ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/347697652

Java Ranger: statically summarizing regions for efficient symbolic execution of Java

Conference Paper · November 2020

DOI: 10.1145/3368089.3409734

CITATIONS
44

READS
385

5 authors, including:

Vaibhav Sharma
University of Minnesota Twin Cities
21 PUBLICATIONS **208** CITATIONS

SEE PROFILE

Soha Hussein
University of Minnesota Twin Cities
11 PUBLICATIONS **84** CITATIONS

SEE PROFILE

Michael William Whalen
Amazon
120 PUBLICATIONS **2,827** CITATIONS

SEE PROFILE

Stephen McCamant
University of Minnesota Twin Cities
75 PUBLICATIONS **4,124** CITATIONS

SEE PROFILE

All content following this page was uploaded by Vaibhav Sharma on 28 December 2020.

The user has requested enhancement of the downloaded file.

Java Ranger: Statically Summarizing Regions for Efficient Symbolic Execution of Java

Vaibhav Sharma*
vaibhav@umn.edu
University of Minnesota
Minneapolis, MN, USA

Soha Hussein*
soha@umn.edu
University of Minnesota
Minneapolis, MN, USA
soha.hussein@cis.asu.edu.eg
University of Ain Shams
Cairo, Egypt

Michael W. Whalen
mwwhalen@umn.edu
University of Minnesota
Minneapolis, MN, USA

Stephen McCamant
mccamant@cs.umn.edu
University of Minnesota
Minneapolis, MN, USA

Willem Visser
wvisser@cs.sun.ac.za
Stellenbosch University
Stellenbosch, South Africa

ABSTRACT

Merging execution paths is a powerful technique for reducing path explosion in symbolic execution. One approach, introduced and dubbed "veritesting" by Avgerinos et al., works by translating a bounded control flow region into a single constraint. This approach is a convenient way to achieve path merging as a modification to a pre-existing single-path symbolic execution engine. Previous work evaluated this approach for symbolic execution of binary code, but different design considerations apply when building tools for other languages. In this paper, we extend the previous approach for symbolic execution of Java.

Because Java code typically contains many small dynamically dispatched methods, it is important to include them in multi-path regions; we introduce *dynamic inlining of method-regions* to do so modularly. Java's typed memory structure is very different from the binary representation, but we show how the idea of static single assignment (SSA) form can be applied to object references to statically account for aliasing.

We have implemented our algorithms in Java Ranger, an extension to the widely used Symbolic Pathfinder tool. In a set of nine benchmarks, Java Ranger reduces the running time and number of execution paths by a total of 38% and 71% respectively as compared to SPF. Our results are a significant improvement over the performance of JBMC, a recently released verification tool for Java bytecode. We also participated in a static verification competition at a top theory conference where other participants included state-of-the-art Java verifiers. JR won first place in the competition's Java verification track.

*These authors contributed equally to this research.

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.
ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7043-1/20/11...$15.00
https://doi.org/10.1145/3368089.3409734

CCS CONCEPTS

• **Theory of computation** → *Program analysis*.

KEYWORDS

symbolic execution, veritesting, path-merging

ACM Reference Format:
Vaibhav Sharma, Soha Hussein, Michael W. Whalen, Stephen McCamant, and Willem Visser. 2020. Java Ranger: Statically Summarizing Regions for Efficient Symbolic Execution of Java. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3368089.3409734

1 INTRODUCTION

Symbolic execution is a popular analysis technique that performs non-standard execution of a program: data operations generate formulas over inputs, and branch constraints along an execution path are combined into a predicate. Originally developed in the 1970s [19], [10], symbolic execution is a convenient building block for program analysis, since arbitrary query predicates can be combined with the logical program representation, and solutions to these constraints are program inputs illustrating the queried behavior. Some of the applications of symbolic execution include test generation [15, 26], equivalence checking [25, 28], vulnerability finding [31, 32], program repair [22], invariant discovery [1], and protocol correctness checking [33]. Symbolic execution tools are available for many languages, including CREST [7] for C source code, KLEE [8] for C/C++ via LLVM, JDart [21] and Symbolic PathFinder (SPF) [24] for Java, and S2E [9], FuzzBALL [4], and `angr` [31] for binary code.

Although symbolic analysis is a popular technique, scalability is a substantial challenge for many applications. In particular, symbolic execution can suffer from a *path explosion*: complex software has exponentially many execution paths, and baseline techniques that explore one path at a time are unable to cover all paths. Dynamic state merging [16, 20] provides one way to alleviate scalability challenges by opportunistically merging execution paths. Avoiding

even a single branch point can provide a multiplicative savings in the number of execution paths, though at the potential cost of making symbolic state representations more complex.

Veritesting [3] is another technique that can dramatically improve the performance of symbolic execution by effectively merging paths. Rather than explicitly merging state representations, veritesting identifies arbitrary fragments of code and encodes it as a disjunctive predicate for symbolic execution. This encoding allows many paths to be collapsed into a single path.

In previous work [3], constructing a bounded static representation of code fragments was shown to allow symbolic execution the ability to find more bugs, and achieve more node and path coverage, when implemented at the X86 binary level for compiled C programs. This motivates us to investigate the benefit of bounded static representation of bytecode fragments for the symbolic execution of Java programs. There are substantial differences between compiled Java programs and C programs. In C programs, most functions are statically dispatched and exceptions do not occur, allowing C compilers to inline code and create relatively large code fragments without non-local jumps. In Java programs, most functions are dynamically dispatched, methods tend to be small, and the compiler assumes an "open-world" so most functions are not inlined. In addition, many, if not most, Java bytecodes can throw exceptions, leading to many small, dynamically dispatched fragments of code with many non-local control jumps. In a naive implementation, such non-local jumps reduce the size of the path-merged code that can be created and increase the branching factor for exploration, leading to poor performance. This makes Java more challenging for creating bounded static representations.

In this paper, we present Java Ranger, an extension of Symbolic PathFinder. Java Ranger operates by identifying particular forms of bytecode fragments that we call a *region*. A region can be one of two types: a *Multi-Path Region*, which corresponds to the Java bytecode fragment of an if-statement, and a *Method Region*, which corresponds to Java bytecode that spans the definition of a method. Java Ranger then utilizes dynamic information from Dynamic Symbolic Execution (DSE) environment such as stack slot and heap values, and later uses them in a series of transformations to perform path-merging. Java Ranger's transformations are designed to transform features of the Java language to a solver constraint.

In our experiments, we demonstrate exponential speedups on benchmarks (in general, the more paths contained within a program, the larger the speedup) over the unmodified Java SPF tool using this approach.

We make the following contributions in this paper:

(1) We propose *Dynamic Method Region Inlining*: this allows us to construct summaries for multi-path regions containing dynamically dispatched method calls, once types are known.
(2) We propose a technique named *Single-Path Cases* for splitting regions into exceptional and non-exceptional outcomes, and use Dynamic Symbolic Execution for the exceptional cases.
(3) We propose *Early-Returns Summarization* to collapse multiple returns into a disjunctive-conditional returned-expressions.

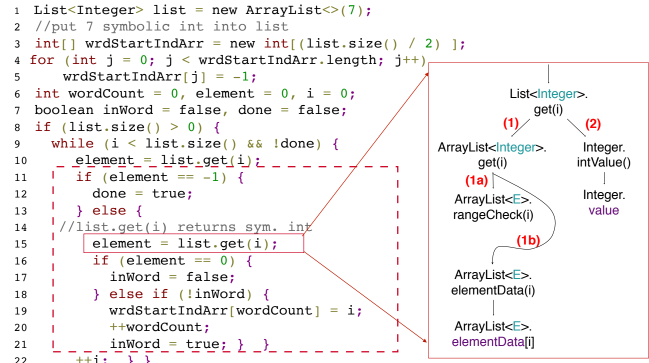

Figure 1: An example where Java Ranger summarizes two multi-path regions

1.1 Motivating Example

Consider the example shown in Figure 1. The code computes the number of words in a list and stores the starting index of each word in `wrdStartIndArr`. The `list` variable refers to an `ArrayList` of 7 `Integer` objects, each of which have an unconstrained symbolic integer as a field. The size of `wrdStartIndArr` is set to half the size of `list`, to account for the maximum possible words that can occur in `list` if all words are one character long (line 3). All elements in `fstWrdIndexArr` are initialized to -1 (lines 4-5). In this example, the concrete value `0` acts as a delimiter for words and the value `-1` acts as string terminator. Two conditions cause execution to exit the while loop, (1) if if all elements in the `list` have been processed or (2) if a string terminator (`-1`) is found.

