ESBMC-Jimple: Verifying Kotlin Programs via Jimple Intermediate Representation

Rafael Menezes∗
University of Manchester
United Kingdom

Daniel Moura
Federal University of Amazonas
Brazil

Helena Cavalcante
Federal University of Amazonas
Brazil

Rosiane de Freitas
Federal University of Amazonas
Brazil

Lucas C. Cordeiro†
University of Manchester
United Kingdom

ABSTRACT
We describe and evaluate the first model checker for verifying Kotlin programs through the Jimple intermediate representation. The verifier, named ESBMC-Jimple, is built on top of the Efficient SMT-based Context-Bounded Model Checker (ESBMC). It uses the Soot framework to obtain the Jimple IR, representing a simplified version of the Kotlin source code, containing a maximum of three operands per instruction. ESBMC-Jimple processes Kotlin source code together with a model of the standard Kotlin libraries and checks a set of safety properties. Experimental results show that ESBMC-Jimple can correctly verify a set of Kotlin benchmarks from the literature; it is competitive with state-of-the-art Java bytecode verifiers. A demonstration is available at https://youtu.be/J6WhNfXvJNc.

CCS CONCEPTS
• Theory of computation → Verification by model checking
• Software and its engineering → Formal software verification

KEYWORDS
Formal Verification, Software Model Checking, Kotlin, Jimple.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:
Rafael Menezes, Daniel Moura, Helena Cavalcante, Rosiane de Freitas, and Lucas C. Cordeiro. 2022. ESBMC-Jimple: Verifying Kotlin Programs via Jimple Intermediate Representation. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA ’22), July 18–22, 2022, Virtual, South Korea. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3533767.3543294

1 INTRODUCTION
Kotlin is a multiplatform programming language [6] to provide a more productive way to develop software on top of the JVM [7]. Kotlin adds functional programming features and safety checks; it

† Also with Federal University of Amazonas.
∗ Also with Federal University of Amazonas.

has full interoperability with JVM [6]. Google added support for Android Development using Kotlin, becoming one of the popular choices among software developers [11]. However, as the Kotlin programming language’s popularity grows, there is a clear need for more associated verification tools to ensure its safety.

Bounded Model Checking (BMC) is a formal verification technique that can check implementation errors in software [4]. The basic idea of the BMC technique is to check (the negation of) a given property at a given depth. A significant strength of BMC is that it analyzes only bounded program runs, thereby achieving decidability. A notable example of a tool that implements BMC is ESBMC [9], which is based on Satisfiability Modulo Theories (SMT) solvers and checks safety properties such as arithmetic overflow, array bounds, pointer safety, and user-specified properties. ESBMC uses its intermediate representation (IR) GOTO to analyze programs, converting loop structures into the GOTO form and supporting object-oriented programming (OOP) features.

We rely on the Jimple IR, a stackless 3-address IR with a set of instructions from the Soot framework [12] to simplify the bytecode analysis. As a result, ESBMC does not track a virtual table, and neither maintains a stack [2], making the verification process more straightforward. Since ESBMC verifies C/C++ [9, 10] programs, a similar approach can be used when dealing with Jimple, as the symbolic execution engine has the support for OOP features. Some verifiers support Java bytecode verification (e.g., JBMC [5] and JayHorn [5]). However, they have shortcomings when verifying Kotlin because of limitations on absent models and assertions, e.g., array initialization on Kotlin relies on its SDK for the implementation.

We describe the first model checker for Kotlin programs, ESBMC-Jimple, which adds Jimple support to ESBMC to verify safety properties in Kotlin programs. Therefore, we can use ESBMC’s verification strategies (e.g., incremental BMC and $k$-induction) and optimizations (e.g., program slicing) to enable the verification of Kotlin programs that otherwise would exhaust the systems resources. To evaluate ESBMC-Jimple, we developed a new set of benchmarks for Kotlin analysis extracted from a GitHub repository [1] and also by translating a subset of the SV-COMP benchmarks from Java safety into Kotlin [3]. Our experimental evaluation has shown that ESBMC-Jimple could correctly verify more Kotlin benchmarks than JBMC [5], a native Java bytecode verifier.
Figure 1: Architectural overview of ESBMC with its extension (Jimple Frontend) for verifying Kotlin programs.

2 TOOL DESCRIPTION

2.1 Architecture and Implementation

Fig. 1 illustrates the ESBMC-Jimple architecture. The flow starts from a Kotlin file, compiled to Jimple (through Soot), and then symbolically executed to produce a logical formula.

