Java Ranger: Static Regions for Efficient Symbolic Execution of Java

Anonymous Author(s)*

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 [10, 14], 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 [11, 20], equivalence checking [19, 22], vulnerability finding [23, 24], and protocol correctness checking [25]. Symbolic execution tools are available for many languages, including CREST [6] for C source code, KLEE [7] for C/C++ via LLVM, JDart [16] and Symbolic PathFinder (SPF) [18] for Java, and S2E [9], FuzzBALL [4], and angr [23] for binary code.

Although symbolic analysis is a popular technique, scalability is a substantial challenge for many applications. In particular it can suffer from a *path explosion*: complex software has exponentially many execution paths, and baseline symbolic execution techniques that explore one path at a time are unable to cover all paths. Dynamic state merging [12, 15] provides one way to alleviate scalability challenges by opportunistically merging dynamic symbolic executors, effectively merging the paths they represent. 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 recently proposed technique that can dramatically improve the performance of symbolic execution by effectively merging paths. Rather than explicitly merging state representations, veritesting encodes a local region of a program containing branches as a disjunctive region for symbolic analysis. This often allows many paths to be collapsed into a single path involving the region. In previous work [3], constructing bounded static code regions was shown to allow symbolic execution 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 using static regions for symbolic execution of Java software (at the Java bytecode level).

Java programmers who follow best software engineering practices attempt to write code in an object-oriented form with common functionality implemented as a Java class and multiple not-too-large methods used to implement small sub-units of functionality. This causes Java programs to make several calls to methods, such as getters and setters, to re-use small common sub-units of functionality. Merging paths within regions in such Java programs using techniques described in current literature is limited by not having the ability to inline method summaries. This is not a major impediment for compiled C code, as the C compiler will usually automatically inline the code for short methods such as get. However, Java has an *open world* assumption, and most methods are *dynamically dispatched*, meaning that the code to be run is not certain until a method is resolved at runtime; if inlining is performed at all, it is by the JRE, so it is not reflected in bytecode.

Not being able to summarize such dynamically dispatch methods can lead to poor performance for naïve implementations of bounded static regions. Thus, to be successful, we must be able to inject the static regions associated with the calls into the dispatching region. We call such regions *higher order* as they require a region as an argument and can return a region that may need to be further interpreted. 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.

Another common feature of Java code that represents the boundary of path merging is *exceptions*. If an exception can potentially be raised in a region, the symbolic executor needs to explore that exceptional behavior. But, it is possible for other unexceptional behavior to also exist in the same region. For example, it can be in the form of a branch nested inside another branch that raises an exception on the other side. Summarizing such unexceptional behavior while simultaneously guiding the symbolic executor towards potential exceptional behavior reduces the branching factor of the region. We propose a technique named *Single-Path Cases* for splitting a region summary into its exceptional and unexceptional parts.

While summarizing higher-order regions and finding single-path cases is useful to improve scalability, representing such summaries in an intermediate representation (IR) that uses static single-assignment (SSA) form provides a few key advantages. (1) It allows region summaries to be constructed by using a sequence of transformations, with each transformation extending to add support for new features such as heap accesses, higher-order regions, and single-path cases. (2) It allows for simplifications such as constant propagation, copy propagation, constant folding to be performed

on region summaries. (3) It makes the construction of region summaries more accessible to users of the symbolic execution tool, thereby making path merging more useful to end-users.

In this paper, we present Java Ranger, an extension of Symbolic PathFinder, that computes such region summaries over a representation we call Ranger IR. Ranger IR has support for inlining method summaries and for constructing SSA form for heap accesses. It also proposes Single-Path Cases as an alternative to multiple transition points as defined by Avgerinos et al. [3].

This paper extends our initial investigations of Java veritesting reported in a paper at the 2017 JPF workshop [?]. Our workshop paper motivated some of the ways in which Java veritesting is different from veritesting for binary code and the value that veritesting could provide, but it did not describe an end-to-end automated implementation. The present paper describes a number of conceptual improvements that are new since the workshop paper, including the Ranger IR, single-path cases, and the details of our higher-order region approach. We have also made a number of architectural changes, such as switching from Soot to Wala for static analysis and using Green for formula representation, and we have integrated Java Ranger as an extension to SPF and evaluated it in several case studies.

1.1 Motivating Example

Consider the example of Java code shown in Figure 1. The example is used to count number of words where characters are represented with 1 and 0 represents space. The `list` object refers to an `ArrayList` of 200 `Integer` objects which have an unconstrained symbolic integer as a field. Checking the first element to be a zero or one introduces a symbolic branch, that requires the symbolic execution to explore both paths. Therefore loop checking for each indexed entry in `list` introduces a branch, which has both sides feasible and which requires the symbolic execution to explore both paths again.

Performing this check over the entire `list` makes symbolic execution need 2^{100} execution paths to terminate (assuming `list` has 200 entries with every even-indexed entry pointing to a new unconstrained symbolic integer). A simple way to avoid this path explosion is to merge the two paths arising out of the `list.get(i) == 0` branch. Such path merging requires us to compute a summary of all behaviors arising on both sides of the branch of the if-statement inside the for loop. If we can construct such a summary beforehand, our symbolic executor can instantiate the summary by reading in inputs to the summary from the stack and/or the heap, and writing outputs of the summary to the stack and/or the heap. Unfortunately, constructing such a summary for this simple region is not straightforward due to the call to `list.get(int)` which is actually a call to `ArrayList<Integer>.get(int)` (`java.util.List<E>.get(int)` is abstract and does not have an implementation). `ArrayList<Integer>.get(int)` internally does the following:

(1) 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.

(2) 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.