This code has a bug. Consider the case when the `list` has the following 7 values: `{1,0,1,0,1,0,1}`. In this case, there are 4 words, where each word is of size 1 character and where 0, 2, 4, and 6 are the starting indices of each word. But, the allocated size of `wrdStartIndArr` is 3 elements since we performed integer division on line 3 (`7/2 = 3`). This incorrect allocation causes an `ArrayOutOfBoundsException` when trying to store the first index of the 4th word in `wrdStartIndArr` at line 19. We ran this code symbolically with the depth-first search heuristic with a dynamic symbolic executor (Symbolic PathFinder) and found that it explored 173 execution paths before finding this bug. This number of explored execution paths depends on the number of symbolic inputs (7 in this example) when exploring with Symbolic PathFinder. Java Ranger however, can find this bug after exploring only one execution path, regardless of the number of symbolic inputs. Java Ranger explores two kinds of outcomes through path-merging: (a) the non-exceptional outcome which includes within-bounds array access , and (b) an exceptional outcome, which explores, in this case the ArrayOutOfBoundsException to find the bug. Java Ranger is able to achieve this reduction in execution paths by merging the paths arising out of the if-branch in line 11 through 21 and exploring all non-exceptional outcomes in a single execution path.

Path-merging of this simple region is not straightforward. The call to `list.get(int)` at the source level results in the following sequence of method calls (Figure 1): (1) It calls `ArrayList<Integer>.get(int)` which internally does two things,

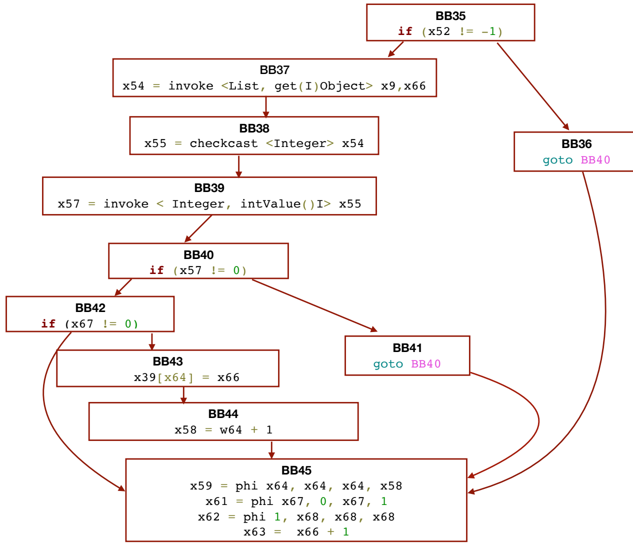

Figure 2: A subgraph of the CFG representing the Region in dashed box in Figure 1, where BB35 and BB45 is the entry and exit blocks of the subgraph, respectively.

(a) It checks if the index argument accesses a value within bounds of the `ArrayList` by calling `ArrayList<E>.rangeCheck(int)`. If this access is not within bounds, it throws an exception. And (b) It calls `ArrayList<E>.elementData(int)` to access an internal array named `elementData` and get the entry at position `i`. This call results in an object of class `Integer` being returned. (2) It calls `Integer.intValue()` on the object returned by the previous step. This call internally accesses the `value` field of the `Integer` to return the `int` primitive value of this object.

The method to be inlined depends on the dynamic type of the object reference for the invoked method. In this example, the dynamic type of `list` is an `ArrayList`, whereas it is declared static type is `List`. Path-merging requires not only inlining the right method but also accounting for the possibility of an exception being raised by `ArrayList<E>.rangeCheck(i)`.

1.2 Java Ranger Overview

Java Ranger operates on top of DSE and attempts to merge paths by creating a *Region Constraint*, a disjunctive formula describing the behavior of the region. It integrates with DSE by utilizing three main features of DSE: (1) Path Condition (*PC*), which is a condition on the input symbols such that if a path is feasible its path condition is satisfiable. Java Ranger adds the constraint describing the merged region to the *PC*. (2) Program Counter (*pc*), which points to the instruction to be executed. Java Ranger changes the *pc* to skip symbolic execution of a successfully merged region or to direct DSE to execute unmerged paths. (3) Runtime/Instantiation time information of local variables on the stack and the heap. Java Ranger utilizes this information to construct a region constraint.

To do path merging, Java Ranger intercepts any symbolic branching *instruction* during DSE, and attempts to recover the corresponding if-then-else *statement structure* by recognizing instructions that belongs to its "then" or "else" side. The recovered statement is represented in Java Ranger's *Intermediate Representation* (IR) which we call a *static statement*. For example, Listing 1 in Figure 3 shows the recovered static statement of the bytecode corresponding to the code region between lines 11-21 in Figure 1. We call this process *Statement Recovery* and it is part of the *Static phase*, described in Section 3.2. The input to the statement recovery is a Control Flow Graph (CFG) of the corresponding region. Java Ranger uses Wala [17] to construct the CFG of the bytecode region. Figure 2 shows the CFG of the multi-path region in lines 11-21.

Given the static statement from the above step, Java Ranger then tries to find instantiation/runtime information in DSE to concretize values of variables and references used in the static statement. We call this phase the *Instantiation Phase* and it consists of nine *instantiation-time* transformations described in Section 3.4. The output of each of these transformations is a more refined, rewritten version of the static statement in Java Ranger IR. We call this output *instantiated statement*. For example, Listing 2 in Figure 3 shows the *instantiated statement* resulting from the *Substitution* transformation with runtime information substituted from DSE, e.g., $x9$ is substituted with the instantiation time object reference 375 of `List` obtained by reading a stack slot in the DSE environment.

The goal of the instantiation phase is to generate an instantiated statement that can be converted into a region constraint which can be conjuncted with the path condition of the DSE. Listing 3 shows the final form of an instantiated statement just before translating it to a region constraint. We call this form of instantiated statement *linearized* since it has no branching structure in it anymore (compare it with Listing 1 and Listing 2). Using the linearized instantiated statement, Java Ranger generates region constraint and conjuncts it with the path condition of the DSE. Java Ranger also re-directs the DSE to execute the instruction that follows the recovered if-statement. This constitutes an exit point for Java Ranger. An *Exit Point* is a program location at which JR transfers execution back to DSE. For example, after generating a region constraint from Listing 3 that represents the merged multi-path region inside the dashed box in Figure 1, the execution of DSE proceeds from the instruction that is following *BB*45 in Figure 2. Generally, there are three kinds of exit points: a program location that corresponds to the conditional branch's immediate post-dominator, a program location that performs a non-local jump in the form of a `return` instruction, and a set of program locations that Java Ranger does not merge and requires DSE exploration. We refer to these three exit points as a non-exceptional and non-returning exit point (NENR), a returning exit point (RE), and single-path exit point (SP) respectively in the rest of this work.

