model is taken from Dafny's) but could be optimized for better performance (e.g., improving triggers [22,39,15]) or for conciseness (e.g., further simplifying type conversions [47]) as needed.

5 Related Work

We summarize related work in the areas most relevant to BYTEBACK: source-level deductive verifiers for Java, and verifiers that target intermediate representations.

Source-level deductive verifiers for Java. Performing deductive verification of functional properties on a program's source code is a widespread approach, as that's where a specification and other kinds of information are readily available and naturally expressible. Among the many source-level verifiers for realistic programming languages—e.g., [30,32,17,6,12,11,36,25]—here we focus on KeY [1,2], Krakatoa [41], and OpenJML [19]: state-of-the-art verifiers for the functional correctness of Java sequential programs with a high degree of automation.

OpenJML and Krakatoa follow the so-called auto-active approach [37]—where the verifier generates verification conditions (VCs) and dispatches them to an automated theorem prover, but the user still indirectly guides the verifier by interactively supplying annotations. OpenJML generates VCs in SMT-LIB format [7], and dispatches them to any SMT solver like Z3 [43] or CVC4 [8]. Krakatoa translates the source program into

Feature	Java version	Example	Support KeY	Krakatoa	OpenJML
G Generic classes	5	`class Box<T> {  T value; }`	!	✗	✓
F Enhanced **for** loop	5	`int[] arr; int res = 0;` `for(int x: arr) res += x;`	✓	✗	✓
A Varargs	5	`int first(int... values)` `{ return values[0]; }`	✓	✗	✗
T Generic type inference	7	`Box<Integer> b = new Box<>();`	✗	✗	✗

D Default methods	8	`interface PlusMinus extends Plus`
		`{ default int minus(int x)`
		`{ return plus(-x); } }`
			✓	✗	✓

L Local type inference	10	`var b = new Box<Integer>();`	✗	✗	✗

S Switch expressions	12	`System.out.println( switch (day)`
		`{ case 0 -> "Mon";`
		`default -> "Other"; });`
			✗	✗	✗

Y Switch expressions with **yield**	13	`System.out.println( switch (day)`
		`{ case 0: m++; yield "Mon";`
		`default: yield "Other"; });`
			✗	✗	✗

P Pattern matching with **instanceof**	14	`if (obj instanceof String str)` `return str + " is String";`	✗	✗	✗

Table 5: Features of the Java language, and which source-code verifiers support them. For each FEATURE: the Java major VERSION when it was introduced, an EXAMPLE snippet of code using the feature, and which Java verifier among Key, Krakatoa, and OpenJML supports (✓), partially supports (!), or does not support (✗) the feature.

the WhyML intermediate verification language IVL, and delegates the generation of VCs to the Why3 system [24]. Using an IVL to generate VCs is an approach pioneered by Spec# [6] and used nowadays by many systems (including BYTEBACK). KeY is built on top of an interactive prover for Java dynamic logic [3]—used as its intermediate representation—but offers features that increase the automation level in practice.

KeY, Krakatoa, and OpenJML all use JML [34] as specification language (more precisely, different variants/subsets of JML [14]). Despite being applicable to verify real-world Java code, they also differ in the subset of Java that they support: Tab. 5 lists several modern features of the Java language and which verifier can analyze them. We compiled the table by reading the tools' official documentation and papers, and by trying out the latest tool versions that are publicly available. It should be clear that this summary is not a criticism of KeY, Krakatoa, or OpenJML—which are state-of-the-art, mature tools with proven applicability to complex verification problems—nor a direct comparison with BYTEBACK. To compile Tab. 5, we actively looked for Java recent feature "variants" that may be cumbersome to support at the source-code level, but are essentially syntactic sugar. Since BYTEBACK easily supports these features by piggybacking off the compiler's bytecode translation, this substantiates our claim that keeping verification tools up to pace with language evolution is practically hard and time-consuming at the source-code level, but substantially easier at the bytecode level.

The difference in feature support reflects the tools' intended verification target. Krakatoa focuses on supporting complex functional specifications of a core subset of Java; thus, it ignores several features that have been available since Java 5 (released in 2002). KeY and OpenJML aim at verifying complex, realistic Java applications [26,28,13,20]; to this end, they enjoy a broader language support and at least parse all Java features up until version 8 (released in 2014); however, several widely

used features are still not available for verification with these tools. For example, KeY relies on an external tool to erase generics and replace them with type `Object` and suitable casts; OpenJML natively supports generics but not all related features—such as the diamond operator <>. Since Java switched to a biannual release schedule, the gap between available language features and verification support has been widening [21].