(3) It calls `Integer.intValue()` on the object returned by the previous step. This call internally accesses the `value` field of `Integer` to return the integer value of this object.

The static summary of `ArrayList<Integer>.get(int)` needs to not only include summaries of all these three methods but also include the possibility of an exception being raised by the included summary of `ArrayList<E>.rangeCheck(i)`. Our extension to path-merging includes using method summaries, either with a single return or no return, as part of region summaries that have method calls [1]. The method whose summary is to be included depends on the dynamic type of the object reference on which the method is being invoked. In our example, the dynamic type of `list` is `ArrayList`, whereas it is declared statically as having the type `List`. Therefore, the summary of `list.get(i)` pulls in the method summary of `ArrayList<E>.get(i)`.

Our *Single-Path Cases* extension to path-merging also allows the possibility of exceptional behavior being included in the summary and explored separately from unexceptional behavior by performing exploration of exceptional behavior in the region on its own execution path.

We will use this example throughout the reset of the paper to show how different transformations changes the region of interest until a summarization of the region is obtained.

2 RELATED WORK

Path explosion is a major cause of scalability limitations for symbolic execution, so an appealing direction for optimization is to combine the representations of similar execution paths, which we refer to generically as *path merging*. If a symbolic execution tool already concurrently 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. [12] explored this technique but found mixed results on its benefits. Kuznetsov et al. [15] developed new algorithms and heuristics to control when to perform such state merging profitably. A larger departure in the architecture of symbolic execution systems is the MultiSE approach proposed by Sen et al. [21], which represents values including the program counter with a two-level guarded structure, in which the guard expressions are optimized with BDDs. The MultiSE approach achieves effects similar to state merging automatically, and provides some architectural advantages such as in representing values that are not supported by the SMT solver.

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" because it pushes symbolic execution further along a continuum towards all-paths techniques used in verification. A veritesting-style technique is a convenient way to add path merging to a symbolic execution system that maintains only one execution state, which is one reason

---

[1]We plan to support methods with multiple returns in the future.

```java
//ArrayList of symbolic Integers
ArrayList<Integer> list =
        new ArrayList<>( initialCapacity: 200);
if (list.size() > 0) {
    int firstElement = list.get(0);
    inWord = firstElement != 0;
    for (int i = 0; i < list.size(); i++){
        if (inWord) {
            //list.get(i) returns
            // a symbolic integer
            if (list.get(i) == 0) {
                ++wordCount;
                inWord = false;
            }
        } else {
            if (list.get(i) != 0) {
                inWord = true;
            }
        }
    }
}
System.out.println("Number of words is:"
        + wordCount);
```
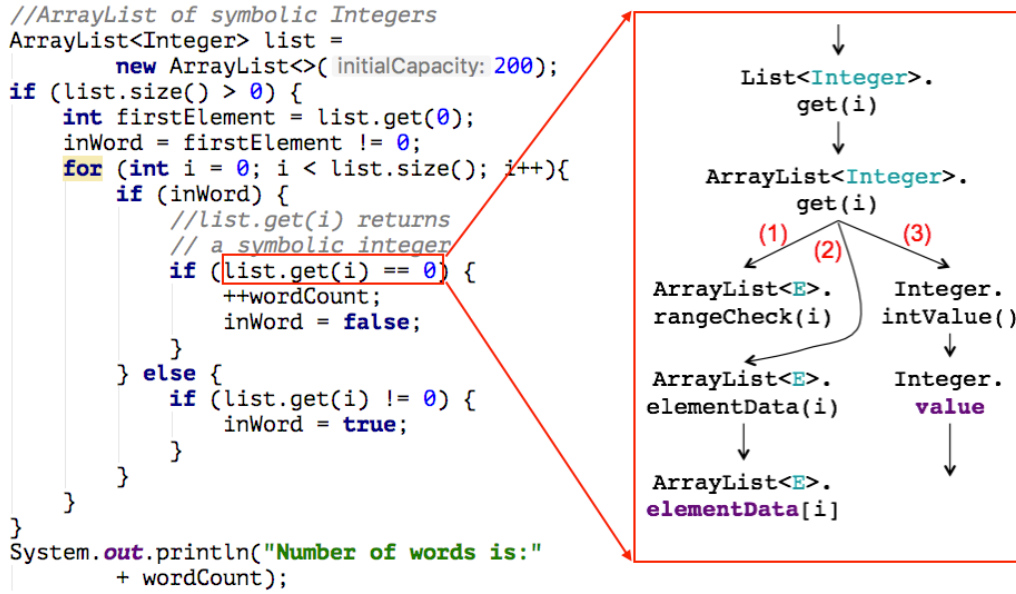
Figure 1: An example demonstrating the need for using a multi-path region summary

we chose it when extending SPF. Avgerinos et al. designed and implemented their veritesting system MergePoint for the application of 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 node and path coverage in a given exploration time. The static regions used by MergePoint are intra-procedural, but they can have any number of "transition points" at which control can be returned to regular symbolic execution. Avgerinos et al. do not provide details about how MergePoint represents memory accesses or integrates them with veritesting, though since MergePoint was built as an extension of the same authors' Mayhem system, it may reuse techniques such as symbolic representation of loads from bounded memory regions [8].

The veritesting approach has been integrated with another binary level symbolic execution engine named angr [23]. However angr's authors found that their veritesting implementation did not provide an overall improvement over their dynamic symbolic execution baseline: though veritesting allowed some new crashes to be found, they observed that giving more complex symbolic expressions slowed down the SMT solver enough that total performance was degraded. We have also observed complex expressions to be a potential cost of veritesting, but we believe that optimizations of the SMT solver interface and potentially heuristics to choose when to use static regions can allow them to be a net asset.