2 RELATED WORK

Path explosion hinders scalable use of symbolic execution, so an appealing direction for optimization is to combine the representations of similar execution paths, which we refer to as *path merging*. If a symbolic execution tool maintains objects representing multiple execution states, a natural approach is to merge these states, especially ones with the same control-flow location. Hansen et al. [16] and Kuznetsov et al. [20] are representative examples of this approach. A similar example can be found in the large-block encoding

```
if (!(x52!=-1 )) then { skip; }
else {
   x54 := invoke(List.get(I)Object, x9, x66);
   x55 := checkcast(Integer, x54);
   x57 := invoke(Integer.intValue()I, x55);
   if (x57 != 0 ) then {
      if (!(x67 != 0 )) then {
         x39[x64] := x66;
         x58 := (x64 + 1 );
      } else { skip; }
   } else { skip; }}
x59:= γ
      (x52!=-1,γ(x57!=0,γ(!(x67!=0),x58,x64),x64),x64);
x61:= γ(x52!=-1,γ(x57!=0,γ(!(x67!=0),1,x67),0),x67);
x62 := γ
      (x52!=-1,γ(x57!=0,γ(!(x67!=0),x68,x68),x68),1);
```

Listing 1: Static Statement

```
if (!(a1!=-1 )) then { skip; }
else {
   x54 := invoke(List.get(I)Object, 375,
         0);
   x55 := checkcast(Integer, x54);
   x57 := invoke(Integer.intValue()I, x55);
   if (x57 != 0 ) then {
      if (!(0 != 0 )) then {
         399[0] := 0;
         x58 := (0 + 1 );
      } else { skip; }
   } else { skip; }}
x59:=γ(a1!=-1,γ(x57!=0,γ(!(0!=0),x58,0),0),0);
x61:=γ(a1!=-1,γ(x57!=0,γ(!(0!=0),1,0),0),0);
x62:=γ(a1!=-1,γ(x57!=0,γ(!(0!=0),0,0),0),1);
```

Listing 2: Instantiated Statement

```
r399[0]_5 := γ(a1!=0, 0, -1);
r399[0]_8 := γ(a1!=-1, r399[0]_5, -1);
x59_1 := γ(a1!=-1, γ(a1!=0, 1, 0), 0);
x61_1 := γ(a1!=-1, γ(a1!=0, 1, 0), 0);
x62_1 := γ(a1!=-1, 0, 1);
```

Listing 3: Linearized Instantiated Statement

Figure 3: Java Ranger's High Level Overview. Blue variables correspond to inputs to the region. Red variables correspond to outputs from the region. `x52, x9, x66, x67, x39,` **and** `x64` **refer to the input of** `element, list, i, inWord, wrdStartIndArr,` **and** `wordCount` **respectively. Similarly,** `r399[0]_8, x59, x61,` **and** `x62` **refer to the outputs of** `wrdStartIndArr[0], wordCount, inWord,` **and** `done` **respectively.**

approach [6] by Beyer et al. for model checking C code. Sen et al.'s MultiSE [27] achieves similar benefits for symbolic execution as part of a different tool architecture.

Another approach to achieve path merging is to statically summarize regions that contain branching control flow. This approach was proposed by Avgerinos et al. [3] and dubbed "veritesting." A veritesting-style technique is a convenient way to add path merging to a symbolic execution system that maintains only one execution state, like SPF. Avgerinos et al. implemented their veritesting system MergePoint to apply binary-level symbolic execution for bug finding. They found that veritesting provided a dramatic performance improvement, allowing their system to find more bugs and have better coverage.

The way that Java Ranger and similar tools statically convert code regions into formulas is similar to techniques used in verification. In the limit where all relevant code in a program can be summarized, such as with WBS and TCAS in Section 4, Java Ranger performs similarly to a bounded symbolic model checker for Java. SPF and Java Ranger build on Java Pathfinder (JPF) [36], which is widely used for explicit-state model checking of Java. The most closely related Java model checking tool is JBMC [11], which shares infrastructure with the C tool CBMC. JBMC performs symbolic bounded model checking of Java code, transforming code and a supported subset of the standard library into SMT or SAT formulas that represent all possible execution paths. The process by which JBMC transforms its internal code representation into SMT formulas is similar to how Java Ranger constructs static regions. However, the dynamic dispatch aspects of Java make creating entirely static representations expensive. We believe that our approach can yield simpler SMT formulas in many cases where it is difficult to completely statically summarize program behavior, and can be used in cases when software is too large and/or complex to be explored completely.

Many other enhancements to symbolic execution have been proposed to improve its performance, including caching and simplifying constraints, summarizing repetitive behavior in loops, heuristic guidance towards interesting code, pruning paths that are repetitive or unproductive, and many domain-specific ideas. A recent survey by Baldoni et. al. [5] provides pointers into the large literature. One approach that is most related to our multi-path regions that have method invocation, is the function-level compositional approach called SMART proposed by Godefroid [14]. SMART differs in being based on single-path symbolic execution instead of static analysis, and targeting C it does not address dynamic dispatch.

3 TECHNIQUE

Conceptually Java Ranger has two main phases, a *static phase*, and a *instantiation phase*. The static phase consists of two transformations and the instantiation phase consists of nine transformations. Note that Java Ranger's default configuration chooses to perform the transformations of both phases dynamically during execution. More concretely the static phase is performed on-the-fly when needed rather than prior to symbolic execution to avoid the expense of creating a static statement for unexecuted methods, classes, and packages. The distinction then between the two phases refers to the dependency of the transformation on dynamic or static information rather than when they are executed. The transformations are discussed in more detail in Sections 3.2 - 3.4, but are summarized below.

The two transformations of the static phase are explained as follows (more details in Section 3.2):

-*IR Statement Recovery:* To more easily implement the Java Ranger transformations, we first convert the control flow graph representation into an internal representation (IR) in the *IR Statement recovery transformation.*

-*Early-Returns Summarization*: Multi-path and method regions can sometimes have more than one return-instruction. DSE needs to explore each return possibility in a separate execution path which can increase the total number of execution paths explored by DSE. In this transformation, Java Ranger collapses return paths into a single return path that can explore all return possibilities.

$\langle val\ v \rangle ::= C \mid Z \mid B \qquad \langle var\ x \rangle ::= .. \mid Id_s \mid Id_{sr}$

$\langle ref\ r \rangle ::= Id \qquad \langle field \rangle ::= Id \qquad \langle class\ c \rangle ::= Id$

$\langle sig\ g \rangle ::= ... \qquad \langle exp\ e \rangle ::= ... \mid \gamma(e_1, e_2, e_3)$

$$\begin{aligned}\langle stmt\ s \rangle ::= &\ x := e \mid s_1\ \mathbf{;}\ s_2 \mid \mathbf{skip} \mid \mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 \\ &\mid x := \mathbf{invoke}(g, r, \overrightarrow{e}) \mid \mathbf{putfield}(r, f, e) \mid \mathbf{getfield}(r, f, e) \\ &\mid \mathbf{return}\ e \mid x := r[e] \mid r[e_1] := e_2 \mid \mathbf{new}\ (c, \overrightarrow{e}) \mid \mathbf{throw}\ e\end{aligned}$$

Figure 4: Main Constructs in Ranger IR

Briefly, the nine instantiation phase transformations are as follows:

-*Alpha-Renaming*: Name clashes of symbolic variables on the *PC* can result in unsound behaviour. In this transformation, Java Ranger avoids this problem by using unique names for newly encountered symbolic variables. For example in listing 3 variables are appended with _1 to distinguish their names.

-*Input Substitution*: Java Ranger needs to bind inputs of the static statement to their runtime values to preserve soundness. It does this by collecting and substituting runtime information from DSE into the input of the static statement.

-*Method Region Inlining* and *Field and Array Reference GSA Creation*: These transformations support method invocation as well as field and array accesses. The former inlines method regions and the latter represents field and array accesses in a Gated-Single-Assignment(GSA) form [23, 34].

-*Simplification*: This transformation pushes concrete values to subsequent variables definitions using constant propagation, copy propagation, and constant folding [2].

-*Single-Path Cases*: Java Ranger avoids merging certain types of statements (object creation and exceptions) because of limitations that Java Ranger has inherited. If Java Ranger had to abort all code regions that contains these two types of statements because it is unable to merge them, then it will miss many useful path-merging instances. This transformation is Java Ranger's way of partially merging paths, while directing SPF to execute unmerged ones.