Verifiers for intermediate representations. Approaches targeting the verification of intermediate representations (IRs) have been introduced in recent years, including SeaHorn [27] and SMACK [49] for LLVM bitcode [33], and JayHorn [50] for Java bytecode. A key difference between ByteBack and these tools are the kinds of properties they are equipped to verify: SMACK, SeaHorn, and JayHorn mainly target low-level implicit correctness properties (such as the absence of unreachable code, null pointer dereferences, and out-of-bound accesses); users can still add simple inline assertions, but there is no support for complex and structured specification elements such as contracts. SeaHorn and JayHorn encode IR instructions into constrained Horn clauses [10], a logic that can be automatically analyzed with symbolic model-checking techniques. This is consistent with these tools' intended usage, as it requires fewer annotations (loop invariants can often be inferred automatically) but also somewhat restricts the properties that can be verified in practice. SMACK, like ByteBack, translates an IR into Boogie programs to perform verification; despite these similarities, it mainly targets the verification of low-level (e.g., embedded) programs [52,29,54] and properties; it defaults to bounded verification (full, unbounded verification is only experimentally supported).

Proof-carrying code [45] is another application of verification techniques to IRs. To ensure a safe execution, compiled programs are distributed with embedded proofs, which the runtime environment checks before starting execution. Due to the difficulty of verifying IRs, proof-carrying code was primarily used for restricted properties such as memory safety. *Proof-transformation* approaches [44,9] overcome this issue by first verifying source-level annotated program "as usual", and then transforming the correctness proofs into proof-carrying IR code [35]. The BML notation takes a different approach [16] to directly annotate bytecode with expressive JML-like specifications.

6 Conclusions

We presented ByteBack, a technique that formally verifies functional source-code properties by working on Java bytecode. In our experiments, we verified programs written in Java that use recently introduced features that even state-of-the-art verifiers do not fully support; as well as some programs written in Scala that ByteBack can also analyze after compiling to bytecode. This suggests that our approach can help simplify keeping up with the evolution of modern programming languages, which regularly add new expressive features that are substantially simplified by compilation to bytecode.

References

1. Ahrendt, W., Beckert, B., Bruns, D., Bubel, R., Gladisch, C., Grebing, S., Hähnle, R., Hentschel, M., Herda, M., Klebanov, V., Mostowski, W., Scheben, C., Schmitt, P.H., Ulbrich, M.: The KeY Platform for Verification and Analysis of Java Programs. In: Giannakopoulou, D., Kroening, D. (eds.) Verified Software: Theories, Tools and Experiments - 6th International Conference, VSTTE 2014, Vienna, Austria, July 17-18, 2014, Revised Selected Papers. Lecture Notes in Computer Science, vol. 8471, pp. 55–71. Springer (2014).