2.2 Jimple Conversion Methodology

To use the ESBMC symbolic execution, we convert the original Jimple IR into GOTO, which is the IR used by ESBMC. It contains instructions for function calls, assignments, declarations, exception handling, and thread creation. For the Jimple IR support, a subset of this language was used (Table 1). Further instructions are not used yet (e.g., try/catch blocks), and some instructions are used internally for analysis (e.g., assume and assert statements). Listing 1 shows how a GOTO program is represented through named Functions, which are implemented using Statements and Expressions.

Listing 1: GOTO grammar subset.

| Statement | Description |
|-----------|-------------|
| ASSIGN(a, b) | Assigns value of b into symbol a. |
| DECL(a, t) | Create a new valid memory object for a of type t. |
| DEAD(a) | Deletes the memory object, making it an invalid object. |
| FUNCTION_CALL(n, ...) | Calls function n with parameters. |
| LABEL(L) | Sets a label L in the code, this label can be used for jumps. |
| GOTO(L) | Goto a specific label L. |
| IF(e, L) | Checks if a condition e is true, and if it is goto label L. |
| RETURN(e?) | Return to the latest FUNCTION_CALL with the optional result e. |
| THROW(e) | Throws an exception with expression e. |

The GOTO language uses Expressions. Those expressions range from binary operations (e.g., add and subtraction) to dereferencing operations (e.g., accessing a member of the struct). Since there is a C frontend, the most needed expressions are implemented. The exceptions are the cmp operations that exist for Java-based languages, implemented during the GOTO conversion as an expression (Expr).

2.2.1 Objects and Classes. Jimple contains OOP elements; every file is defined via class members, which include fields and functions that can be virtual or static. These members can be overridden during inheritance; functions with the same name can have multiple specializations by changing their parameters. To model classes, we rely on a similar approach to verify C++ programs [10]. We map the virtual fields into structs and the static fields into global variables (within a scoped namespace). However, the GOTO language does not support polymorphism and neither for function inside a structure. To solve this, we used the following approach. (1) Generate an identifier for each function using its name and parameter types. A function foo with return type int and parameter type of double is renamed to foo_int_double. (2) Due to how Jimple handles functions parameters, we convert each parameter as a global variable; at the start of each function this value is then read. This operation is marked as atomic for the GOTO flow. (3) For virtual functions, we add an extra argument into the function: a pointer of its own class. A virtual function bar() inside the class A would become bar(A*). This parameter behave as the this pointer of the object.

This approach simplifies the inheritance process as the inherited classes have the same fields that can be used for access. Jimple IR handles the virtual table, stating from which class of the inheritance tree the method should be called. For the object memory model, we use the one for C++ programs [9]. In this model, primitive types are allocated into the stack; constructed objects (i.e., new or newarray) use the heap concept. We do not model a garbage collector; we assume that it will be valid for the entire verification flow once something is allocated into the heap.

2.2.2 Jimple Conversion. The Jimple conversion begins with generating types for the classes; every class is converted into a struct containing all its virtual fields. Each Jimple statement is converted into an equivalent statement in GOTO for functions. Table 2 contains the translations used for every Jimple statement; it makes assumptions of previously conversions performed: DeclarationLists (e.g., int a, b, c;) were converted into a multiple individual declarations, at identifiers (e.g., @this, @param1) were injected properly.
2.2.3 Illustrative Example. We use the example in Listing 2 to show how the conversion process works. There exists a class named Foo, which contains a mutable field member and the increment method, which increments the member field. After the Jimple file is generated via Soot [12], ESBMC can verify it by invoking:

```
esbmc <file>.jimple --k-induction --overflow-check
```

where <file>.jimple is the Jimple file to verify, --overflow-check enables checking for signed integer overflows, and --k-induction sets the k-induction as the proof rule. The full list of options can be seen by using the flag --help.

Listing 2: Example of a Kotlin program that defines the Class Foo used to illustrate the GOTO conversion.

```
class Foo(var member: Int) {
  fun increment(N: Int): Int {
    member = member + N
    return member
  }
}
```

Listing 3 contains the GOTO version of the increment function in Listing 2. Here, we see the function name (as described in Subsection 2.2.1). Next, Jimple adds intermediate variables (e.g., r0, i0, i1). The “at” identifiers, i.e., @this and @parameter0 indicate the object pointer to itself and the first function parameter, respectively.

```
increment_2 (Foo:increment_2):
  Foo • r0;
  signed int i0, $i1, $i2, $i3;
  r0 = (Foo • @this; $i0 = @parameter0;
  $i1 = r0 → member;
  $i2 = $i1 + i0; r0 → member=$i2;
  $i3 = r0 → member; RETURN: $i3
```

ESBMC will unroll the program during the symbolic execution, converting it to an SSA form. In Listing 2, we can check the increment function; we make its arguments non-deterministic.