-*Linearization and Green*: Even after removing all Java-specific statements above, a potential Java Ranger's statement is still not reduced enough to a solver constraint. These two transformations remove if-statement structures and generate a region constraint.

In the rest of this section we introduce JR's IR grammar, discuss JR's static phase, present Java Ranger's instantiation algorithm, and finally discuss Java Ranger's instantiation time transformations.

3.1 Java Ranger IR Grammar

Figure 4 shows the grammar of the IR for JR. Values are characters, integers, and booleans. In addition to program variables and symbolic variables Id_s, JR uses Id_{sr} to capture return-symbolic variables. Java Ranger IR identifies references ref, fields $field$, classes $class$, and method signatures sig.

Java Ranger extends usual expressions with a γ-expression (seen in Listing 1): $\gamma(e_1, e_2, e_2)$, which describes the GSA form, where e_1 is the condition and e_2 and e_3 are the returned expressions if the condition was evaluated to true or false.

Statements include assignment, composition, **skip**, **if**, method invocation: x:= **invoke**(g,r,$\overrightarrow{e}$) , return statement: **return** e, put field statement: **putfield**(r, f, e) to put the value of expression e in the field f of reference r, get field statement: x := **getfield**(r, f), array load: x := r$[e]$ and array store: r$[e_1]$:= e_2, object creation: **new**(c,$\overrightarrow{e}$) where c is the class type and $\overrightarrow{e}$ is the constructor parameters, and finally **throw** statement to represent an exception being thrown in the JR IR.

3.2 Static Phase

The goal of the static phase is to identify multi-path regions that can be summarized and to create an intermediate representation (IR) for the region while collapsing return-statements. Listing 1 shows the output of the static phase. There are 2 main transformations in this phase:

1) IR Statement Recovery

In this transformation, multi-path regions and method regions are identified by first recovering their corresponding CFG using Wala. Next, the CFG is converted to the Java Ranger's *intermediate representation (IR).* Java Ranger distinguishes two types of regions. (1) *Multi-Path Region*: this corresponds to the Java bytecode of an acyclic subgraph of the control-flow graph (CFG). It begins from the basic block containing a conditional branch and ends at this basic block's immediate post-dominator. A node *d* immediately post dominates a node *n* if every path from the *n* node to exit node must go through *d* and *d* does not strictly dominate any other node that strictly dominates *n*. For example, the immediate post-dominator of BB40 is BB45, while the immediate post-dominator of BB43 is BB44.

Since Java allows if-statements to be nested, a multi-path region may also contain other multi-path regions. For example, the sub-graphs from BB30-BB45, and BB35-BB45 in Figure 2 are both recognized as two multi-path regions. The reason for recognizing both multi-path regions is that, while it may not be possible to successfully path merge the larger multi-path region, the inner region could be merged. (2) *Method Region*: this corresponds to Java bytecode that spans the definition of a method. For example, the larger CFG (not shown), that contains the CFG in Figure 2, including method entry and exit blocks constitutes a method region. The recovered Java Ranger IR captures the multi-path or method region in GSA form.

The algorithms of the static statement recovery are similar to those used for decompilation [39]. Starting from an initial basic block in a control-flow graph, the algorithm first finds the immediate post-dominator of all *normal* control paths, that is, paths that do not end in an exception or return instruction. It then looks for nested self-contained subgraphs.

If, for any subgraph, the post-dominator is also a predecessor of the node, we consider it a loop and discard the region. For example, in Figure 1, the subgraph of the CFG that spans the entire while-loop from line 9-22, is discarded. On the other hand, the subgraph of the CFG that corresponds to the enclosed if-statement that spans lines 11-21 is going to be created, because the immediate post-dominator of BB30 is BB40 and it is not its predecessor (Figure 2).

During this process, *local inputs*, where runtime-values are going to be used, and *outputs*, where the result of path merging is going

to be written to, of the static statement are identified. In particular, given a JR statement, the first *use* of a stack slot is deemed a local input and the last *def* of a stack slot is deemed a local output.
For example, consider the region in the dashed box in Figure 1 with static statement shown in Listing 1. Here `x64` and `x9`, in Listing 1, correspond to the input of `wordCount`, and `list`. Similarly `x59`, `x61`, and `x62` corresponds to the output of `wordCount`, `inWord`, and `done` respectively.
- **Create Gated Single Assignment(GSA)**: Part of the statement recovery is the creation of the GSA from a Static-Single-Assignment (SSA) form. Thus in this transformation Java Ranger changes ϕ-statements, that join local variable updates from different branches, to an assignment with the right-hand-side expression being a γ-expression. The γ-expression have an extra parameter (than the ϕ-expression) that describes the condition of the matching if-statement. For example, the right-hand side of the assignment to `x59` (output of `wordCount`) in Listing 1 is the GSA form for the first ϕ-expression shown in BB45 in Figure 2. The GSA captures the conditions as well as possible assignments for `x59`.
2) Early-Return Summarization
Code regions can be exited due to function calls, exceptions, or return-statements. In this transformation, we factor out the predicate that describes paths leading to a return-statement, i.e., RE exit point. This is done by creating and maintaining two expressions, *conditional-return-expressions* and *return path constraint (RPC)*. The conditional-return-expression contains all possible expressions enclosed in return-statements, predicated by their relative path condition in the static statement.

For example, consider a simple if-statement: $if(x1 > 1)\ return\ x2\ else\ return\ x3;$. Here $x2$ and $x3$ needs to be captured in the conditional-return-expression. Thus Java Ranger creates a new symbolic-return-variable $x_r \in Id_{sr}$, and assigns it to γ-expression that describes the return-expressions, i.e., $x_r = \gamma(x1 > 1, x2, x3)$. The *RPC* on the other hand contains the disjunctive conditions that describe all return-conditions. For the example above, the *RPC* will be $(x1 > 1 \,\|\, x1 <= 1)$ which can be reduced to *true* to indicate that an early-return must occur in this simple static statement on both sides of the branch.

3.3 Java Ranger Instantiation Algorithm

The goal of the instantiation process, described in Algorithm 1, is to use runtime values to convert a static statement to a linearized instantiated statement and finally to a region constraint.

The algorithm starts when DSE is about to execute a multi-path region that begins with a conditional branch instruction with a symbolic operand(s) for which JR has a static statement s.

First, JR creates its initial environment ω_o from the current DSE state δ_{sym}. JR state includes: JR's local variables map and the static statement's local inputs. JR state also includes a map for creating GSA form for writes to objects referenced in the static statement. This map is used to merge writes to fields of objects and contents of arrays.

Java Ranger ensures the uniqueness of variable in s by running alpha-renaming transformation (line 3). Then, Java Ranger runs transformations from the lines (6-14) repeatedly until a fixpoint is reached. These transformations perform different operations such as substitution and method inlining.