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-SR in Section 4, Java Ranger performs similarly to a bounded symbolic model checker for Java. SPF and Java Ranger build on Java Pathfinder (JPF) [?], which is widely used for explicit-state model checking of Java and

provides core infrastructure for instrumentation and state backtracking. Another family of Java analysis tools that use formula translation (also called verification condition generation) are ESC/-Java [?], ESC/Java2 [?], and OpenJML [?], though these tools target static error checking and verification of annotated specifications.

Perhaps the most closely related Java model checking tool is JBMC [?], which has recently been built sharing infrastructure with the similar 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 sometimes described as (static) symbolic execution, but it has more in common with how Java Ranger constructs static regions than with the symbolic execution that vanilla SPF performs.) In cases that Java Ranger can completely summarize, we would expect its performance to be comparable to JBMC's; an experimental comparison is future work. But static region summaries are more important as an optimization to speed up symbolic execution on software that is too large and/or complex to be explored exhaustively.

A wide variety of 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. [?] provides pointers into the large literature. One approach that is most related to our higher-order static regions is the function-level compositional approach called SMART proposed by Godefroid [?]. Like Java Ranger's function summaries, SMART summarizes the behavior of a function in isolation from its calling context so that the summary can be

reused at points where the function is used. But SMART uses single-path symbolic execution to compute its summaries, whereas Java Ranger uses static analysis: this makes Java Ranger's summary more compact at the expense of requiring more reasoning by the SMT solver. Because SMART was implemented for C, it does not address dynamic dispatch between multiple call targets.

3 TECHNIQUE

Input: Region R;
somethingChanged = true;
execute early-return transformation(R);
execute alpha-renaming transformation(R);
repeat
    while (*somethingChanged*) **do**
        execute substitution transformation(R);
        execute high-order transformation(R);
        execute fields transformation(R);
        execute arrays transformation(R);
        execute references alpha-renaming transformation(R);
        execute simplification transformation(R);
    end
    execute high-order transformation(R);
until !(*somethingChanged*);
execute single-path cases(R);
execute linearlization transformation(R);
execute to-green transformation(R);
if *R successfully transformed* **then**
    populate output;
else
    abort;
end
populate output;
        Algorithm 1: Ranger general pesudo-code

To add path merging to SPF, we first pre-compute static summaries of arbitrary code regions with more than one execution path and we also pre-compute method summaries.

Figure **??** describes the main steps that Java Ranger deploys to summarize the region. Intuitively, Java Ranger starts with a symbolic multi-path region expressed in Ranger grammar 3 (described in section 3.1). Java Ranger then proceeds as follows, first it performs early return transformation, by eliminating eliminating return statements, this is followed by alpha-renaming on variables in the region. Then Java Ranger enters two fix-point loops, the inner loop terminates when no further changes is detected on the region after repeatedly applying substitution, high-order, fields, arrays, reference alpha-renaming and simplification transformations. can be made on the region. The outer loop ensures that at least one step of high order region was attempt without further changes before it declares reaching a fixed point.

After reaching a fixed point, single path cases are determined, then flattening the IR of the region to a sequence of assignment statements followed by a transformation that transforms the region into the corresponding solver syntax using Green. Finally if all

transformations were successful then Java Ranger populates the output to SPF and advances it to the right position, otherwise it aborts.

In the following sections, we describe each of the transformations performed by Java Ranger both formally by defining the semantics of each transformation and informally by showing how each transformation is realized on our example 1

3.1 Statement Recovery

3.1.1 Statement Recovery Setup. To bound the set of code regions we analyze, we start by specifying a method M in a configuration file. Next, we construct a set containing only the class C that contains M. We then get another set of classes, C', such that every class in C' has at least one method that was called by a method in a class in C. This step which goes from C to C' discovers all the classes at a call depth of 1 from C. We continue this method discovery process up to a call depth of 2. While we can increase the call depth in our method discovery process, we found that summarizing arbitrary code regions with more than 2 calls deep, did not lead to practically useful region summaries. After obtaining a list of methods, we computed static summaries of regions in these methods and method summaries as explained in Section 3.1. After computing static region and method summaries, we process them as a sequence of transformations described in the next section 3.3 and summarized in Figure 2.
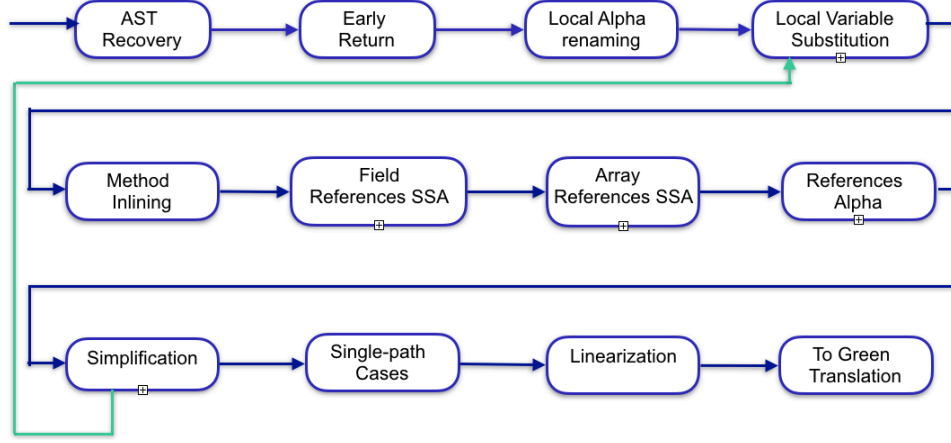
3.1.2 IR-Statement Recovery. In this step, a Java Ranger is created from the corresponding CFG. Since regions of interest for our technique are bounded by the branch and meet of a given acyclic subgraph. The intuition is that path explosion during execution of loops is driven by conditional logic within the loop, rather than the loop itself. Starting from an SSA form, the first transformation recovers a tree-shaped AST for the subgraphs of interest. While this step is not strictly necessary, it substantially simplifies subsequent transformations.

