SPouT: Symbolic Path Recording during Testing - a Concolic Executor for the JVM *

Malte Mues (✉)[1], Falk Howar[1,2], and Simon Dierl[1]

[1] TU Dortmund University, Dortmund, Germany
[2] Fraunhofer ISST, Dortmund, Germany
{malte.mues, falk.howar, simon.dierl}@tu-dortmund.de

Abstract. In this paper, we present SPouT, a concolic executor for the `Java` virtual machine. To the user, SPouT is a `java` executable that takes some additional parameters for setting the values of concolic inputs and produces symbolic traces over variables under observation during the execution. Technically, SPouT extends the JVM implementation provided by the Espresso guest language for the GraalVM. Therefore, SPouT is the first concolic executor build on an industrial JVM. In this paper, we describe the architectural design of SPouT, detail how the partial symbolic analysis of `Java`'s strings is implemented in SPouT, and show its performance and versatility by comparing it to other analysis tools for `Java` programs.

1 Introduction

Symbolic analysis of `Java` applications at scale is a tough technical challenge. The `Java` platform has many semantically rich features (reflection, lambda expressions, annotations, etc.) and the Java Virtual Machine is a complex execution environment. Tools for the symbolic analysis of `Java` programs broadly fall into three categories: tools that translate `Java` code or some intermediate representations, e.g., bytecode, into a representation amenable to formal analysis (examples are KeY [1], JayHorn [7], and JBMC [6]), tools that instrument bytecode and execute it on an unmodified Java Virtual Machine (COASTAL is one recent example), and tools that do not modify the programs, but instrument a Java Virtual Machine to analyze bytecode during execution. Java PathFinder [24] was the first successful model checker for `Java` and its analyzer JPF-VM, a `Java`-based JVM implementation running on top of a normal JVM, served as a basis for many tools in this third category (e.g., SPF [17], Java Ranger [21], and JDart [12]).

* This work has been partially funded by an Amazon Research Award. This version of the contribution has been accepted for publication, after peer review (when applicable) but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: https://doi.org/10.1007/978-3-031-17108-6_6. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms

For all three categories, the soundness of analyses hinges on correctly modeling (or not changing) the semantics of the `Java` language or the specified behavior of a JVM and all three approaches have distinct advantages and drawbacks: Working on `Java` code directly removes a big portion of complex technical machinery from the analysis but has to be adapted to new language features and manually transferred to other languages. Tools that instrument bytecode instructions are light-weight with respect to their execution environment (a standard JVM) but for a complete symbolic analysis, the symbolic semantics of stack and heap operations have to be woven into the analyzed code. Instrumenting a Java Virtual Machine enables analysis of all programs and language features that are compiled to JVM bytecode but requires a JVM implementation to instrument. Developing and maintaining a sufficiently feature-complete JVM that powers the analysis is cumbersome: e.g., as of today, the JPF-VM only supports the analysis of `Java` version 8 bytecode, while `Java` 11 has been the LTS version since 2018, and was succeeded by Java 17 (LTS) in September of 2021.

In this paper, we present SPouT, a concolic executor for the `Java` Virtual Machine. To the user, SPouT is a `java` executable that takes some additional parameters for setting the values of concolic inputs and produces symbolic traces over variables under observation during the execution. Technically, SPouT instruments the Espresso guest language for the GraalVM [25]. Espresso is an existing full-fledged virtual machine written in `Java` and maintained by Oracle. We describe the architectural design of SPouT and detail how the partial symbolic analysis of `Java`'s standard string library is implemented in SPouT.

We evaluate the performance of SPouT and its versatility by comparing it to other analysis tools for `Java` programs in a series of usage examples and by analyzing the results of SV-COMP 2022, where we used SPouT as a component in the GDart [15] tool ensemble for the dynamic symbolic execution of `Java` bytecode. Our results show that the architectural design of SPouT is a viable alternative to the design of existing research tools. Further, the described experiments demonstrate that SPouT analyzes programs that the JPF-VM cannot execute. This enables further research on dynamic symbolic execution for `Java`.

Related Work. There are two areas of research influencing the design of SPouT: The symbolic encoding of `Java` programs during the analysis and the encoding of string specific operations in general. In most cases, converting a program to its analysis representation is only one step implemented as part of an analyzer. As we are only interested in the symbolic encoding approach in the context of SPouT, we shortly describe some selected techniques for preparing the analysis target in `Java` tools participating at SV-COMP without discussing the analysis approaches in detail. The encoding of string operations in the symbolic analysis is also a very active research area, and we cannot discuss all results here. Instead, we present selected examples with a strong influence on this paper.

All presented techniques generate a logical representation of the `Java` program. However, they vary in the concrete style of encoding and the involved abstraction level. JayHorn [7] compiles the `Java` program into horn clauses and operates on the `Java` byte code. The analysis is completely executed on the sym-

bolic encoding. JBMC converts `Java` code in to CBMC's [5] goto-language and continues the analysis on this intermediate format. SPF [17] was the first symbolic execution engine build on top of JPF and encodes the JVM bytecode under analysis into a symbolic representation using the JPF-VM. Java Ranger [21] reuses this infrastructure for encoding the program. COASTAL's[3] concolic executor uses the ASM[4] bytecode manipulation framework to weave the symbolic constraints recording code into the program under analysis. The tools conceptually closest to SPouT are JDart [12] and SymJEx [8]. JDart instruments the JPF-VM to record symbolic constraints during concrete execution, but behaves mostly identical to SPouT except for the trace reporting. SymJEx uses the GraalVM compiler frontend to lift `Java` bytecode into an intermediate representation for performing symbolic execution. In contrast SPouT executes bytecode using the Espresso VM running on the GraalVM. As this paper focuses mainly on dynamic testing methods for `Java`, we skip most literature dealing with static testing methods for `Java` (e.g. Julia [22] or PMD[5]).