Algorithm 1: JR Static Statement Instantiation Algorithm

1 Input: (Ranger IR Statement s, DSE δ_{sym});
2 ω_o = construct-initial-state(s, δ_{sym});
3 $(\omega_\alpha, s_\alpha)$ = α-renaming(ω_o, s);
4 $(\omega_{bf}, s_{bf}) = (\omega_\alpha, s_\alpha)$;
5 ω_{af} = *null*; s_{af} = null;
6 **repeat**
7   **if** $(\omega_{af}, s_{af}) \neq null$ **then** $\omega_{bf}, s_{bf} = (\omega_{af}, s_{af})$;
8   s_{sub} = substitute local inputs(ω_{bf}, s_{bf});
9   (ω_{hg}, s_{hg}) = inline method regions(ω_{bf}, s_{sub});
10   (ω_f, s_f) = create ref. GSA(ω_{hg}, s_{hg});
11   (ω_{ar}, s_{ar}) = create arr. GSA(ω_f, s_f);
12     $(\omega_{smpl}, s_{smpl})$ = simplify (ω_{ar}, s_{ar});
13     $(\omega_{af}, s_{af}) = (\omega_{smpl}, s_{smpl})$;
14 **until** $(\omega_{bf}, s_{bf}) = (\omega_{af}, s_{af})$
15 (ω_{sp}, s_{sp}) = collect single-path cases(ω_{af}, s_{af})
16 s_{ln} = linearize(s_{sp});
17 **if** *is-linearized*(ω_{sp}, s_{ln}) **then**
18   e = generate constraint(ω_{sp}, s_{ln})
19     $PC = PC \wedge e$;
20   populate outputs (ω_{sp});
21   pc = address of first inst. after s;
22 **else** abort;   `/* resume DSE from cond. branch */`

Note that there is no particular order to run some of the transformations. For example, a statement may have a method invocation on an object which itself is the result of a prior field access. Inlining a method's static statement requires knowing the runtime type of the object to which it is bound, which can only be obtained after running the *F*ield References GSA transformation. But, once the method's static statement has been inlined, it may include another method invocation on an object which is the result of another field access. Therefore, JR runs the transformations in lines 6-14 until a fixpoint, in which the post-state region is unchanged from the pre-state region, is reached.

Then Java Ranger factors out paths that involve object creation and exceptions in line 14. These *single-path cases* must be performed by the DSE due to architectural limitations in SPF/JPF. In line 15 Java Ranger runs the linearization transformation. If the resulting statement s_{sp} is *linearized*, that is s_{sp} is a composition of assignment statements with no branching, then Java Ranger creates the solver constraint in line 18, and adds it to the path condition (*PC*). Then Java Ranger populates the instantiated statement's outputs (`r399[0]_8, x59_1, x61_1`, and `x62_1` in Figure 3) to the stack or the heap, where `r399[0]` is the output of the first element of `wrdStartIndArr`. Java Ranger also sets the program counter (*pc*) to the address of the bytecode instruction that occurs as the first instruction in the immediate post-dominator. In our example, this is the address of the instruction that performs the `addition` operation in *BB*45 in Figure 2. These operations are shown on lines 19-21 of Algorithm 1.

Line 21 describes the case when the immediate post-dominator is the NENR exit point of the multi-path region. Similarly, at the SP and/or the RE exit point(s), Java Ranger transfers the control back to DSE if these exit points are feasible in the instantiated statement. The exploration of SP and RE exit points is done by creating exploration branching in the DSE (not shown in the algorithm). If a fully-linearized form of the instantiated statement cannot be produced, JR aborts and allows DSE to resume execution (line 22).

3.4 Instantiation-Time Transformations

In this section, we explain only seven transformations of the instantiation phase. We elide the discussion of the two transformations, the alpha-renaming and the simplification transformations, as they were previously explained in Section 3.

1) Input Substitution: In this transformation, Java Ranger collects the runtime values for *inputs* (predetermined in the static phase) and substitutes them. For example, in Listing 1, `x52, x9, x66, x67, x39,` and `x64` are identified as inputs for `element, list, i, inWord, wrdStartIndArr,` and `wordCount` respectively, and are substituted by DSE's runtime values (Listing 2).

2) Method Region Inlining: This transformation inlines the static statement for a method invocation. It is, in general, impossible to know statically which method to inline, and thus JR uses instantiation-time values and type information to figure out (if possible) the method that is about to be invoked. This is possible if JR is able to find out which concrete reference is used in the invocation. For recursive functions, JR inlines methods up to a user-specified parametric depth.

The effects of this transformation on our motivating example are:

```
if (!(a1 != -1 )) then { skip; }
else { x4_3 = getfield(375, size);
       if (!(0 < x4_3)) then throw else skip;
       x4_4 = getfield(375, elementData);
       x5_4 = x4_4[0]; ... }
x59 := γ(a1!=-1, γ(x57!=0, γ(!(0!=0), x58, 0), 0), 0);
x61 := γ(a1!=-1 γ(x57!=0, γ(!(0!=0), 1, 0), 0), 0);
x62 := γ(a1!=-1 γ(x57!=0, γ(!(0!=0), 0, 0), 0), 1);
```

The above listing shows the result of inlining $ArrayList.get(I)Object$ in the static statement of the multi-path region in Figure 1. To do that, $x9$ in the recovered static statement in Listing 1 is substituted by 375, the runtime concrete value of the reference `list`. Then, using the concrete type of 375 allows Java Ranger to inline $375.get(I)Object$ shown above.

3) Field References GSA Creation: This transformation abstracts reference updates and lookups by capturing their semantics through fresh local GSA variables that describe the computation. Note that since local variables obtained from the CFG are already in an Static-Single-Assignment form, Java Ranger needs only a simpler transformation (the Gated Single Assignment transformation) to create their equivalent GSA form. This is unfortunately not the case for references, and thus this transformation is used to create the GSA for reference updates and lookups.

At the NENR exit point of the instantiated statement, field assignments to the same field are merged using γ-expression.

In the motivating example, the field references transformation on the above statement concretizes $x4_3$. In this case, to determine the value of variable $x4_3$, the transformation looks up reference number 375 in SPF's representation of the heap, extracts the field 'size' and determines that it is bound to the concrete value 7.

4) Array References GSA Creation: Similar to the field GSA transformation, this transformation translates array accesses to symbolic variables that reflect the array computations. It maintains a path-specific copy of every array when it is first accessed using a concrete array reference within an instantiated statement. Reads and writes of arrays are then performed on a path-specific copy of the array. All array copies are merged at the NENR of the instantiated statement. The merged array copy represents the array's outputs of the instantiated statement. Out-of-bounds array accesses are explored as a SP exit point. The effect of this transformation on the last shown statement introduces the fresh symbolic variables `r399[0]_8, r399[1]_9,` and `r399[2]_10` $\in Id_s$. These three symbolic variables describe possible assignments to elements of `wrdStartIndArr` with a γ condition[1].

```
if (!(a1 != -1)) then { skip; }
else {...}
r399[2]_8 := γ(a1!=-1, r399[1]_5, -1);
r399[2]_9 := γ(a1!=-1, r399[1]_6, -1);
r399[2]_10 := γ(a1!=-1, r399[2]_7, -1);
x59 := γ(a1!=-1, γ(x57!=0, γ(!(0!=0), x58, 0), 0), 0);
x61 := γ(a1!=-1 γ(x57!=0, γ(!(0!=0), 1, 0), 0), 0);
x62 := γ(a1!=-1 γ(x57!=0, γ(!(0!=0), 0, 0), 0), 1);
```

5) Single-Path Cases Generation: Summarizing object creation while path-merging requires maintaining a symbolic heap. Such language features cannot be summarized and must be executed using SPF because of the way SPF is architected and integrated into JPF. In this phase, we build a guard predicate which avoids paths that contain object creation and exceptions. We call these paths single-path cases, and use SPF to execute them. The outcome of this process is: (a) an JR statement that captures non-SP exit point behavior in the instantiated statement and (b) a predicate that is used to explore the SP exit point behaviour in the instantiated statement.

In fact, the bug in our motivating example, Figure 1, is found when Java Ranger directs the DSE to explore the SP exit point using the SP predicate for the out-of-bounds array access. The added predicate for directing DSE to this particular path is: `(x61_1 == 0 ) && (!((x59_1 < 3 ) && ( x59_1 >= 0 )))`, where `x59` and `x61` are the output of the `wordCount` and `inWord`.

6) Linearization: At the point in Algorithm 1 when this transformation is run, all Java features other than if-statements, composition and assignments statements have been removed from within the instantiated statement. This transformation then prepares the instantiated statement by replacing if-statements with a composition of its "then" and "else" statements. This is correct as long as the conditions of the eliminated if-statements are captured within γ-expressions. Running this transformation, after a fixpoint has been reached, produces the linearized instantiated statement in Listing 3.