The algorithms are similar to those used for those used for decompilation [28] but with slightly different goals:

- The algorithm must be *accurate* but need not be *complete*. That is, obfuscated regions of code need not be translated into a tree form.
- The algorithm must be *lightweight* in order to be efficiently performed during analysis. Thus, algorithms that use global fixpoint computations are too expensive to be used for our purposes.

Starting from an initial SSA node, 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 graph, the post-dominator is also a predecessor of the node, we consider it a loop and discard the region.

The algorithm systematically attempts to build regions for every branch instruction, even if the branch is already contained within another region. The reason is that it may not be possible to instantiate the larger region depending on whether summaries can be found for *dynamically-dispatched* functions, and whether references are *uniquely determinable* for region outputs.

Figure 2: Overview of transformations on Ranger IR to create and instantiate regions up to a fix-point.

$$\langle stmt \rangle ::= \langle stmt \rangle ; \langle stmt \rangle \mid \langle exp \rangle := \langle exp \rangle \mid \text{skip} \mid x.\langle stmt \rangle$$
$$\mid \langle exit\_stmt \rangle \mid \text{if } \langle exp \rangle \text{ then } \langle stmt \rangle \text{ else } \langle stmt \rangle$$
$$\mid \text{invoke}(\langle exp \rangle, \langle exp \rangle, \langle exp \rangle) \mid \text{exit}$$

$$\langle exit\text{-}stmt \rangle ::= \text{new } \tau \mid \text{return } \langle exp \rangle \mid \text{throw } \langle exp \rangle$$

$$\langle exp \rangle ::= \langle val \rangle \mid \langle var \rangle \mid \langle exp \rangle \ op_b \ \langle exp \rangle \mid op_u \ \langle exp \rangle$$
$$\mid \text{get\_field}(\langle exp \rangle, \langle exp \rangle) \mid \text{put\_field}(\langle exp \rangle, \langle exp \rangle)$$
$$\mid \text{array\_load}(\langle exp \rangle, \langle exp \rangle)$$
$$\mid \text{array\_store}(\langle exp \rangle, \langle exp \rangle, \langle exp \rangle)$$
$$\mid \text{gamma}(\langle exp \rangle, \langle exp \rangle, \langle exp \rangle)$$
$$\mid \langle exp \rangle.\langle exp \rangle$$

$$\langle op_b \rangle ::= + \mid - \mid * \mid \div \mid \& \mid \text{bitwise-or} \mid \oplus \mid \% \mid == \mid \neq \mid \leq \mid \geq \mid \&\& \mid$$
$$\text{logical-or} \mid > \mid < \mid \ll \mid \gg \mid \ggg$$

$$\langle op_u \rangle ::= - \mid \sim$$

$$\langle val \rangle ::= (\ ) \mid \mathbb{Z} \mid \mathbb{C}$$

$$\langle var \rangle ::= \text{ID\_}\mathbb{N}$$

Figure 3: Context Free Grammar for Java Ranger IR

Value-maps	$\Delta_r : loc \rightarrow$	(val, exp)
	$\Delta_s : loc \rightarrow$	(val, exp)
Type-maps	$\Gamma_r : var \rightarrow$	τ
	$\Gamma_s : ref \rightarrow$	τ
Tegion-map	$\Psi : \tau \rightarrow$	$x.s$
	$\Theta : var \rightarrow$	loc
Single-path-constraints	$\Sigma : \{exp\}$	
Early-return-constraints	$\Sigma_{ret} : x_{ret}$	$\{exp\}$

Figure 4: Environments used in Ranger.

The outcome of this step is a statement in Java Ranger grammar 3. Basically the grammar is composed of statements *stmt* and expressions *exp*. Statements in Java Ranger contains: sequential composition (*stmt*; *stmt*), assignment (*exp* = *exp*), a skip, parametric statements *x.stmt*, exit-statements *exit*, which determine single path regions and they are *new τ*, *throw exp* and *return exp*, if-then-else, method invocation *invoke(exp, exp, exp)* and especial *exit* statement to bookmark exit point of a region/statement.

Java Ranger expressions on the other hand are: values (unit ( ) , positive and negative integers $\mathbb{Z}$ and characters $\mathbb{C}$) , variables (these are subscripted with integers to facilitate having ssa form for vars D_$\mathbb{N}$), binary operations op_b, unary operations op_u, $get - field(exp, exp)$ and $put - field(exp, exp, exp)$ for accessing and changing fields, $array - load(exp, exp)$ and $array - store(exp, exp, exp)$ for accessing and changing array elements, a special gamma expression, $gamma(exp, exp, exp)$, that operates like ? : in Java. Gamma expression are fundamentally important to Java Ranger because it is how Java Ranger determines the conditions of different paths as well as the evaluation of variables along these paths. It worth pointing out that Java Ranger defines 3 types of variables, a field or array variable (created to maintain fields and arrays ssa), an early return variable (to carry early return values) and any other type of variables. We assume the existance of a special function $gen_v ar$ that would generate one variable in the right sequence for each of these types.

In the remaining sections we will use x to range over any type of variable, x_{ref} to variables constructed for fields and arrays and x_{ret} for variables constructed for early return construction. Similarly we will use s to range over statements, e to range over expressions and v to range over values.