Previous work on encoding string operations for the analysis of `Java` generally uses bitvector encodings (e.g. [3, 18]) or automata based encodings built from collected string graphs (e.g. [4, 18, 20]). Instead, SPouT encodes the string operations using SMT-Lib's theory of strings [2] as an abstraction and leaves the decision making to the SMT solver working with SPouT's output. This cleanly decouples the constraint generation (as part of concolic execution) and solving.

The Java String Analyzer (JSA) by Christensen et al. [4] is one of the first major static string analyzer for `Java` build using the automata theory stimulating many follow up work. Redelinghuys et al. [18] used it for deciding string graphs, their intermediate representation of string operations in `Java` programs. They compare the automata theory with a bitvector encoded version decided by Z3 [13]. Their main result is that there is no significant difference between using automatons or bitvectors for representing strings symbolically. Instead, the combination of constraints limiting the string content and its length influences performance. SMT-Lib allows to express these constraints in SPouT's encoding and from our experience in previous work [14], today's string solvers support them well. Bjørner et al. [3] describe the integration of a bitvector based encoding for string operations in the dynamic symbolic execution of .NET programs with PEX [23]. They split the check into numeric checks first and the content constraint next, allowing the precise modelling of exceptions caused by wrong relations between indices and the string length, e.g., if the index is outside of the string size for the `charAt` method. SPouT follows this encoding pattern, but instead of encoding the string content parts as bitvectors, we encode it in the SMT-Lib string theory. Shannon et al. [19] describe different techniques for instrumenting the analysis enabling to intercept the operations on string values on the `Java` standard library API level and evaluate replacing the `String` class with a `SymbolicString` class for interception the calls. SPouT directly instruments

[3] https://github.com/DeepseaPlatform/coastal
[4] https://asm.ow2.io
[5] https://pmd.github.io

$$
\begin{aligned}
\mathit{trace} &::= (\mathit{decl} \mid \mathit{decision} \mid \mathit{err} \mid \mathit{abort} \mid \mathit{assume})^* \quad [\texttt{ENDOFTRACE}] \\
\mathit{decl} &::= [\texttt{DECLARE}] \quad \mathit{def} \\
\mathit{decision} &::= [\texttt{DECISION}] \quad \mathit{expr} \quad // \quad \texttt{branchCount} = i, \texttt{branchId} = j \\
\mathit{err} &::= [\texttt{ERROR}] \quad \mathit{cause} \\
\mathit{abort} &::= [\texttt{ABORT}] \quad \mathit{cause} \\
\mathit{assume} &::= [\texttt{ASSUMPTION}] \quad \mathit{expr} \quad // \quad \texttt{sat} = b
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{def} \in \text{SMTLib Fun. Defs.} \qquad i, j \in \mathbb{N}_0 \\
&\mathit{expr} \in \text{SMTLib Assertions} \qquad b \in \{\mathit{true},\ \mathit{false}\} \qquad \mathit{cause} \in \text{Text}
\end{aligned}
$$

Fig. 1: The BNF grammar for a trace that summarizes a concolic execution run.

the `String` class inside the JVM instead of replacing it in the implementation. As part of their experiments, Shannon et al. pointed out the problem of domain crossing whenever strings are converted to numbers. Their encoding does not express the semantic implications precisely. We still have the same problems today with SPouT.

Outline. Section 2 describes the internals of SPouT. In Section 3, we provide usage examples and evaluate the performance of SPouT by comparing it to several other analysis tools for the JVM.

2 SPouT: Directing the flow of Espresso

SPouT[6] (*Symbolic Path Recording during Testing*) implements concolic execution in the Espresso[7] Java Virtual Machine for Oracle's GraalVM[8]. As it extends Espresso, SPouT – like Espresso – is licensed under the GPLv2. SPouT's symbolic additions are bundled in a single `Concolic` class. This avoids scattering the functionality across the virtual machine. SPouT extends *Espresso* with the functionality to (a) maintain symbolic annotations for values on the stack and on the heap, (b) compute the effect of bytecode instructions on these symbolic annotations, (c) record path constraints on branching points, and (d) inject concolic values into the analysis. Using substitution methods as a general extension method of the GraalVM, we demonstrate how to intercept the invocation of standard library methods to encode them symbolically on higher levels than the executed bytecode instructions. SPouT uses this technique for encoding operations of the `Java` string library.

[6] https://github.com/tudo-aqua/spout, available under GPLv2
[7] https://github.com/oracle/graal/tree/master/espresso
[8] https://github.com/oracle/graal/

2.1 SPouT's Design

SPouT is built as a `java` executable. Command-line arguments added to the invocation of SPouT allow seeding the concolic variables. SPouT will report back the collected symbolic constraints and the result using the trace language defined by the BNF grammar in Figure 1. Trace logs can contain symbolic variable declarations, assumptions and decisions (i.e., SMT-Lib assertions over symbolic variables), as well as errors and abort statements. For decisions and assertions minimal information about the *shape* of a trace, i.e., branch counts and branch directions are communicated, furthering the easy integration of SPouT as a component in other analyses. SPouT uses a `Verifier` class with nondeterministic value factories for all supported concolic data types, e.g., `int`, `boolean`, `String`, etc., to allow programmatic definition of concolic variables in the driver method of an analysis.

We demonstrate the usage of SPouT by analyzing the small Kotlin[9] program shown in Listing 2 (Kotlin is compiled to JVM bytecode). The program generates an integer value, using the `Verifier.nondetInt()` method that is instrumented by SPouT during concolic execution, i.e., SPouT will return the configured concrete value, create a symbolic variable for the return value and track its influence through path constraints. In the program, the returned concolic integer is used in the test `(x > 0)`, guarding an assertion violation.

```
import tools.aqua.concolic.Verifier
fun main() {
    val x = Verifier.nondetInt()
    if (x > 0)
        assert(false)
}
```

Listing 2: A simple Kotlin program with a guarded assertion violation and a call to method `Verifier.nondetInt()` that returns concolic values.