https://doi.org/10.1007/978-3-319-12154-3_4, https://doi.org/10.1007/978-3-319-12154-3_4
2. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification—The KeY Book, Lecture Notes in Computer Science, vol. 10001. Springer (2016). https://doi.org/10.1007/978-3-319-49812-6, http://dx.doi.org/10.1007/978-3-319-49812-6
3. Ahrendt, W., de Boer, F.S., Grabe, I.: Abstract object creation in dynamic logic. In: Cavalcanti, A., Dams, D. (eds.) FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5850, pp. 612–627. Springer (2009). https://doi.org/10.1007/978-3-642-05089-3_39, https://doi.org/10.1007/978-3-642-05089-3_39
4. Appel, A.W.: Modern compiler implementation. Cambridge University Press, 2nd edn. (2002)
5. Barnett, M., Chang, B.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A Modular Reusable Verifier for Object-Oriented P rograms. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures. Lecture Notes in Computer Science, vol. 4111, pp. 364–387. Springer (2005). https://doi.org/10.1007/11804192_17, https://doi.org/10.1007/11804192_17
6. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: the Spec# experience. Commun. ACM **54**(6), 81–91 (2011). https://doi.org/10.1145/1953122.1953145, https://doi.org/10.1145/1953122.1953145
7. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org (2016)
8. Barrett, C.W., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 171–177. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_14, https://doi.org/10.1007/978-3-642-22110-1_14
9. Barthe, G., Grégoire, B., Pavlova, M.: Preservation of proof obligations from Java to the Java virtual machine. In: Automated Reasoning, 4th International Joint Conference, IJCAR. Lecture Notes in Computer Science, vol. 5195, pp. 83–99. Springer (2008). https://doi.org/10.1007/978-3-540-71070-7_7
10. Bjørner, N.S., Gurfinkel, A., McMillan, K.L., Rybalchenko, A.: Horn clause solvers for program verification. In: Fields of Logic and Computation II – Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday. Lecture Notes in Computer Science, vol. 9300, pp. 24–51. Springer (2015). https://doi.org/10.1007/978-3-319-23534-9_2, https://doi.org/10.1007/978-3-319-23534-9_2
11. Blanc, R., Kuncak, V., Kneuss, E., Suter, P.: An overview of the Leon verification system: verification by translation to recursive functions. In: Proceedings of the 4th Workshop on Scala, SCALA@ECOOP 2013, Montpellier, France, July 2, 2013. pp. 1:1–1:10. ACM (2013). https://doi.org/10.1145/2489837.2489838, https://doi.org/10.1145/2489837.2489838
12. Blom, S., Huisman, M.: The VerCors Tool for Verification of Concurrent Programs. In: Jones, C.B., Pihlajasaari, P., Sun, J. (eds.) FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8442, pp. 127–131. Springer (2014). https://doi.org/10.1007/978-3-319-06410-9_9, https://doi.org/10.1007/978-3-319-06410-9_9

13. de Boer, M., de Gouw, S., Klamroth, J., Jung, C., Ulbrich, M., Weigl, A.: Formal Specification and Verification of JDK's Identity Hash Map Implementation. In: ter Beek, M.H., Monahan, R. (eds.) Integrated Formal Methods - 17th International Conference, IFM 2022, Lugano, Switzerland, June 7-10, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13274, pp. 45–62. Springer (2022). https://doi.org/10.1007/978-3-031-07727-2_4, https://doi.org/10.1007/978-3-031-07727-2_4
14. Boerman, J., Huisman, M., Joosten, S.J.C.: Reasoning about JML: differences between KeY and OpenJML. In: Integrated Formal Methods – 14th International Conference, IFM 2018, Maynooth, Ireland, September 5-7, 2018, Proceedings. Lecture Notes in Computer Science, vol. 11023, pp. 30–46. Springer (2018). https://doi.org/10.1007/978-3-319-98938-9_3, https://doi.org/10.1007/978-3-319-98938-9_3
15. Chen, Y., Furia, C.A.: Triggerless happy – intermediate verification with a first-order prover. In: iFM. Lecture Notes in Computer Science, vol. 10510, pp. 295–311. Springer (2017)
16. Chrzaszcz, J., Huisman, M., Schubert, A.: BML and Related Tools. In: de Boer, F.S., Bonsangue, M.M., Madelaine, E. (eds.) Formal Methods for Components and Objects, 7th International Symposium, FMCO 2008, Sophia Antipolis, France, October 21-23, 2008, Revised Lectures. Lecture Notes in Computer Science, vol. 5751, pp. 278–297. Springer (2008). https://doi.org/10.1007/978-3-642-04167-9_14, https://doi.org/10.1007/978-3-642-04167-9_14
17. Cohen, E., Dahlweid, M., Hillebrand, M.A., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A Practical System for Verifying Concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5674, pp. 23–42. Springer (2009). https://doi.org/10.1007/978-3-642-03359-9_2, https://doi.org/10.1007/978-3-642-03359-9_2
18. Cohen, E., Moskal, M., Schulte, W., Tobies, S.: Local verification of global invariants in concurrent programs. In: Proceedings of CAV. pp. 480–494. Lecture Notes in Computer Science, Springer (2010)
19. Cok, D.R.: OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse. In: Dubois, C., Giannakopoulou, D., Méry, D. (eds.) Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, Grenoble, France, April 6, 2014. EPTCS, vol. 149, pp. 79–92 (2014). https://doi.org/10.4204/EPTCS.149.8, https://doi.org/10.4204/EPTCS.149.8
20. Cok, D.R.: Java automated deductive verification in practice: Lessons from industrial proof-based projects. In: Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV. Lecture Notes in Computer Science, vol. 11247, pp. 176–193. Springer (2018). https://doi.org/10.1007/978-3-030-03427-6_16, https://doi.org/10.1007/978-3-030-03427-6_16
21. Cok, D.R.: JML and OpenJML for Java 16. In: Cok, D.R. (ed.) FTfJP 2021: Proceedings of the 23rd ACM International Workshop on Formal Techniques for Java-like Programs, Virtual Event, Denmark, 13 July 2021. pp. 65–67. ACM (2021). https://doi.org/10.1145/3464971.3468417, https://doi.org/10.1145/3464971.3468417
22. Dross, C., Conchon, S., Kanig, J., Paskevich, A.: Reasoning with triggers. In: Lecture Notes in Computer Science. pp. 22–31. EPiC Series, EasyChair (2012)
23. Ekman, T., Hedin, G.: The JastAdd system – modular extensible compiler construction. Sci. Comput. Program. **69**(1-3), 14–26 (2007). https://doi.org/10.1016/j.scico.2007.02.003, https://doi.org/10.1016/j.scico.2007.02.003
24. Filliâtre, J., Paskevich, A.: Why3 - Where Programs Meet Provers. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice

of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7792, pp. 125–128. Springer (2013). https://doi.org/10.1007/978-3-642-37036-6_8, https://doi.org/10.1007/978-3-642-37036-6_8
25. Furia, C.A., Nordio, M., Polikarpova, N., Tschannen, J.: AutoProof: Auto-active functional verification of object-oriented programs. International Journal on Software Tools for Technology Transfer **19**(6), 697–716 (2016)
26. de Gouw, S., de Boer, F.S., Bubel, R., Hähnle, R., Rot, J., Steinhöfel, D.: Verifying OpenJDK's Sort Method for Generic Collections. J. Autom. Reason. **62**(1), 93–126 (2019). https://doi.org/10.1007/s10817-017-9426-4, https://doi.org/10.1007/s10817-017-9426-4
27. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn Verification Framework. In: Kroening, D., Pasareanu, C.S. (eds.) Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9206, pp. 343–361. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_20, https://doi.org/10.1007/978-3-319-21690-4_20
28. Hiep, H.A., Maathuis, O., Bian, J., de Boer, F.S., van Eekelen, M.C.J.D., de Gouw, S.: Verifying OpenJDK's LinkedList using KeY. CoRR **abs/1911.04195** (2019), http://arxiv.org/abs/1911.04195
29. Huang, B., Ray, S., Gupta, A., Fung, J.M., Malik, S.: Formal security verification of concurrent firmware in SoCs using instruction-level abstraction for hardware. In: Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA, June 24-29, 2018. pp. 91:1–91:6. ACM (2018). https://doi.org/10.1145/3195970.3196055, https://doi.org/10.1145/3195970.3196055
30. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In: Bobaru, M.G., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6617, pp. 41–55. Springer (2011). https://doi.org/10.1007/978-3-642-20398-5_4, https://doi.org/10.1007/978-3-642-20398-5_4
31. Kassios, I.T.: Dynamic frames: Support for framing, dependencies and sharing without restrictions. In: Proceedings of FM. pp. 268–283. Lecture Notes in Computer Science, Springer (2006)
32. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Aspects Comput. **27**(3), 573–609 (2015). https://doi.org/10.1007/s00165-014-0326-7, https://doi.org/10.1007/s00165-014-0326-7
33. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: 2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA. pp. 75–88. IEEE Computer Society (2004). https://doi.org/10.1109/CGO.2004.1281665, https://doi.org/10.1109/CGO.2004.1281665
34. Leavens, G.T., Schmitt, P.H., Yi, J.: The Java Modeling Language (JML) (NII shonan meeting 2013-3). NII Shonan Meet. Rep. **2013** (2013), https://shonan.nii.ac.jp/seminars/016/
35. Lehner, H., Müller, P.: Formal Translation of Bytecode into BoogiePL. Electron. Notes Theor. Comput. Sci. **190**(1), 35–50 (2007). https://doi.org/10.1016/j.entcs.2007.02.059, https://doi.org/10.1016/j.entcs.2007.02.059
36. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers. Lecture Notes in