7) Constraint Generation: This transformation translates the fully linearized statement to region constraint in Green [35]. This

[1]Definitions of `r399[0]_5, r399[1]_6,` and `r399[2]_7` elided for space reasons.

is done by translating statement composition into conjunction, assignments as equality constraints with assignments of γ-expression being translated as a disjunctive equality expressions.

4 EVALUATION

4.1 Implementation

We implemented Java Ranger as an extension of Symbolic PathFinder [24]. We used the existing *listener* framework in SPF that invokes a callback function for each bytecode instruction executed by SPF. JR adds a listener to SPF that, on every symbolic branch, attempts path merging as described in Algorithm 1. JR uses the incremental solving mode of the Z3 theorem prover [12] with the bitvector theory. The incremental solving mode significantly reduces the number of times a constraint has to be passed to the solver. JR uses a heuristic to estimate the number of paths through a linearized statement. The linearized instantiated statement is used only if the estimated number of paths in the linearized instantiated statement is greater than the number of exit points in it. This heuristic avoids use of path-merging when it may not have been beneficial. Our implementation of Java Ranger is publicly available [30].

4.2 Experimental Setup

Table 1: Benchmark programs used to evaluate Java Ranger

Bench mark name	Description	SLOC	# classes	# methods
WBS	component to make aircraft brake safely	265	1	3
TCAS	maintain aircraft altitude separation	300	1	12
replace	search & replace pattern in input	795	1	19
Nano XML	XML Parser	4610	17	129
Siena	event notification middleware	1256	10	94
Schedule	priority scheduler	306	4	27
Print Tokens2	lexical analyzer	570	4	30
ApacheCLI	command-line parser	3612	18	183
Mer Arbiter	flight software comp. of NASA JPL Mars Exploration Rovers	4697	268	553

We sought answers to the following research questions.

RQ1: *Does Java Ranger reduce the number of execution paths and running time in a program when exploring all feasible behaviors?*

RQ2: *Does Java Ranger reduce the time required for checking for the absence of runtime errors in a program?*

RQ3: *How much does each Java Ranger feature contribute to performance?*

RQ4: *How does Java Ranger compare to other state-of-the-art Java verifiers?*

We present the experimental setup used to answer each of these research questions and the corresponding evaluation below.

RQ1: We evaluated the performance of Java Ranger using the nine benchmarking programs presented in Table 1. We obtained the first eight from the evaluation set used by Wang et al. [37] and the last one (MerArbiter) from Yang et al. [40]. We used SPF as the Dynamic Symbolic Executor (DSE) for comparing it with Java Ranger. Path merging is useful in symbolic execution when it is used for checking a property on all feasible behaviors or finding all bugs in a program. We compared JR with SPF when exploring all feasible paths through each benchmark. All benchmarks were single-threaded execution, we leave exploration of static regions in multi-threaded programs to future work. We used a wall time budget of 12 hours for every benchmark. We ran every benchmark with the most number of symbolic inputs with which an exploration of all feasible behaviors could be completed in a wall time budget of 12 hours. The maximum heap size was limited for all the benchmark runs with JR and SPF to 8 GB. We ran all of our experiments on a machine running Ubuntu 16.04.6 LTS with Intel(R) Xeon(R) CPU E5-2623 v3 processor and 192 GB RAM. We report our results in Table 2.

JR achieves a total reduction of 38% and 71% reduction in total running time and number of execution paths respectively across all benchmarks. It also achieves an average of 63% reduction in the number of solver queries. It is able to significantly reduce the total number of execution paths on every benchmark where it finds beneficial use of a static statement for a multi-path region.

JR achieves a significant speed-up over SPF with 5 (WBS, TCAS, NanoXML, ApacheCLI, MerArbiter) of the 9 benchmarks in running time and number of execution paths. It also achieves a modest 21% and 40% reduction in running time and number of execution paths respectively with the PrintTokens2 benchmark.

JR reduces the number of execution paths by about 88% in replace but incurs an increase in execution time by 187%. The 67.3% reduction in the number of solver queries causes a 240% increase in solver time spent by JR. In the future, we plan to mitigate such negative effects of path-merging by integrating JR with a query count estimation heuristic [20].

While not instantiating any statement, JR incurs a 2.6% running time overhead on Siena. This primarily results from JR's checking if a conditional branch has symbolic operands and lookup of a static statement for every symbolic branch. The total running time of Schedule with SPF and JR is very small (1.5 and 2.5 seconds respectively) compared to other benchmarks. JR's static analysis always adds about 2-3 seconds to the total time: this accounts for loading the WALA framework, constructing a class hierarchy for all classes in the classpath, and building the CFG for all methods in Wala IR. On benchmarks like Schedule with a small total running time, this overhead from static analysis is a higher percentage of the total running time.

RQ2: Since path-merging brings symbolic execution closer to symbolic bounded model-checking, we also used these benchmarks to compare JR with JBMC [11]. JBMC is a Java model checker that verifies programs by unwinding loops and looks for runtime exceptions. We ran each of our benchmarks with JBMC with the same number of non-deterministic inputs reported in the "# sym inputs" column of Table 2. We configured JBMC to unwind loops in each

Table 2: Comparing execution time and path count between JR and SPF

Bench mark name	#sym input	tool	total time (sec)	% red. in time	static analysis time (sec)	# exec. paths	%red. in # exec. paths	% red. in # queries	# summ. used
WBS	15	SPF	4427.7	99.9	0.0	7.96E+06	100	100.00	-
	30	JR	4.2		2.3	1.00E+00			140
TCAS	24	SPF	353.1	99.1	0.0	3.92E+04	100	100.00	-
	120	JR	3.1		1.7	1.00E+00			40
replace	11	SPF	1145.3	-187.0	0.0	7.57E+05	88.1	67.30	-
	11	JR	3287.6		5.9	9.04E+04			6502
Nano XML	7	SPF	5741.4	46.2	0.0	3.61E+06	84.6	81.00	-
	7	JR	3087.1		3.1	5.54E+05			147185
Siena	6	SPF	5571.9	-2.6	0.0	2.99E+06	0	0.00	-
	6	JR	5715.6		7.3	2.99E+06			0
Schedule	3	SPF	1.5	-70.3	0.0	3.43E+02	0	0.00	-
	3	JR	2.5		3.4	3.43E+02			0
Print Tokens2	5	SPF	17045.8	21.3	0.0	3.06E+06	40.4	38.70	-
	5	JR	13421.3		25.1	1.82E+06			1981982
Apache CLI	5+1	SPF	4121.4	45.7	0.0	2.48E+05	92.9	99.10	-
	5+1	JR	2238.1		5.3	1.76E+04			168907
Mer Arbiter	24	SPF	9494.0	80.3	0.0	2.53E+05	83.9	81.50	-
	24	JR	1873.4		3.6	4.08E+04			59845
Summary	-	SPF	47901.9	38.13	0	1.89E+07	71	-	-
	-	JR	29632.8		57.7	5.51E+06			2364601

Table 3: Comparing total running time of SPF, Java Ranger, JBMC over the 9 benchmarks for verifying the absence of common runtime errors. TO indicates timeout, given a 7 day time limit. Times for SPF and JR are reported in seconds.

tool name	WBS	TCAS	replace	Nano XML	Siena	Schedule	Print Tokens2	Apache CLI	Mer Arbiter
SPF	4427	353.1	1145.3	5741.4	5571.9	1.5	17045.8	4121.4	9494
JBMC	0.7	2.2	TO	TO	TO	5.62E+05	TO	TO	TO
Java Ranger	4.2	3.1	3287.6	3087.1	5715.6	2.5	13421.3	2238.1	1873.4

benchmark a given number of times and to add an assertion whose violation indicates that a loop was not unrolled sufficient times. We performed binary search to find the smallest loop bound for each benchmark that would not cause a loop unwinding assertion violation with JBMC. The smallest loop bounds we found with this binary search for every benchmark were as follows: ApacheCLI=39, Siena=8, PrintTokens2=82, replace=12, NanoXML=10, Schedule=10, WBS=11, TCAS=11. For MerArbiter, we could not get JBMC to falsify a loop-unwinding assertion for any positive value of the loop-unwinding parameter. Therefore, we finally ran it with a loop bound of 1. We ran every benchmark presented in Table 1 using SPF, Java Ranger and JBMC where all three tools looked for common runtime errors such as null dereferences, accessing out-of-bounds entries in arrays, type cast errors, and division-by-zero errors. We used a 7 day timeout for all three tools. We ran every benchmark with the same number of symbolic inputs with all three tools. The number of symbolic inputs is reported in the `# sym inputs` column of Table 2.