The `kotlin` command basically wraps a `java` invocation and adds Kotlin resources to the classpath. We execute the compiled Kotlin program concolically by using the GraalVM with SPouT as follows:

```
export JAVA_HOME=/path/to/spoutvm
/path/to/kotlin/bin/kotlin -J"-truffle" -J"-ea" \
  -Dconcolic.ints=0 \
  -cp [verifier-stub]:. MainKt
```

The argument `-J"-truffle"` is passed to the GraalVM `Java` binary and selects the Truffle-based Java implementation, while `-J"-ea"` enables assertion checking. `-Dconcolic.ints=0` is parsed by SPouT and instructs it to use 0 as the concrete value for the first concolic integer value. The argument takes a comma-separated list of values that seed the `Verifier` class and controls this way the execution along a specific path. Once the preseeded values are exhausted, default values are used. Similar arguments exist for all primitive data types and string values. For the sake of brevity, we omit parts of the executions output and

[9] https://kotlinlang.org/

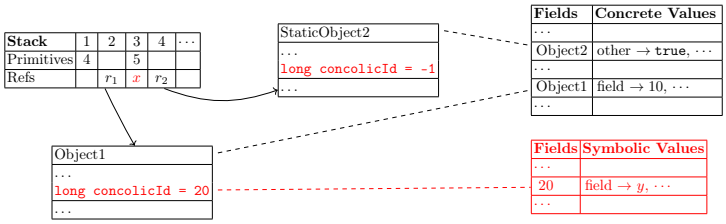

Fig. 2: Memory architecture of SPouT. Black parts provided by Espresso. Red parts added by SPouT.

only show the recorded trace with abbreviated decision branch information:

[DECLARE] (declare-fun `__int_0` () (_ BitVec 32))
[DECISION] (assert (bvsle `__int_0` #x00000000)) //b.Count = 2, b.Id = 1
[ENDOFTRACE]

In the trace, the concolic integer value is represented by the variable `__int_0`.

2.2 Memory Architecture

The JVM state is represented in the memory using the stack and the heap. In the following, we describe how these memory structures are extended with symbolic annotations, the concolic bytecodes propagate the symbolic annotations and record symbolic constraints, and substitution methods enable the symbolic encoding of methods.

Symbolic Stack and Heap. As sketched in Figure 2, Espresso uses two arrays of identical size to represent the stack; one for primitive values and one for object references. They are populated alternatingly (for each index, either a primitive or an object reference is stored). SPouT leverages this layout and stores concolic information about primitive values at the (unused) corresponding index in the object reference array and vice versa. As a consequence, Espresso takes care of all stack operations (e.g., copying values and annotations between frames). On the heap, Espresso represents every instantiated object by a `StaticObject` that – among other things – maintains a field table and stores contents of field in an optimized (native) location. SPouT extends these container objects with unique concolic ids and stores symbolic contents of fields in a map indexed by id. This mechanism operates lazily, creating map entries only for objects with concolically tainted fields, keeping the memory overhead of this analysis minimal. SPouT extends the `getfield` and `setfield` instructions to propagate symbolic annotations between stack and heap.

Concolic Bytecodes. Concolic bytecode implementations are used for computing symbolic effects of instructions and for recording path constraints (e.g., on branching instructions). SPouT extends the implementation of all bytecodes that compute values, e.g., `iadd`, or introduce branching conditions, e.g., `if_icmpne`. Listing 3 shows the concolic extension of the `iadd` instruction: After the concrete effect on the stack is computed by Espresso, SPouT computes the

```
void iadd(long[] primitives, Object[] refs, int top) {
    int i1 = popInt(primitives, top - 1);
    int i2 = popInt(primitives, top - 2);
    int iRes = i1 + i2;
    putInt(primitives, top - 2, iRes);
    // added concolic operation
    putConcolic(refs, top -2, sadd(i1, i2, iRes,
        popConcolic(refs, top-1), popConcolic(refs, top-2)));
}

Concolic sadd(int i1, int i2, int iRes, Concolic c1, Concolic c2) {
    if (c1 == null && c2 == null) return null;
    if (c1 == null) c1 = concFromConstant(i1);
    if (c2 == null) c2 = concFromConstant(i2);
    return new Conc(iRes, new Expression(IADD, c1.symb(), c2.symb()));
}
```

Listing 3: Implementation of Concolic `iadd` Bytecode.

symbolic effect only if needed, keeping the impact on performance as small as possible.

Substituting Methods. Espresso provides a mechanism for substituting individual methods with customized versions (i.e., executing custom code instead of the actual method). SPouT leverages this mechanism for two purposes: (a) user-defined concolic values are injected via methods of the `Verifier` class[10], and (b) the concolic semantics of operations on strings are implemented in substituted implementations. A concolic semantic consists of two parts: the concrete computation that changes the heap state and the update of the symbolic values.

2.3 Symbolic Encoding of String Operations

The analysis of string operations is especially important for reasoning on `Java` programs [4, 10, 18] and encoding them on the string level has various benefits over instrumenting the primitive types that represent a string (c.f. [3, 19]). In the following, we describe how SPouT encodes string operations, present three concrete challenges in the encoding, and describe the limitations. The tree challenges are: faithful error handling, numeric and string semantic of characters, and regular expressions.