Computer Science, vol. 6355, pp. 348–370. Springer (2010). https://doi.org/10.1007/978-3-642-17511-4_20, http://dx.doi.org/10.1007/978-3-642-17511-4_20
37. Leino, K.R.M., Moskal, M.: Usable auto-active verification. In: Usable Verification Workshop. http://fm.csl.sri.com/UV10/ (2010)
38. Leino, K.R.M., Müller, P.: Object invariants in dynamic contexts. In: Proceedings of ECOOP. pp. 491–516. Lecture Notes in Computer Science, Springer (2004)
39. Leino, K.R.M., Pit-Claudel, C.: Trigger selection strategies to stabilize program verifiers. In: CAV. pp. 361–381. Lecture Notes in Computer Science, Springer (2016)
40. Leino, K.R.M., Schulte, W.: Using history invariants to verify observers. In: Proceedings of ESOP. pp. 80–94. Lecture Notes in Computer Science, Springer (2007)
41. Marché, C., Paulin-Mohring, C., Urbain, X.: The KRAKATOA tool for certificationof JAVA/JAVACARD programs annotated in JML. J. Log. Algebraic Methods Program. **58**(1-2), 89–106 (2004). https://doi.org/10.1016/j.jlap.2003.07.006, https://doi.org/10.1016/j.jlap.2003.07.006
42. Meyer, B.: Introduction to the Theory of Programming Languages. Prentice Hall (1990)
43. de Moura, L.M., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24, https://doi.org/10.1007/978-3-540-78800-3_24
44. Müller, P., Nordio, M.: Proof-transforming compilation of programs with abrupt termination. In: Proceedings of SAVCBS. pp. 39–46. ACM (2007), https://doi.org/10.1145/1292316.1292321
45. Necula, G.C.: Proof-carrying code. In: Lee, P., Henglein, F., Jones, N.D. (eds.) POPL. pp. 106–119. ACM Press (1997). https://doi.org/10.1145/263699.263712
46. Paganoni, M., Furia, C.A.: Byteback fm 2023 replication package (Nov 2022). https://doi.org/10.5281/zenodo.7337205, https://doi.org/10.5281/zenodo.7337205
47. Pearce, D.J., Utting, M., Groves, L.: Verifying Whiley programs with Boogie. J. Autom Reasoning (2022). https://doi.org/https://doi.org/10.1007/s10817-022-09619-1
48. Polikarpova, N., Tschannen, J., Furia, C.A., Meyer, B.: Flexible invariants through semantic collaboration. In: Proceedings of FM. Lecture Notes in Computer Science, vol. 8442, pp. 514–530. Springer (2014)
49. Rakamaric, Z., Emmi, M.: SMACK: Decoupling Source Language Details from Verifier Implementations. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8559, pp. 106–113. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_7, https://doi.org/10.1007/978-3-319-08867-9_7
50. Rümmer, P.: JayHorn: a Java model checker. In: Murray, T., Ernst, G. (eds.) Proceedings of the 21st Workshop on Formal Techniques for Java-like Programs, FTfJP@ECOOP 2019, London, United Kingdom, July 15, 2019. p. 1:1. ACM (2019). https://doi.org/10.1145/3340672.3341113, https://doi.org/10.1145/3340672.3341113
51. Summers, A.J., Drossopoulou, S., Müller, P.: The need for flexible object invariants. In: Proceedings of IWACO. pp. 1–9. ACM (2009)
52. Sung, C., Paulsen, B., Wang, C.: CANAL: A Cache Timing Analysis Framework via LLVM Transformation. CoRR **abs/1807.03329** (2018), http://arxiv.org/abs/1807.03329
53. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L.J., Lam, P., Sundaresan, V.: Soot - a java bytecode optimization framework. In: MacKay, S.A., Johnson, J.H. (eds.) Proceedings of the

1999 conference of the Centre for Advanced Studies on Collaborative Research, November 8-11, 1999, Mississauga, Ontario, Canada. p. 13. IBM (1999), https://dl.acm.org/citation.cfm?id=782008
54. Zhang, Y., Zuck, L.D.: Formal Verification of Optimizing Compilers. In: Negi, A., Bhatnagar, R., Parida, L. (eds.) Distributed Computing and Internet Technology - 14th International Conference, ICDCIT 2018, Bhubaneswar, India, January 11-13, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10722, pp. 50–65. Springer (2018). https://doi.org/10.1007/978-3-319-72344-0_3, https://doi.org/10.1007/978-3-319-72344-0_3