The verification conditions are then encoded in the form of \( C \land \neg P \), as shown in Listing 4:

```
Listing 4: Verification condition.
C := @this = nondet ()
∧ r0 = @this ∧ @parameter0 = nondet ()
∧ i0 = @parameter0 ∧ $i1 = r0 → member
P := overflow ("+", $i0, $i1)
```

\( P \) contains the assignments; both function arguments, i.e., @this and @parameter0 are set as nondeterministic. \( P \) is the property that adding \( i0 \) with \( $i1 \) cannot lead to an overflow.

3 EXPERIMENTAL EVALUATION

We evaluated ESBMC-Jimple using a benchmark suite and compared its performance against JBMC [5] since it supports Java Bytecode verification efficiently. Other tools were considered, Jayhorn [8] and Joogie [2]. However, we could not use Jayhorn and neither Joogie for our benchmarks. Tools, benchmarks, and evaluation results are available on a supplementary web page.1

Description of Benchmarks. We developed a small suite of benchmarks (see Table 3) for evaluating ESBMC-Jimple. These benchmarks contain execution paths that can trigger bugs in Kotlin applications and execution paths without property violations. They also include the following properties: reachability, overflows, and null-pointer exception. The benchmarks contain non-deterministic behavior, modeled through the Java Random function.

3.1 Objective and Setup

Our main experimental questions are:

- **EQ1** (soundness) Can the tool prove (partial) correctness?
- **EQ2** (performance) How long does ESBMC-Jimple take to verify a Kotlin application?
- **EQ3** (completeness) Can the tool correctly identify bugs in Kotlin programs?

We set up the experiments on a Ubuntu 20.04 with 160 GB of RAM running a 25-core Intel KVM CPU. If the tool could produce a counterexample, it is manually tested on the benchmark. Importantly, all presented execution times are CPU times, i.e., only the elapsed periods spent in the allocated CPUs, measured with the times system call (POSIX system). Since neither ESBMC-Jimple nor JBMC supports Kotlin input directly, we first compile the Kotlin source file into Java Bytecode and then convert it back to Jimple. Then, the Java Bytecode is given as input to JBMC; the Jimple is given for the ESBMC-Jimple.

3.2 Results

Table 3 shows the results; ESBMC-Jimple could verify all benchmarks originating from Kotlin programs correctly. Note that JBMC was not developed for verifying Kotlin programs, which thus did not handle some constructs correctly. For example, regarding the overflow benchmarks (TC0 and TC1), ESBMC-Jimple could detect the overflow in both cases; nondeterministic values were used for the addition of positive numbers (i.e., \( X + Y ≥ 0, VX ≥ 0, Y' ≥ 0 \)).

1https://doi.org/10.5281/zenodo.6514235
ESBMC-Jimple could produce a counterexample that led to the overflow. For TC2 and TC3, we use the same approach, but for negative numbers (i.e., $X + Y \leq 0, VX \leq 0, Y \leq 0$), ESBMC-Jimple could also produce counterexamples. However, JBMC was unable to identify any violations in TC0-TC3.

Division by zero benchmarks (TC4 and TC5) ensure that the denominator of a division could never be zero. ESBMC-Jimple and JBMC could correctly identify the flaws and produce the respective counterexamples. Similarly, for array bounds check benchmarks (TC6-TC9), ESBMC-Jimple and JBMC had the same outcome; they could detect bounds violations for nondeterministic arrays originating from Kotlin programs. We also evaluated benchmarks with user assertion violations (T10-TC15) and without violation (TC16-TC29). ESBMC-Jimple could verify these benchmarks correctly, but JBMC refuted the assertions, incorrectly triggering a violation for all benchmarks stated as safe.

Both EQ1 and EQ3 can be confirmed through our experiments, as ESBMC-Jimple could reason over the safety of Kotlin programs and generate valid counterexamples that trigger the property violation. Regarding performance (EQ2), due to ESBMC’s efficient support for bug finding and proof strategies (e.g., incremental BMC, k-induction), ESBMC-Jimple could quickly refute safety properties or prove (partial) correctness of Kotlin programs.