Encoding. SPouT encodes `Java`'s string operations symbolically using a mixture of integer and string theory constraints in the SMT-Lib language. While for some operations in `Java`, the SMT-Lib language contains matching counterparts (shown in Table 1), others exhibit different semantics in edge cases or are not expressible (shown in Table 2). String operations are the first datatype

[10] https://github.com/tudo-aqua/verifier-stubs

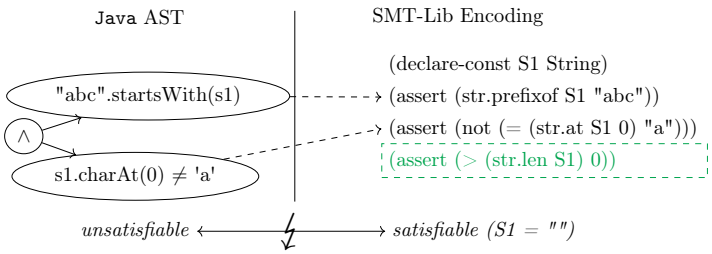

Fig. 3: The black boxes on the right demonstrate the naive mapping from the `Java` AST to an SMT-Lib encoding. The bottom part shows the semantic mismatch: the test is unsatisfiable in the `Java` language while the SMT-Lib semantics admit an empty string as a valid model for S1.

Table 1: Mapping from the `Java` standard string library to SMT-Lib.

Java Operation	SMT-Lib Operation	Java Operation	SMT-Lib Operation
concat(s1, s2)	(str.++ *s1 s2*)	isDigit(c1)	(*str.is_digit c1*)
contains(s1, s2)	(*str.contains s1 s2*)	length(s1)	(*str.len s1*)
contentEquals(s1, seq2)	(= *s1 seq2*)	replace(s1, s2, s3)	(*str.replace_all s1 s2 s3*)
endsWith(s1, s2)	(*str.suffixof s1 s2*)	startsWith(s1, s2)	(*str.prefixof s2 s1*)
equals(s1, s2)	(= *s1 s2*)	substring(s1, i1)	(*str.substr s1 i1* (*str.lens1*))
indexOf(s1, s2)	(*str.indexof s1 s2* 0)	substring(s1, i1, i2)	(*str.substr s1 i1* (− *i2 i1*))
indexOf(s1, s2, i1).	(*str.indexof s1 s2 i1*)		

with symbolic encoding in SPouT that introduce an abstraction of the bytecode semantics of the JVM in the symbolic operation. The substitution methods intercept the method invocation and encodes the effects symbolically. But for continuing the concrete execution after the return of the string operation, SPouT must execute the string operation concretely. This concrete execution must not be visible in the constraint tree, as the symbolic encoding describes the semantic already using a higher abstraction. Therefore, SPouT pauses the general bytecode level constraint recording in the scope of the substitution method and resumes it on return, when it leaves the scope.

Faithful Error Handling. Consider the example in Figure 3. On the left, we have a path constraint that is similar to a previously discussed example by Redelinghuys et al. [18] for comparing automatons and bitvector encodings. Assume that `s1` is a concolic string variable and recorded constraints require `s1` to be a prefix of the string `"abc"` but the first character of `s1` must not be `'a'`. Under the `Java` semantics, this is unsatisfiable: the empty string throws an `IndexOutOfBoundsException` when accessing its 0-th character and non-empty strings cannot match both constraints. SMT-Lib defines the `str.prefixof` and `str.at` operations that are mostly comparable to the `Java` methods `startsWith` and `charAt`. The black part on the right of the figure shows the direct mapping from `Java` to these functions. But there are differences in the semantics of corner cases. In SMT-Lib, the problem is satisfiable by the empty string, since accessing an index beyond the string length yields an *error value* that is not equal to `"a"`.

```
void main(String[] args) {
  String arg =
    Verifier.nondetString();
  if (arg.length() < 1) return;
  char c = arg.charAt(0);
  assert Character.toUpperCase(c)
    != Character.toLowerCase(c);
}
```

Fig. 4: The StaticCharMethods02 task from SV-COMP 2022.

```
// ...
25: iload_2
26: invokestatic #6 //toUpperCase
29: iload_2
30: invokestatic #7 //toLowerCase
33: if_icmpne    44
36: new          #8
// ...
```

Fig. 5: The bytecode for the assert statement from Figure 4.

Therefore, for the concolic analysis of the `charAt(i)` method on the symbolic string `s1`, a check on the index is required. The index `i` must be greater or equal to zero and less than the string length of `s1` to be successful in the `Java` SE library. The green constraint on the right side is an added implicit assumption modeling that `charAt` did not throw an exception. This implies that encoding a single `Java` constraint may yield two path constraints in SMT-Lib, one modeling invalid access operations and one modeling valid ones. The exception path and the branching condition guarding it becomes visible in the tree.

Numeric and String Semantics of Characters. `Java` uses a character type for its string and numeric semantic, but the compiled bytecode only utilizes integer operations. In contrast, the SMT-Lib is only aware of the string semantic for a character. Therefore, for encoding a problem in SMT-Lib, the string theory semantic is sometimes easier. For example, consider the `charAt` method. In the SMT-Lib language, `charAt` returns a value of type string. In the `Java` language, it is a character. Inside the JVM, the character is represented as an integer. Figure 3 encodes the comparison of the charAt result against the character 'a' as a comparison in the string semantic using the SMT-Lib. Otherwise, this small example involves in the encoding of the numeric equality many cast operations from a string to an integer to a 32-bit bitvector. For this case, lifting from the `Java` representation to the string semantic is easier than using casting expressions in the symbolic encoding.

Consider the example in Figure 4. The character at position zero is extracted from a nondeterministic string. Next, the assertion statement compares the result from the to upper case and lower case conversion. Figure 5 shows the compiled bytecode of the assertion check in Figure 4. The `invokestatic` bytecode takes the integer from the stack and replace it with another integer. This is a lookup in a large map that cannot be encoded in SMT-Lib using an numeric value in the bitvector theory. On the other side, some solvers, e.g., CVC4, support the functions `toUpper` and `toLower` in the string theory. Using them, it is possible to encode the assertion semantic in the string theory. Assuming ARG represents the symbolic string, the logical encoding is: $(= \ (toUpper\ (str.at\ \mathsf{ARG}\ 0))\ (toLower\ (str.at\ \mathsf{ARG}\ 0)))$. To allow this flexible in encoding the semantic, the character values in the JVM require a string semantic and a bitvector semantic equivalent in the encoding. SMT-Lib supports

Table 2: Functions that cannot be mapped directly and precisely from the Java standard string library to the SMT-Lib language, or do not maintain their semantic.