We present our results in Table 3. The rows labeled `SPF` and `Java Ranger` repeat the time reported in the `total time (sec)` column of Table 2. We found JBMC was the fastest among the three tools at being able to verify the absence of any runtime errors in both WBS and TCAS. But, we found JBMC to be much slower with the remaining 7 benchmarks. JBMC was able to complete verification with Schedule in about 7 days. For the remaining six benchmarks, JBMC did not finish in 7 days as indicated by `TO` in Table 3.

RQ3: JR can be separated into four path-merging features.

(F1) JR only transforms multi-path regions with a single NENR exit point. This includes multi-path regions that have local, stack, field, or array outputs.

(F2) JR inlines statements for methods called from a multi-path region into the statement of the multi-path region.

(F3) JR instantiates static statements with an SP exit point

(F4) JR uses early-return summarization to allow statements to have a RE exit point.

To answer RQ3, we evaluated the effect each feature has as more features are cumulatively used in JR. We set up an experiment

Table 4: Presenting the ratio of three metrics with path-merging to the same three metrics without path-merging for 7 benchmarks where any path-merging was done. A ratio less than 1 indicates path-merging was beneficial (smaller is better). Path-merging features accumulate from left to right.

(a) Comparing running time

Bench mark name	basic p.m.	+method inlining	+single path cases	+early return summ.
WBS	0.0007	0.0007	0.0007	0.0006
TCAS	0.39	0.01	0.01	0.01
replace	1.36	2.10	2.82	2.87
Nano XML	1.31	1.28	1.54	0.54
Print Tokens2	0.88	0.76	0.78	0.79
Apache CLI	0.17	2.54	0.50	0.54
Mer Arbiter	0.24	0.21	0.21	0.20

(b) Comparing number of execution paths

Bench mark name	basic p.m.	+method inlining	+single path cases	+early return summ.
WBS	1.2E-07	1.2E-07	1.2E-07	1.2E-07
TCAS	0.24	2.5E-05	2.5E-05	2.5E-05
replace	0.63	0.90	0.12	0.12
Nano XML	1.00	1.00	1.00	0.15
Print Tokens2	0.84	0.84	0.84	0.60
Apache CLI	0.07	0.07	0.07	0.07
Mer Arbiter	0.16	0.16	0.16	0.16

where starting with no path-merging (aka SPF), we added path-merging features in the aforementioned order (F1-F4). For every benchmark where any path-merging was performed, we computed the ratio of a metric with a set of path-merging features enabled to the same metric's value seen without path-merging. The two metrics we measured were the running time, the number of execution paths explored. We present the results of this comparison in Table 4. The "basic p.m." column represents only enabling of the F1 feature in JR. The "+ method inlining" column enables the F1 and F2 (inlining of method statements) features in JR. The "+ single-path cases" column enables the F1, F2, and F3 (single-path cases) features in JR. The "+ early return summ." column enables all the four features with early-return summarization. Table 4 shows summarizing multi-path regions that have a NENR exit point (F1) is most often useful. This observation matches our intuition that such multi-path regions occur most frequently in Java. The addition of method statement inlining (F2) provides a major reduction in all three metrics in TCAS. This observation matches an observation made manually from TCAS' source code that multi-path regions in it often invoke methods that can be summarized by Java Ranger. The addition of single-path cases (F3) provides a major reduction in the number of solver queries in the replace and NanoXML benchmarks. Early-return summarization (F4) provides a significant reduction in the number of execution paths and number of solver queries in the NanoXML and PrintTokens2 benchmarks. The benefit from this feature results from these benchmarks containing multi-path regions that contain multiple RE exit points. Table 4 shows that every path-merging feature present in JR has a beneficial impact on at least one benchmark in our set.

Table 5: Comparing Java Ranger with participants of the JavaOverall category of SV-COMP 2020

tool	score	# correct true results	# correct false results	# incorrect results
JayHorn [18]	278	109	92	1
SPF [24]	442	135	172	1
COASTAL [13]	472	135	202	0
JDart [21]	524	150	224	0
JBMC [11]	527	151	225	0
Java Ranger	549	173	203	0

RQ4: Java Ranger participated in a static verification competition named SVComp [29] The competition consisted of a Java verification track in which six Java verifiers competed over 416 benchmark programs. These benchmarks spanned regression tests introduced by each of the participating tools. The benchmarks also included implementations of algorithms for commonly used data structures such as tries and red-black trees. The competition's setup placed a total memory limit of 15 GB and a limit of 8 CPU cores. The wall time limit for running each benchmark in the competition was 15 minutes. We report the results from our participation as well as scores of all competition participants in the Java track in Table 5.

Java Ranger was the best performing tool in the Java verification track in the competition [38]. Of the total 416 Java verification tasks that were used in the competition, Java Ranger instantiated at least one static statement on 96 different benchmarks. The static statement for a multi-path region can be instantiated more than once on each benchmark because it is possible for the symbolic executor to encounter the same multi-path region more than once while running the benchmark. In total, Java Ranger instantiated 356 distinct static statements with the total number of instantiated statements being 20,182. Java Ranger also inlined a method statement a total of 62,857 times while instantiating static statements.

Java Ranger finished with a "unknown" result on 40 of the 416 verification tasks used in the competition. 22 of these were caused due to a lack of support for symbolic strings. Java Ranger defaults to vanilla SPF when it finds no opportunity for path-merging. On these 22 benchmarks, SPF's lack of stable symbolic string support caused a crash. Similarly, 9 of the 40 "unknown" results occurred due to missing support for symbolic array lengths in multi-dimensional

arrays in SPF. 8 of the 40 "unknown" results ran into a timeout. The last "unknown" result was caused due to our limiting of the depth of exploration choices for the competition.

5 DISCUSSION & FUTURE WORK

Java Ranger attempts to perform path merging whenever possible without optimizing towards making fewer solver calls. We plan to work towards implementing heuristics that can measure the effect of path merging on the rest of the program. JR currently lacks support for symbolic object and array references. Supporting these would require integrating our implementation with SPF's lazy initialization [24] to let summaries contain symbolic object references.

Generating test cases that cover all branches is a useful application of dynamic symbolic execution. If applied as-is, test generation will undo the benefits of path-merging. We intend to extend JR towards test generation for merged paths in the future by targeting test generation towards a coverage criterion such as Modified Condition/Decision Coverage.

Path merging allows symbolic execution to explore interesting parts of a program sooner. But, the effect of path merging on search strategies, such as depth-first search remains to be investigated. We plan to explore the integration of such guidance heuristics with path merging in the future. Finally, we plan to expand on our formalism to prove completeness as well as soundness.

6 CONCLUSION

We have investigated the use of static summarizations to improve the performance of symbolic execution of Java. For good performance, we had to extend earlier work to account for Java's dynamic dispatch and likelihood of exceptions. Our experiments demonstrate that static summarization may yield significant performance improvements over single-path symbolic execution. Java Ranger provides evidence that inlining method summarizations by using type information available at runtime can lead to a further reduction in the number of execution paths. Java Ranger's use of path-merging is crucial to giving it an edge over existing Java verifiers as demonstrated by its participation in a static verification competition in a top theory conference. Java Ranger reinterprets and extends path merging for symbolic execution of Java bytecode and may allow symbolic execution to scale to exploration of real-world Java programs.