3.2 Region Definition

Once the statement of a multi-path region has been recovered, its corresponding environment is populated. This includes identifying region boundary and creating local variable inputs, outputs, type, and stack slot tables for the region. The region boundary is used to identify boundaries of the region w.r.t local variables. This is used later to constrain the computation and population of Ranger IR environment tables. For example, the local variable input table is populated with first *use* in the region boundary that map to a given stack slot. The output table is populated with the last *def*

of a local variable at merge point of the region. The local variable type table is populated for all variables that lay within the boundaries of the region, this is initially done by inquiring the static analysis framework, WALA [1] but is later changed by inferring types of local variables at instantiation and during type propagation transformation 3.3.

We also construct a stack slot table as part of a region's Ranger IR summary. The stack slot table maps Ranger IR variables to a stack slot, if they correspond to a local variable in the source code. We populate the stack slot table by obtaining a variable to stack slot mapping from WALA. We also assume that, if at least one variable used in a ϕ-expression is a local variable, then all variables used in that ϕ-expression must belong to the same stack slot. We use this assumption to further propagate stack slot information in the stack slot table across all ϕ-expressions encountered at merge points of regions.

Formally we define the following structures:

value-map: Java Ranger and SPF value map Δ_r and Δ_s that maps location, i.e., stack slots, to concrete values val or symbolic values exp.

type-map: Java Ranger and SPF value map Γ_r and Γ_s that maps vars or references, to types τ.

region-map: Java Ranger defines a map Ψ from type τ to a parametric statement $x.s$ that defines the parameteric statement region in Java Ranger.

Single-Path-Constraint-List: A list Σ of single-path constraints.

Early-Return-Constraint-map: A map Σ_{ret} from early-return vars x_{ret} to early-return constraints.

3.3 Instantiation-time Transformations

Early Return Transformation: In this transformation, Java Ranger collects that conditions for branches that have early return in them. In figure 5 we show the rule of having an early return on the else side $early{-}return_1$ where neither inner statements s_1 or s_2 are conditional with early return. In this case, the statement is transformed to appending an assignment statement to the early return result to a new early return var x_{ret}, also the condition under which the early return value occurs is added to the early return environment $\Sigma_{ret} \cup (x_{ret}, e_2)$

Rule $early{-}return_2$ describes the situation where both the statements $s1$ and $s2$ contains inner conditions with early return statements. In this case a new result variable is created x_{ret} and assigned to the gamma expression $x_{ret} = gamma((c1 \wedge e_1), e'_1, e'_2)$ describing conditions and values assigned under them. In addition the context of early return variables-conditions is updated.

Renaming Transformation: In Alpha renaming transformation, all Ranger IR variables are renamed to ensure their uniqueness before further processing takes place. This is particularly important not only to ensure uniqueness of variables among different regions, but also to ensure uniqueness of variable names of the *same* region which might be instantiated multiple times on the same path, i.e., a region inside a loop will be instantiated multiple times.

We do not define this transformation formally due to space, but it is basically straight forward by generating a unique prefix for each variable and updates the new name in the statement as well as all the contexts in which it appears.

Local Variable Substitution Transformation: During this transformation we eagerly bring in all dynamically known constant values, symbolic values and references from stack slots into the region for further processing.

More formally, rule $substitution$ given a parametric statement $x.s$, and if x was mapped to a location l then its dynamic, i.e., concrete or symbolic (v, e), substitute x in s.

Higher-order Regions Transformation: This transformation is initiated when a method invocation is encountered during local variable substitution. At this point, we perform three steps. (1) the region that corresponds to the called method is retrieved and alpha renaming of Ranger IR variables corresponding to local variables is applied on it. (2) Ranger IR expressions that correspond to the actual parameters are evaluated and used to substitute the formal parameters by repeatedly applying local variable substitution transformation over the method region. (3) When no more higher-order regions can be inlined, the resulting substituted method region is inlined into the outer region.

Formally, rule $high{-}order_1$ describes the inlining process for methods that has no return values, and $high{-}order_2$ describes inlining of methods that have a single return values. Methods of multiple returns, should have been normalized out with the early-return transformation discussed previously. Not the rules describes invoke virtual in Java, however Java Ranger supports as well invoke static and invoke interface.

For $high_order{-}1$ initially after evaluating the reference of the invoke by either finding it concerte value through substitution or obtaining it from the updated type table, the type is used to retrieve the parametric region for the method being invoked. Substitution of the parameters then follows and direct inlining occurs. In rule $high{-}order_2$, inlining of the return value translates to an assignment statement.

Field References SSA form: The field references transformation translates reads and writes of fields in Java bytecode into corresponding Ranger IR statements. In order to translate all field accesses to SSA form, this transformation creates a summary of the semantics represented by the field accesses in the region. This transformation constructs a new field access variable for every field assignment on every path within the region. This new field access variable construction makes use of two monotonically increasing subscripts. It uses a path subscript to distinguish between assignments to the same field on the same execution path. It uses a global subscript to distinguish between assignments to the same field across execution paths. At the merge point of the region, field assignments done on the same field are merged using Gated Single Assignment (GSA) [17]. Each merged field access variable has its own path and global subscripts and represents the output of the region into its field. The path subscript helps us resolve read-after-write operations on the same execution path and find the latest write into a field on an execution path. The global subscript helps us distinguish between field accesses across multiple execution paths.