Java Operation	SMT-Lib Operation	Comment
charAt(s1, i1)	$(str.at\ s1\ i1)$	The charAt function requires some error handling in Java not represented in the SMT-Lib function $str.at$.
compareTo(s1, s2)	$(str.\ <\ s1\ s2)$ $(str.\ <=\ s1\ s2)$	SMT-Lib has lexicographic ordering operations but they need to be embedded in the evaluation of compareTo splitting the three value result logic to binary decisions.
compareTo-IncoreCase(s1, s2)	–	There is no mapping in SMT-Lib allowing the encoding of the ignore case semantic. Using solver specific operations as toUpper allow to work around this limitation.
equals-IgnoreCase(s1, s2)	–	The same problem as for compareToIcnoreCase applies.
isLetter(c1) isUpperCase(c1)	–	It is possible to use $str.to\_code$ to convert $c1$ into a code point. But afterwards the unicode table defining which code points are within the target domain have to be encoded as well. In practice, we have only archived to encode this for limited ranges on the code point.
join(s1, s2[])	–	There is no way for expressing a join on a symbolic string array yet as we have not really a way to express the capacity of an array symbolically.
lastIndexOf(s1, s2)	(declare-const x Int) $(and\ (=\ (str.at\ s1\ x)\ s2)$ $(not\ (exists\ ((y\ Int))$ $(and\ (<\ x\ y)\ (<\ y\ (str.len\ s1))$ $(not\ (=\ (str.at\ s1\ y)\ s2))))))$	We can encode this using helper variables, but it is leaving the QF_SLIA theory as quantifiers are required. Therefore, the encoding is not within the official theory definition of the SMT-Lib anymore as $SLIA$ is not defined as official theory.
matches(s1, s2)	$(str.in\_re\ s1\ \ldots)$	Depending on the complexity of the pattern involved in $s2$, this is possible. But the pattern contained in $s2$ needs to be transformed to SMT-Lib first.
parseFP(s1, fpSize) parseInt(s1)	–	It is not possible to model this in SMT-Lib at the moment.
split(s1, s2)	–	It is not possible to transfer this except for a concrete example describing that the concatenation of the new subparts with the separator s2 are equal to s1.
reverse(s1)	–	Reversing a string is not supported in todays SMT solvers.
strip(s1)	$(and\ (not\ (=\ (str.at\ s1\ 0)\ "\ "))$ $(not\ (=\ (str.at\ s1\ (-$ $(str.len\ s1)\ 1))\ "\ ")))$	While this encoding implies that the first and last character are no whitespaces, it is no possible to express that a string might be shorter after strip in this encoding.
stripIndent(s1) stripLeading(s1) stripTrailing(s1) trim(s1)	–	See the problem with strip. The same applies for these methods.
toString(fp1) toString(i1) toString(c1)	–	There is no symbolic encoding in SMT-Lib that allows to convert a numeric value in its string representation.
toUpper(s1) toLower(s1)	$(str.upper\ s1)$ $(str.lower\ s1)$	These functions are not supported in the official SMT-Lib standard. CVC4 supports it as a custom interface.
insert(s1, s2, i1) delectCharAt(s1, i1) delete(s1, i1, i2)	-	These functions do not have a counterpart in SMT-Lib but can be encoded using $substring$ to split the existing string and gluing the remaining parts together using $concat$.

`str.to_code` and `nat2bv` that allows to express the numeric code point of a String into a bitvector, but it is not a true semantic link representing the dual semantics of the character data type in the `Java` language in SMT-Lib. At the moment, SPouT uses mainly the string semantics, but we are still investigating the best way to deal with this representation problem in a more general solution that also support cases where the character is used numerically in the program.

Encoding Regular Expressions. In general, the `Java` string library separates two kinds of regular expressions: those that use backreferences and those that do not. Backreferences in regular expression work similar to the Perl regular expression language[11]. In the `Java` context they are also called capture groups. For example, a regular expression with a capture group is: *name: (*)?$*. It matches any character after the string "name: " until the end of the line. `Java` allows

[11] https://www.pcre.org

to extract the group, the part matched between the brackets, if the regular expression matches. E.g., if the string is "name: SPouT", the group is SPouT. SMT-Lib does not support backreferences in the regular expression language and SPouT does not support them in the encoding. However, Loring et al. [11] present ideas on encoding and solving constraints with regular expressions using groups for JavaScript by applying a CEGAR based algorithm in combination with SMT solvers.

Regular expressions without capture groups are supported and allow the encoding of string operations like `matches` or `replace`. The main technical challenge is transforming the regular expression into the automaton that is encoded in the SMT-Lib. For example, consider the following regular expression in the `Java` language: "Date: \d \d-\d\d-\d\d". It has to be decomposed into the different parts first and then combined into the SMT-Lib constraint. SPouT only reports the regular expression in the `Java` syntax and the tool that uses this encoding has to parse the `Java` regular expression string into an SMT-Lib constraint. GDart uses the brics automaton library[12] for the conversion from the string representation into an automaton representation. The first step is resolving the `Java` specific range definition, in this case "\d" for a digit. An equivalent regular expression is "Date: [0-9][0-9]-[0-9][0-9]-[0-9][0-9]". This regular expression is then converted to SMT-Lib:

$$
\begin{aligned}
(re.++(str.to\_re\,\text{"}Date:\text{"})\\
&(re.range\,\text{"}0\text{"}\,\text{"}9\text{"})\,(re.range\,\text{"}0\text{"}\,\text{"}9\text{"})\,(str.to\_re\,\text{"}-\text{"})\\
&(re.range\,\text{"}0\text{"}\,\text{"}9\text{"})\,(re.range\,\text{"}0\text{"}\,\text{"}9\text{"})\,(str.to\_re\,\text{"}-\text{"})\\
&(re.range\,\text{"}0\text{"}\,\text{"}9\text{"})\,(re.range\,\text{"}0\text{"}\,\text{"}9\text{"}))
\end{aligned}
$$

In this final form, the regular expression can be used in the symbolic encoding of operations. The values produced in a SMT-Lib model for such an expression matches the `Java` semantic during concrete execution.