Table 3: Experimental results, where column “Found” indicating whether a counterexample was provided. The bottom lines are the results summary.

| Benchmark   | JBMC | ESBMC-Jimple |
|-------------|------|--------------|
| IDs         | Property | Found | CE | Found | CE |
| TC0-1       | Overflow | No     | No | Yes   | Yes |
| TC2-3       | Underflow | No     | No | Yes   | Yes |
| TC4-5       | Div-by-zero | Yes    | Yes | Yes   | Yes |
| TC6-9       | Out-of-bounds | Yes    | Yes | Yes   | Yes |
| TC10-15     | Assertion Fail | Yes    | No  | Yes   | Yes |
| TC16-20     | No violation | Yes   | N/A | No    | N/A |
| Correct Results |       | 57%    | 100% |
| Confirmed Results |       | 38%    | 100% |
| Total CPU Time |      | 3.563s | 18.878s |

## 4 RELATED WORK

Some tools also leverage Java verification using bytecode for the analysis, mainly JBMC [5] and JayHorn [8]. JBMC has an architecture similar to ESBMC-Jimple, having to translate Java bytecode into a GOTO program. JayHorn is a software verifier for Java bytecode that generates a set of constrained Horn clauses. The architecture of JayHorn contains the Soot framework as its front-end, with transformation validated with Randoop. It then uses a CHC solver to check the input program safety. Unfortunately, both JBMC and JayHorn have shortcomings (as seen in our experiments) when verifying Kotlin code. Another similar work is Joogie [2], which translates jimple code into the Boogie language. Boogie is a description language that can be translated into SMT formulas to be checked by an SMT solver. Joogie focuses on verifying Java applications and could verify real-world Java programs successfully.

## 5 CONCLUSIONS AND FUTURE WORK

We presented and evaluated ESBMC-Jimple, the first software model checker for Kotlin applications, which relies on the Jimple IR from the compiled Kotlin program and the ESBMC verification engine. ESBMC-Jimple can handle various features from Kotlin, including classes, inheritance, and polymorphism. Furthermore, ESBMC-Jimple outperformed another state-of-the-art Java bytecode verifier since it could detect and generate more real counterexamples for the evaluated benchmarks. We will support exception handling, more operational models for the Kotlin standard, and extend the range of properties for future work.

### ACKNOWLEDGMENTS

This research was partially sponsored by Motorola Mobility Comércio de Produtos Eletrônicos Ltda and Flextronics da Amazônia Ltda, according to Federal Law n° 8.387/1991, through agreement n° 004/2021, signed with ICOMP/FUAM.

### REFERENCES

[1] The Algorithms. 2022. TheAlgorithms/Kotlin (Github repository). https://github.com/TheAlgorithms/Kotlin
[2] Stephan Arlt, Philipp Rümmer, and Martin Schäf. 2013. Joogie: From java through jimple to boogie. In Proceedings of the 2nd ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis. 3–8.
[3] Dirk Beyer. 2022. Progress on software verification: SV-COMP 2022. In TACAS Vol. 15244. Springer Nature, 375.
[4] Edmund M Clarke, Thomas A Hennziger, Helmut Veith, Roderick Bloem, et al. 2018. Handbook of model checking. Vol. 10. Springer.
[5] Lucas Cordeiro, Pascal Kesseli, Daniel Kroening, Peter Schrammel, and Marek Trtik. 2018. JBMC: A bounded model checking tool for verifying Java bytecode. In CAV. 183–190.
[6] Kotlin Foundation. 2022. Kotlin Language. https://kotlinlang.org/
[7] Bruno Góis Mateus and Matias Martinez. 2019. An empirical study on quality of Android applications written in Kotlin language. Empirical Software Engineering 24, 6 (2019), 3356–3393.
[8] Temesghen Kahsai, Philipp Rümmer, and Martin Schäf. 2019. JayHorn: A Java Model Checker - (Competition Contribution). In TACAS (LNCS, Vol. 11429). Springer, 214–218.
[9] B. Fischer L. Cordeiro and J. MARQUES-SILVA. 2012. SMT-Based Bounded Model Checking for Embedded ANSI-C Software. IEEE Transactions on Software Engineering (TSE) 38 (2012), 957–974.
[10] Felipe R Monteiro, Mikhail R Gadela, and Lucas C Cordeiro. 2022. Model checking C++ programs. STVR 32, 1 (2022), e1793.
[11] Victor Oliveira, Leopoldo Teixeira, and Felipe Ebert. 2020. On the Adoption of Kotlin on Android Development: A Triangulation Study. In SANER. 206–216.
[12] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In CASCON First Decade High Impact Papers. 214–224.