Array References SSA form: The array references transformation translates reads and writes of arrays in Java bytecode into corresponding Ranger IR statements. In order to translate all array accesses to SSA form, this transformation creates an execution path-specific copy of every array when it is first accessed within a region. Reads and writes of arrays are then done on a path-specific

$$\frac{\Theta(x) = l, \quad \Delta_s(l) = (v, e)}{\Theta, \Delta_s, x.s \mapsto_{sub} \Theta, \Delta_s, [(v,e)/x]s} \text{ substitution} \qquad \frac{}{, throw\ e \mapsto_{single}, exit} \text{ single-path}_1$$

$$\frac{\Gamma_r, \Delta_r, e_1 \mapsto_{sub} \Gamma_r, \Delta'_r, v_1 \quad \Gamma_s(v_1) = \tau, \quad \Gamma'_r, \Psi(\tau) = x.s_2, \quad \Delta'_r, e_2 \mapsto_{sub} \Delta''_r, v_2}{\Gamma_r, \Delta_r, s_1; invoke(e_1, e_2) \mapsto_{high} (\Gamma_r \cup \Gamma'_r), \Delta''_r, s_1; [v_2/x]s_1} \text{ high-order}_1$$

$$\frac{\Gamma_r, \Delta_r, e_1 \mapsto_{sub} \Gamma'_r, \Delta'_r, v, \quad \Gamma_s(v) = \tau, \quad \Gamma'_r, \Psi(\tau) = x.(s; return\ e'), \quad \Delta'_r, e_2 \mapsto_{sub} \Delta''_r, v_2}{\Gamma_r, \Delta_r, e = invoke(e_1, e_2) \mapsto_{high} (\Gamma_r \cup \Gamma'_r), \Delta''_r, [v_2/x]s; e = e'} \text{ high-order}_2$$

$$\frac{}{\Sigma, if\ e\ then\ (s_1; exit; s'_1)\ else\ s_2 \mapsto (\Sigma \vee e), s_2} \text{ single-path}_2 \qquad \frac{}{\Sigma, if\ e\ then\ s_1\ else\ (s_2; exit; s'_2) \mapsto (\Sigma \vee !e), s_1} \text{ single-path}_3$$

$$\frac{}{\Sigma, if\ e\ then\ (s_1; exit; s'_1)\ else\ s_2; exit; s'_2 \mapsto (\Sigma \vee true), skip} \text{ single-path}_4 \qquad \frac{}{\Sigma, if\ e\ then\ s_1\ else\ s_2 \mapsto \Sigma, s_1; s_2} \text{ linearlization}$$

$$\frac{}{\Sigma, x := gamma(e_1, e_2, e_3) \mapsto \Sigma, (e_1 \wedge x = e_2) \vee (!e_1 \wedge x = e_3)} \text{ to-green}$$

$$\frac{gen\_id(ret) = x_{ret} \quad x''_{ret} \notin s_1 \quad x'_{ret} \notin s_2}{\Sigma_{ret}, if\ e_1\ then\ s_1\ else\ (s_2; return\ e_2) \mapsto \Sigma_{ret} \cup (x_{ret}, e_2), (if\ e_1\ then\ (s_1; return\ e_2)\ else\ s_2); x_{ret} := e_2} \text{ early-return}_1$$

$$\frac{\Sigma, s_1 \mapsto \Sigma_{ret} : (w1_{ret}, c_1), s'_1; w1_{ret} := e''_1; return\ e'_1 \quad \Sigma, s_2 \mapsto \Sigma_{ret} : (w2_{ret}, c_2), s'_2; (w2_{ret} := e''_2; return\ e'_2) \quad gen\_id(ret) = x_{ret}}{\Sigma_{ret}, if\ e_1\ then\ s_1\ else\ s_2 \mapsto \Sigma_{ret} \cup (x_{ret}(e_1 \wedge c_1) \vee (e_1 \wedge c_2)), (if\ e_1\ then\ (s_1; return\ e_2)\ else\ s_2); x_{ret} = gamma((c1 \wedge e_1), e'_1, e'_2)} \text{ early-return}_2$$

Figure 5: Evaluation Rules for Ranger Transformations

copy of the array. All array copies are merged at the merge point of multi-path regions. The merged array copy represents array outputs of the region.

Type Propagation: Ranger IR needs to have type information for its variables so that it can construct corresponding correctly-typed Green variables during the final transformation of the region summary to a Green formula. Having accurate type information is also important for looking up the correct higher-order method summary. As part of region instantiation, Java Ranger infers types of Ranger IR variables in the region summary by using JPF's runtime environment. Types of local variables are inferred during the local variable substitution transformation and types of field reference and array reference variables are inferred during their respective transformations. Using these inferred types, the type propagation transformation propagates type information across assignment statements, binary operations, and variables at leaf nodes of γ functions.

Simplification of Ranger IR: The Ranger IR constructed by earlier transformations computes exact semantics of all possible behaviors in the region. Representation of such semantics as a formula can often lead to unnecessarily large formulas, which has the potential to reduce the benefits seen from path merging [23]. For example, if an entry in an array is never written to inside a region, the array reference transformation can still have an array output for that entry that writes a new symbolic variable into it. The region summary would then need to have an additional constraint that makes the new symbolic variable equal the original value in that array entry. Such conjuncts in the region summary can be easily eliminated with constant propagation, copy propagation, and constant folding [2]. Ranger IR also has statement and expression classes that use a predicate for choosing between two statements (similar to an `if` statement in Java) and two sub-expressions (similar to the C ternary operator) respectively. When both choices are syntactically equal, the predicated statement and expression objects can be substituted with the statement or expression on one of their two choices. Such

statements and expressions were simplified away to use one of their two choices. Ranger IR performs these two simplifications on such predicated statements and expressions along with constant folding, constant propagation, and copy propagation.