Limitations. The presented encoding method works well for modeling operations on strings and the concatenation helper methods that do not involve code point handling (e.g., `concat`, `startsWith`, `equals`, and `indexOf`). Table 2 shows methods that are part of the `Java` standard library, but currently have no direct semantic counter part in the SMT-Lib language. We partition them in roughly four groups: Those that can be expressed combining other SMT-Lib functions, those that cannot be expressed, those that require restrictions on character, and those that are used for value serialization and deserialization.

E.g., `compareTo` and `insert` can be modeled using a combination of multiple SMT-Lib functions. An insertion can be expressed by creating two substrings from a string and putting them together again by concatenating the first part, the new content between and the second part.

Some methods require unbounded path enumeration in the encoding. A prominent example is the `split` method that cannot be expressed in SMT-Lib yet. For the current path, it is possible to encode the structure of the concrete

[12] https://www.brics.dk/automaton/

string and how a new string leading to an increased array looks like. A semantic-preserving encoding of the `split` result as a symbolic array is not yet possible.

Many of the functions that are not expressible in SMT-Lib at the moment apply restriction on certain characters of the string or a single character, e.g., `isLetter`, `strip`, and `trim`. Without support in the string theory, e.g., encoding an equality comparison between a trimmed string value and its not trimmed counterpart is impossible. In `Java`, an example for this case that evaluates to true is: " Hello".trim().equals("Hello"). As SMT-Lib does not have a notion of a single character and custom range checks on them, encoding is impossible.

The last group of problems includes serialization and deserialization for different primitive data types, e.g. `parseFloat`, which converts a `String` into a `float`. This function cannot be expressed symbolically in the current version of SMT-Lib. Supporting the parsing function requires also linking the two theories as a single variable has a representation in both theories.

2.4 Supported Languages and Implemented Features

SPouT aims to analyze JVM bytecode programs and can – in theory – process any program that is compiled to JVM bytecode using only primitive types (e.g., `Java`, Scala, and Kotlin programs with primitive types). As mentioned previously, for higher level data types, e.g., strings, a modeling of the standard library in the form of substitute methods is required. We developed substitutes for `Java` programs as a part of SPouT. Using SPouT with other languages as Scala or Kotlin requires additional standard library abstractions suitable to the language, although it is already possible to load programs in these languages, including their runtime libraries, using SPouT.

SPouT analyzes all JVM primitive types (i.e., `boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, and `double`) concolically (including boxed objects) by generating symbolic constraints in SMT-Lib bitvector and floating point theory. It also tracks concolic array length for one-dimensional and multi-dimensional arrays and models `System.arraycopy`, enabling analysis of collections and arrays.

Since GraalVM is a polyglot virtual machine and Espresso implements a JVM as a guest language, SPouT benefits from GraalVM's JIT optimization.

3 Demonstration and Evaluation

We evaluate the versatility SPouT in a number of small usage examples by comparing it to other tools for the analysis of `Java` programs. We have demonstrated its performance in SV-COMP 2022 for the first time, where we used SPouT as the concolic executor in the GDart tool ensemble. This paper, takes a closer look on the performance of the string encoding in comparison with other tools. In addition, we show two examples, why SPouT will stimulate future research.

Performance of the String Encoding. Since modeling of string operations is a major challenge when analyzing the security of `Java` web applications, we

Table 3: Comparison of the SV-COMP tools in the `Java` track on the securibench task subset consisting of 113 task in total. The results are taken from the official SV-COMP runs.

Result \ Tool	GDart (using SPouT)	JDart	SPF	Java Ranger	JayHorn	COASTAL	JBMC
Correct	95	100	85	76	0	22	100
Incorrect	0	0	3	0	0	90	0
Unkown	10	12	22	37	113	1	8
Error	8	1	3	0	0	0	5

Table 4: Comparing GDart's capability enabled by SPouT with other `Java` tools from SV-COMP on the described examples. (+) means the tool solves the example with the expected verdict, (×) means the tool does not solve the example, (−) does not reach the expected verdict, and † means the tool crashes or times out (15 min). - means we could not run the tool on the example.

Tool \ Example	java11	Scala	Kotlin	JIT	Maven
GDart (using SPouT)	(+) 4.9 s	(+) 10.7 s	(+) 5.7 s	(+) 26.4 s	(+) 69.1 s
JDart	(−) 2.7 s	-	-	†	-
SPF	(×) 0.9 s	-	-	(×) 1.1 s	-
Java Ranger	(×) 0.9 s	-	-	(×) 2.3 s	-
JayHorn	(−) 2.4 s	-	-	†	-
JBMC	(×) 1.1 s	-	-	(+) 1.5 s	-
COASTAL	(×) 1.1 s	-	-	(×) 1.2 s	-

added the securibench benchmark suite[13] [9,10] to the set of `Java` instances in SV-COMP 2022. The securibench benchmark set is inspired by web application security threats and contains many instances that use String operations. Table 3 compares the different tools reporting results for SV-COMP 2022 on this subset of tasks. JDart and JBMC both solve 100 tasks correctly, five more than GDart. Compared to Java Ranger, GDart correctly solves 19 more tasks. JayHorn solves none of these tasks. The only tools reporting incorrect answers are SPF (3) and COASTAL (90). For the 10 tasks GDart reports as unknown, 5 are due to a triggered exception in the symbolic reasoning component, 3 cannot be solved as the `StringTokenzier` is not symbolically modeled in SPouT, and two are due to problem specific errors. For the 8 error tasks, GDart exhausts the resource limits. While the summary suggests that JDart and JBMC solve 5 more tasks than GDart, this is not the case. JDart solves 4 tasks for which GDart exhausts the resources and also solves 4 tasks for which GDart reports unknown results. The resource exhaustion happens in the symbolic execution of GDart and are not explicitly related to SPouT. However, GDart

[13] https://github.com/tudo-aqua/securibench-micro

solves 3 tasks that JDart does not solve due to different instrumentations of the `toLowerCase` method. In direct comparison with JBMC, GDart solves 6 tasks that JBMC does not solve mostly, because GDart supports reflection within the JVM, and JBMC does not. On the other side, JBMC solves 5 tasks for which GDart exhausts the resource limit and 6 tasks that GDart does not solve. The distances for unknown tasks are less surprising considering that JDart and JBMC support operations on the `StringTokenizer`.