ACKNOWLEDGMENT

The research described in this paper has been supported in part by the Google Summer of Code program and by the National Science Foundation under grant 1563920.

REFERENCES

[1] 2014. *ISSTA 2014: Proceedings of the 2014 International Symposium on Software Testing and Analysis.* Association for Computing Machinery, New York, NY, USA.

[2] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 2007. *Compilers: principles, techniques, and tools.* Vol. 2. Addison-wesley Reading, MA, USA.

[3] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. 2014. Enhancing Symbolic Execution with Veritesting. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014).* ACM, New York, NY, USA, 1083–1094. https://doi.org/10.1145/2568225.2568293

[4] Domagoj Babić, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. 2011. Statically-Directed Dynamic Automated Test Generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11).* Association for Computing Machinery, New York, NY, USA, 12–22. https://doi.org/10.1145/2001420.2001423

[5] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3, Article Article 50 (May 2018), 39 pages. https://doi.org/10.1145/3182657

[6] Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M. Keremoglu, and Roberto Sebastiani. 2009. Software model checking via large-block encoding. *9th International Conference Formal Methods in Computer Aided Design, FMCAD 2009*, 25 – 32. https://doi.org/10.1109/FMCAD.2009.5351147

[7] J. Burnim and K. Sen. 2008. Heuristics for Scalable Dynamic Test Generation. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering.* IEEE, New York, NY, USA, 443–446. https://doi.org/10.1109/ASE.2008.69

[8] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08).* USENIX Association, USA, 209–224.

[9] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2012. The S2E Platform: Design, Implementation, and Applications. *ACM Trans. Comput. Syst.* 30, 1 (2012), 2:1–2:49. http://doi.acm.org/10.1145/2110356.2110358

[10] Lori A. Clarke. 1976. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Trans. Software Eng.* 2, 3 (1976), 215–222. https://doi.org/10.1109/TSE.1976.233817

[11] Lucas Cordeiro, Pascal Kesseli, Daniel Kroening, Peter Schrammel, and Marek Trtik. 2018. JBMC: A Bounded Model Checking Tool for Verifying Java Bytecode. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 183–190.

[12] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[13] Jaco Geldenhuys, Justin Stigling, and Willem Visser. 2020. COASTAL: Concolic analysis tool for Java. https://github.com/DeepseaPlatform/coastal. (2020).

[14] Patrice Godefroid. 2007. Compositional Dynamic Test Generation. *SIGPLAN Not.* 42, 1 (Jan. 2007), 47–54. https://doi.org/10.1145/1190215.1190226

[15] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05).* ACM, New York, NY, USA, 213–223. https://doi.org/10.1145/1065010.1065036

[16] Trevor Hansen, Peter Schachte, and Harald Søndergaard. 2009. State Joining and Splitting for the Symbolic Execution of Binaries. In *Runtime Verification*, Saddek Bensalem and Doron A. Peled (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 76–92.

[17] IBM. 2006–2020. WALA. http://wala.sourceforge.net/wiki/index.php/Main_Page. (2006–2020). Accessed: 2018-11-16.

[18] Temesghen Kahsai, Philipp Rümmer, and Martin Schäf. 2019. JayHorn: A Java Model Checker. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen (Eds.). Springer International Publishing, Cham, 214–218.

[19] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394. http://doi.acm.org/10.1145/360248.360252

[20] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient State Merging in Symbolic Execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12).* ACM, New York, NY, USA, 193–204.

[21] Kasper Luckow, Marko Dimjašević, Dimitra Giannakopoulou, Falk Howar, Malte Isberner, Temesghen Kahsai, Zvonimir Rakamarić, and Vishwanath Raman. 2016. JDart: A Dynamic Symbolic Analysis Framework. In *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (Lecture Notes in Computer Science)*, Marsha Chechik and Jean-François Raskin (Eds.), Vol. 9636. Springer, New York, NY, USA, 442–459.

[22] T. Nguyen, M. B. Dwyer, and W. Visser. 2017. SymInfer: Inferring program invariants using symbolic states. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE).* 804–814. https://doi.org/10.1109/ASE.2017.8115691

[23] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. MacCabe. 1990. The Program Dependence Web: A Representation Supporting Control-, Data-, and Demand-driven Interpretation of Imperative Languages. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI '90).* ACM, New York, NY, USA, 257–271. https://doi.org/10.1145/93542.93578

[24] Corina S. Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, and Neha Rungta. 2013. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. *Automated Software Engineering* 20, 3 (01 Sep 2013), 391–425. https://doi.org/10.1007/s10515-013-0122-2

[25] David A Ramos and Dawson R. Engler. 2011. Practical, Low-effort Equivalence Verification of Real Code. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 669–685. http://dl.acm.org/citation.cfm?id=2032305.2032360

[26] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*. ACM, New York, NY, USA, 263–272. https://doi.org/10.1145/1081706.1081750

[27] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. 2015. MultiSE: Multi-path Symbolic Execution Using Value Summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 842–853. https://doi.org/10.1145/2786805.2786830

[28] V. Sharma, K. Hietala, and S. McCamant. 2018. Finding Substitutable Binary Code for Reverse Engineering by Synthesizing Adapters. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, Los Alamitos, CA, USA, 150–160. https://doi.org/10.1109/ICST.2018.00024

[29] Vaibhav Sharma, Soha Hussein, Michael W. Whalen, Stephen McCamant, and Willem Visser. 2020. Java Ranger at SV-COMP 2020 (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems*, Armin Biere and David Parker (Eds.). Springer International Publishing, Cham, 393–397.

[30] Vaibhav Sharma, Soha Hussein, Michael W. Whalen, Stephen McCamant, and Willem Visser. 2020. java-ranger: v1.0.0. (Jun 2020). https://doi.org/10.5281/zenodo.3907232

[31] Yan Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, New York, NY, USA, 138–157. https://doi.org/10.1109/SP.2016.17

[32] Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, San Diego, CA, 1–16.

[33] Wei Sun, Lisong Xu, and Sebastian Elbaum. 2017. Improving the Cost-effectiveness of Symbolic Testing Techniques for Transport Protocol Implementations Under Packet Dynamics. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. ACM, New York, NY, USA, 79–89. https://doi.org/10.1145/3092703.3092706

[34] Peng Tu and David Padua. 1995. Efficient Building and Placing of Gating Functions. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI '95)*. Association for Computing Machinery, New York, NY, USA, 47–55. https://doi.org/10.1145/207110.207115

[35] Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. 2012. Green: Reducing, Reusing and Recycling Constraints in Program Analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. ACM, New York, NY, USA, Article 58, 11 pages. https://doi.org/10.1145/2393596.2393665

[36] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. 2003. Model Checking Programs. *Automated Software Engineering* 10, 2 (01 Apr 2003), 203–232. https://doi.org/10.1023/A:1022920129859

[37] H. Wang, T. Liu, X. Guan, C. Shen, Q. Zheng, and Z. Yang. 2017. Dependence Guided Symbolic Execution. *IEEE Transactions on Software Engineering* 43, 3 (March 2017), 252–271. https://doi.org/10.1109/TSE.2016.2584063

[38] Phillipp Wendler. 2020. SV-COMP 2020 – JavaOverall – BenchExec Results. https://sv-comp.sosy-lab.org/2020/results/results-verified/META_JavaOverall.table.html. (2020).

[39] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. 2015. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantics-Preserving Transformations. In *The 2015 Network and Distributed System Security Symposium*. The Internet Society, Reston, VA, USA. https://doi.org/10.14722/ndss.2015.23185

[40] Guowei Yang, Corina S. Păsăreanu, and Sarfraz Khurshid. 2012. Memoized Symbolic Execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA 2012)*. ACM, New York, NY, USA, 144–154. https://doi.org/10.1145/2338965.2336771

View publication stats