Single Path Cases: This transformation collects path predicates inside a region that lead to *non-nominal* exit point. This is an alternative approach to that was presented in [3]. In our work we define non-nominal exit point to be points inside the region that either define exceptional behavior or involve behavior that we cannot summarize, i.e, object creation and throw instructions. The intuition here is that, we want to maximize regions that Java Ranger can summarize, even if the summarization is only partial. We use this pass of transformation to identify such points, collect their path predicates and prune them away from the Ranger IR statement. The formal rules of this transformation is defined in rule single-path$_1$ and single-path$_2$, where the former transforms statements that are considered exit points, then the later constructs the matching constraints to explore the single path of interest. The outcome of this process, is a more simplified and concise statement that represent the nominal behavior of the Ranger region. The collected predicate is later used to guide the symbolic execution to explore non-nominal paths, which Java Ranger had not summarized.

Linearization: Ranger IR contains translation of branches in the Java bytecode to if-then-else statements defined in the Ranger IR. But the if-then-else statement structure needs to be kept only as long as we have more GSA expressions to be introduced in the Ranger IR. Once all GSA expressions have been computed, the Ranger IR need not have if-then-else statements anymore. The γ functions introduced by GSA are a functional representation of branching, which lets us capture the semantics of everything happening on both sides of the branch. Since the linearization transformation linearlization is done after every field and array entry has been unaliased and converted to GSA, dropping if-then-else statements from the Ranger IR representation of the region summary reduces redundancy in its semantics and converts it into a stream

of GSA and SSA statements.

Translation to Green: At this point Ranger region contains only compositional statements as well as assignment statements that might contain GSA expressions in them. This transformation starts off by translating Ranger variables to Green variables of the right type using the region type table. Then Ranger statements are translated. More precisely, compositional statements are translated into conjunction, assignment statements are translated into Green equality expressions. For assignment statements that have GSA expressions to-green, these are translated into two disjunctive formulas that describes the assignment if the GSA condition or its negation were satisfied.

3.4 Checking Correctness Of Region Summaries

The Ranger IR computed as a result of performing the transformations described in Figure 2 should correctly represent the semantics of the summarized region. If it does not, then using the instantiated region summary can cause symbolic exploration to explore the wrong behavior of the subject program. We checked the correctness of our instantiated region summaries by using equivalence-checking as defined by Ramos et al. [19]. We designed a test harness that first executes the subject program with a set of symbolic inputs using SPF and capture the outputs of the subject program. Next, the test harness executes the same subject program with the same set of symbolic inputs using Java Ranger and capture the outputs of the subject program once again. Finally, the test harness compares outputs returned by symbolic execution with SPF and Java Ranger. If the outputs do not match, then a region summary used by Java Ranger did not contain all the semantics of the region it summarized. We symbolically execute all execution paths through this test harness. If no mismatch is found between outputs on any execution path, we conclude that all region summaries used by Java Ranger must correctly represent the semantics of the regions they summarized. We performed correctness-checking on all results reported in this paper.

4 EVALUATION

4.1 Experimental Setup

We implemented the above mentioned transformations as a wrapper around the Symbolic PathFinder [18] tool. To make use of region summaries in Symbolic PathFinder, we use an existing feature of SPF named *listener*. A listener is a method defined within SPF that is called for every bytecode instruction executed by SPF. Java Ranger adds a path merging listener to SPF that, on every instruction, checks (1) if the instruction involves checking a symbolic condition, and (2) if Java Ranger has a pre-computed static summary that begins at that instruction's bytecode offset. If both of these conditions are satisfied, Java Ranger instantiates the multi-path region summary corresponding to that bytecode offset by reading inputs from and writing outputs to the stack and the heap. It then conjuncts the instantiated region summary with the path condition and resumes symbolic execution at the bytecode offset of the end of the region. Our implementation, named Java Ranger, wraps around SPF and can be configured to run in four modes. (1) In mode 1, it runs vanilla SPF without any path merging enabled. (2) In mode

2, it summarizes multi-path regions only and instantiates them if they are encountered. (3) In mode 3, it summarizes and instantiates methods, by utilizing high ordder regions, along with multi-path regions as done in mode 2. (4) In mode 4, it uses single-path cases along with multi-path region and method summaries used in mode 3.

4.2 Evaluation

In order to evaluate the performance of Java Ranger, we used the following benchmarking programs commonly used to evaluate symbolic execution performance. (1) Wheel Brake System (WBS) [29] is a synchronous reactive component developed to make aircraft brake safely when taxing, landing, and during a rejected take-off. (2) Traffic Collision Avoidance System (TCAS) is part of a suite of programs commonly referred to as the Siemens suite [13]. TCAS is a system that maintains altitude separation between aircraft to avoid mid-air collisions. (3) Replace is another program that's part of the Siemens suite. Replace searches for a pattern in a given input and replaces it with another input string. We used the translation of the Siemens suite to Java as made available by Wang et al. [27]. We also manually created a variant of TCAS by converting regions of code with return statements on every execution path to regions of code with a single return statement at the merge point of the region. We refer to this variant of TCAS as TCAS-SR (TCAS with Single Returns). We also created variants of WBS, TCAS, and TCAS-SR by running them for multiple steps.

We ran WBS, TCAS, TCAS-SR, and Replace using Java Ranger and present our results from instantiating region summaries in Table 1, 2. Table 1, 2 shows that Java Ranger achieves a significant improvement using path-merging with WBS. The "# exec paths" column is the number of execution paths required to completely explore the benchmark. "runtime (msec)" is the time taken for dynamic analysis (excludes static analysis time). "# inst. regs" is the total number of regions that were instantiated at least once when running each benchmark. "# inst. methods" is the number of higher-order regions that were used and "inst." is the total number of instantiations that was done when running the benchmark. Table 1, 2 shows that we achieve a significant improvement using path-merging with WBS. Java Ranger outperforms SPF when running TCAS too but it doesn't allow path merging to summarize the entire program to a single execution path, as in the case with WBS. We manually analyzed TCAS and found the only barrier to our path-merging was the presence of several code regions containing a return instruction on every execution path. We manually performed a semantics-preserving transformation to the source code of TCAS to create another version, TCAS-SR, that has a single return value at the end of all such multi-path regions. We plan to automate this transformation in the future, see future work section 5. Table 1, 2 shows the benefit of such a *multiple-returns-to-single-return* transformation. It allows Java Ranger to summarize the entire TCAS-SR program into a single execution path. This is made possible by the use of higher-order regions as shown in Table 1, 2. In mode 2, Java Ranger explores 18 execution paths with TCAS-SR, whereas in mode 3 (mode 2 + higher-order region support), it needs to only explore a single execution path because it can instantiate 28 higher-order method summaries. The bottom half of Table 1, 2