Performance in SV-COMP 2022. GDart solved 471 out of 586 tasks using SPouT, which is the third highest amount of correctly solved task, following behind JBMC (506) and JDart (522). As GDart is by design stronger in finding errors (302) than proving the absence of an error (169), it ranks in the SV-COMP point schema fourth after Java Ranger (solving 466 tasks, 204 without error, 262 with error) (c.f. SV-COMP results[14]).

Demonstration of Versatility. We have implemented a number of demonstration examples[15] for GDart's capability enabled by SPouT and report the performance of other `Java` tools in Table 4 in comparison. SPouT allows GDart to run more examples than any other tool. The examples demonstrate the following:

Modern Java Byte Codes. Being built on top of a full-fledged JVM, SPouT is capable of analyzing modern `Java` bytecode that uses `Java` 11 features as demonstrated by the `java11` example. This is a major advantage of SPouT compared to JPF-VM based tools that load only `Java` 8 bytecode.

Analysis of arbitrary JVM languages. SPouT allows execution of arbitrary JVM bytecode programs, even compiled Kotlin and Scala programs, if they are loaded along with their runtimes. The `Kotlin` and `Scala` examples demonstrate this. However, as mentioned previously, support for the Kotlin and Scala standard library is incomplete.

JIT Optimization. Since SPouT runs on the GraalVM, it benefits from the GraalVM's just-in-time compiler. In the `jit` example, we can observe a 2.5-fold speed-up for hot code during concolic execution. On the other hand, it has to be noted that the execution with Espresso (version 21.2.0) is 10 to 20 times slower than the native GraalVM in our examples.

Maven and SpringBoot. The `springboot` example shows how SPouT can be used out-of-the-box as the JVM for concolically executing test cases in a Maven build process. We demonstrate this for a test case of a containerized Spring Boot web application with mocking and code injection that uses the complete Spring Boot application stack.

The `springboot` example, in particular, shows the strength of the tool architecture of SPouT: We were not able to run the other tools on this example as the test is executed as a JUnit test case from within a build tool (maven). SPouT is implemented as a feature of a `Java` executable and simply executes the build

[14] https://sv-comp.sosy-lab.org/2022/results/results-verified/
[15] available on GitHub: https://github.com/tudo-aqua/gdart-examples

system. We enable the concolic execution feature only for the unit test. We are confident, though we could not test it, that the other tools cannot analyze the unit test since it starts a Tomcat web container including database, injects mock code into the test case (generated and compiled at runtime), and relies on interception of method calls (also configured at runtime). As a consequence, the actual behavior of the application only emerges during execution in the Tomcat.

Enabler for Future Research Projects. The versatility of SPouT and the industry grade GraalVM running it allows scaling our security detection research further and investing into scalable dynamic symbolic execution engines. We will explain this in two examples. Both examples are currently not runnable with the JPF-VM and demonstrate the large potential for future research enabled in this area using the concolic runner SPouT. The examples are:

Evaluating Jaint on Jenkins. The Jaint framework [16] combines dynamic symbolic execution and dynamic tainting. Today, it is build on top of the JPF-VM using JDart. After evaluating Jaint on benchmarks, the next scaling step is detecting existing injection vulnerabilities, e.g., the OS command injection in the Jenkins Git Client Plugin 2.8.4[16]. But the JPF-VM prevented that we have been able to analyze these kinds of tasks. Due to the incomplete support of the `Java` standard library in the JPF-VM, it was not possible to create temporary directories required to model a concrete driver for analyzing the plugin dynamically. Using the new SPouT executor, we are today able to run the examples dynamically and record constraints. This enables future research project that focus on the application of dynamic symbolic execution for the analysis of `Java` and, therefore, also allows empirical experiments that measure Jaint's scalability.

Dynamic Symbolic Execution of Log4Shell. Using SPouT, we have successfully analyzed the Log4Shell example project[17] with dynamic symbolic execution. As of today, we can demonstrate that the symbolic annotations for dynamic symbolic execution travel across the project and the method substitution pattern works on this spring boot application as well for injecting monitors on the Log4j classes.

4 Conclusion

In this paper, we presented SPouT, a concolic executor build on top of the GraalVM. We detailed the integration of the symbolic annotation recording into the concrete bytecode execution of the Espresso JVM and, for the first time, formalized SPouT's symbolic trace format. Moreover, we present how GraalVM features are used to encode symbolic operations on strings at the library level. We use SPouT as the concolic executor of GDart, but are confident that the component might be useful for other research projects as well, e.g., for fuzzing `Java` programs.

[16] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-10392
[17] https://github.com/christophetd/log4shell-vulnerable-app

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. Lecture Notes in Computer Science, vol. 8471, pp. 55–71. Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-12154-3_4, VSTTE 2014
2. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (May 2021), https://smtlib.cs.uiowa.edu, release: 2021-05-12
3. Bjørner, N., Tillmann, N., Voronkov, A.: Path feasibility analysis for string-manipulating programs. In: Kowalewski, S., Philippou, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 5505, pp. 307–321. Springer, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00768-2_27, TACAS 2009
4. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise analysis of string expressions. In: Cousot, R. (ed.) Static Analysis. Lecture Notes in Computer Science, vol. 2694, pp. 1–18. Springer, Berlin, Heidelberg (2003). https://doi.org/10.1007/3-540-44898-5_1, SAS 2003
5. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-c programs. In: Jensen, K., Podelski, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 2988, pp. 168–176. Springer, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_-15, TACAS 2004
6. Cordeiro, L., Kesseli, P., Kroening, D., Schrammel, P., Trtik, M.: JBMC: A bounded model checking tool for verifying Java bytecode. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification. Lecture Notes in Computer Science, vol. 10981, pp. 183–190. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_10, CAV 2018
7. Kahsai, T., Rümmer, P., Sanchez, H., Schäf, M.: JayHorn: A framework for verifying Java programs. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification. Lecture Notes in Computer Science, vol. 9779, pp. 352–358. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_19, CAV 2016
8. Kloibhofer, S., Pointhuber, T., Heisinger, M., Mössenböck, H., Stadler, L., Leopoldseder, D.: SymJEx: symbolic execution on the GraalVM. In: Proceedings of the 17th International Conference on Managed Programming Languages and Runtimes. pp. 63–72. Association for Computing Machinery, New York, NY, USA (Nov 2020). https://doi.org/10.1145/3426182.3426187, MPLR 2020
9. Livshits, B.: Improving Software Security with Precise Static and Runtime Analysis. Ph.D. thesis, Stanford University (2006)
10. Livshits, V.B., Lam, M.S.: Finding security vulnerabilities in Java applications with static analysis. In: 14th USENIX Security Symposium. pp. 271–286. USENIX Association, San Diego, CA (2005), https://www.usenix.org/legacy/publications/library/proceedings/sec05/tech/livshits.html, SEC 2005
11. Loring, B., Mitchell, D., Kinder, J.: Sound regular expression semantics for dynamic symbolic execution of JavaScript. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp.

425–438. Association for Computing Machinery, New York, NY, USA (Jun 2019). https://doi.org/10.1145/3314221.3314645, PLDI 2019
12. Luckow, K., Dimjašević, M., Giannakopoulou, D., Howar, F., Isberner, M., Kahsai, T., Rakamarić, Z., Raman, V.: JDart: A dynamic symbolic analysis framework. In: Chechik, M., Raskin, J.F. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 9636, pp. 442–459. Springer, Berlin, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_-26, TACAS 2016
13. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24, TACAS 2008
14. Mues, M., Howar, F.: Data-driven design and evaluation of SMT meta-solving strategies: Balancing performance, accuracy, and cost. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 179–190. IEEE, New York (Nov 2021). https://doi.org/10.1109/ASE51524.2021.9678881, ASE 2021
15. Mues, M., Howar, F.: GDart: An ensemble of tools for dynamic symbolic execution on the Java Virtual Machine (competition contribution). In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 13244, pp. 435–439. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-030-99527-0_27, TACAS 2022
16. Mues, M., Schallau, T., Howar, F.: Jaint: A framework for user-defined dynamic taint-analyses based on dynamic symbolic execution of Java programs. In: Dongol, B., Troubitsyna, E. (eds.) Integrated Formal Methods. Lecture Notes in Computer Science, vol. 12546, pp. 123–140. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-63461-2_7, IFM 2020
17. Păsăreanu, C.S., Visser, W., Bushnell, D., Geldenhuys, J., Mehlitz, P., Rungta, N.: Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. Automated Software Engineering **20**(3), 391–425 (Sep 2013). https://doi.org/10.1007/s10515-013-0122-2
18. Redelinghuys, G., Visser, W., Geldenhuys, J.: Symbolic execution of programs with strings. In: Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference. pp. 139–148. Association for Computing Machinery, New York, NY, USA (Oct 2012). https://doi.org/10.1145/2389836.2389853, SAICSIT 2012
19. Shannon, D., Ghosh, I., Rajan, S., Khurshid, S.: Efficient symbolic execution of strings for validating web applications. In: Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009). pp. 22–26. Association for Computing Machinery, New York, NY, USA (Jun 2009). https://doi.org/10.1145/1555860.1555868, DEFECTS 2009
20. Shannon, D., Hajra, S., Lee, A., Zhan, D., Khurshid, S.: Abstracting symbolic execution with string analysis. In: Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007). pp. 13–22. IEEE, New York (Sep 2007). https://doi.org/10.1109/TAIC.PART.2007.34, TAICPART-MUTATION 2007
21. Sharma, V., Hussein, S., Whalen, M.W., McCamant, S., Visser, W.: Java Ranger: statically summarizing regions for efficient symbolic execution of Java. In: Pro-

ceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 123–134. Association for Computing Machinery, New York, NY, USA (Nov 2020). https://doi.org/10.1145/3368089.3409734, ESEC/FSE 2020

22. Spoto, F.: The Julia static analyzer for Java. In: Rival, X. (ed.) Static Analysis. Lecture Notes in Computer Science, vol. 9837, pp. 39–57. Springer, Berlin, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53413-7_3, SAS 2016
23. Tillmann, N., de Halleux, J.: Pex–white box test generation for .NET. In: Beckert, B., Hähnle, R. (eds.) Tests and Proofs. Lecture Notes in Computer Science, vol. 4966, pp. 134–153. Springer, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-79124-9_10, TAP 2008
24. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Automated Software Engineering **10**(2), 203–232 (Apr 2003). https://doi.org/10.1023/A:1022920129859
25. Würthinger, T., Wimmer, C., Wöß, A., Stadler, L., Duboscq, G., Humer, C., Richards, G., Simon, D., Wolczko, M.: One VM to rule them all. In: Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software. pp. 187–204. Association for Computing Machinery, New York, NY, USA (Oct 2013). https://doi.org/10.1145/2509578.2509581, Onward! 2013