Benchmark Name	# sym inputs	mode	run time (msec)	static ana. time (msec)	#exec. paths	#solver queries	solver time (msec)	#inst. regions	#inst. methods	#inst.
WBS-1step	3	1	2.6E+02	0.0E+00	2.4E+01	4.6E+01	2.1E+01	0.0E+00	0.0E+00	0.0E+00
WBS-1step	3	2	5.7E+02	1.0E+03	1.0E+00	0.0E+00	0.0E+00	6.0E+00	0.0E+00	6.0E+00
WBS-1step	3	3	5.3E+02	1.2E+03	1.0E+00	0.0E+00	0.0E+00	6.0E+00	0.0E+00	6.0E+00
WBS-1step	3	4	9.2E+02	1.3E+03	1.0E+00	1.2E+01	3.3E+02	6.0E+00	0.0E+00	6.0E+00
WBS-1step	3	5	9.2E+02	1.4E+03	1.0E+00	1.2E+01	3.1E+02	6.0E+00	0.0E+00	6.0E+00
WBS-5steps	15	1	4.4E+06	0.0E+00	8.0E+06	1.6E+07	3.1E+06	0.0E+00	0.0E+00	0.0E+00
WBS-10steps	30	5	5.8E+04	2.2E+03	1.0E+00	2.8E+02	5.4E+04	1.6E+01	0.0E+00	1.4E+02
TCAS-1step	12	1	2.7E+03	0.0E+00	2.0E+02	2.3E+03	1.5E+03	0.0E+00	0.0E+00	0.0E+00
TCAS-1step	12	2	3.6E+03	3.0E+03	9.8E+01	1.1E+03	8.6E+02	1.3E+01	0.0E+00	3.6E+02
TCAS-1step	12	3	1.3E+03	4.5E+03	1.0E+00	0.0E+00	0.0E+00	4.0E+00	2.8E+01	4.0E+00
TCAS-1step	12	4	1.7E+03	4.5E+03	1.0E+00	8.0E+00	7.5E+01	4.0E+00	2.8E+01	4.0E+00
TCAS-1step	12	5	1.4E+03	3.4E+03	1.0E+00	8.0E+00	1.1E+02	4.0E+00	2.8E+01	4.0E+00
TCAS-2steps	24	1	3.5E+05	0.0E+00	3.9E+04	4.7E+05	3.0E+05	0.0E+00	0.0E+00	0.0E+00
TCAS-10steps	120	5	3.9E+03	1.8E+03	1.0E+00	8.0E+01	2.2E+03	4.0E+00	2.8E+02	4.0E+01
replace	11	1	1.2E+06	0.0E+00	7.6E+05	4.3E+06	8.2E+05	0.0E+00	0.0E+00	0.0E+00
replace	11	2	1.1E+06	3.4E+03	4.8E+05	3.1E+06	6.7E+05	2.7E+01	0.0E+00	5.1E+04
replace	11	3	1.6E+06	4.0E+03	6.8E+05	4.6E+06	1.0E+06	2.5E+01	1.7E+04	5.4E+04
replace	11	4	7.0E+06	4.1E+03	8.2E+04	3.8E+06	5.3E+06	3.4E+01	6.3E+03	1.4E+05
replace	11	5	8.5E+06	4.4E+03	8.2E+04	3.9E+06	6.3E+06	4.0E+01	6.3E+03	2.9E+05
NanoXML	7	1	4.4E+06	0.0E+00	3.6E+06	8.3E+06	2.2E+06	0.0E+00	0.0E+00	0.0E+00
NanoXML	7	2	5.0E+06	6.8E+02	3.6E+06	8.3E+06	2.4E+06	2.0E+00	0.0E+00	5.3E+03
NanoXML	7	3	5.0E+06	8.7E+02	3.6E+06	8.3E+06	2.3E+06	2.0E+00	0.0E+00	5.3E+03
NanoXML	7	4	1.8E+07	1.1E+03	3.6E+06	9.3E+06	1.2E+07	7.0E+00	7.7E+04	4.9E+05
NanoXML	7	5	2.7E+06	1.4E+03	5.5E+05	1.7E+06	1.6E+06	8.0E+00	3.2E+03	2.0E+05
NanoXML	8	5	2.6E+07	6.8E+02	3.6E+06	1.1E+07	1.8E+07	8.0E+00	2.7E+04	1.4E+06

Table 1: Java Ranger Performance on WBS, TCAS, Replace, and NanoXML

also shows the benefits of path-merging with Java Ranger increase as we run more steps of WBS, TCAS, and TCAS-SR. We ran as many steps of each of these 3 benchmarks as possible with a 6 hour timeout using Java Ranger in modes 1 (runs Java Ranger without path merging) and 3 (runs Java Ranger with multi-path region summaries and higher-order region summaries). While, in mode 1, Java Ranger can only run 5 steps of WBS in less than 6 hours, in mode 4 (path merging with multi-path regions, higher-order regions, and single-path cases), Java Ranger can easily execute up to 10 steps in less than 4 seconds. A similar benefit is seen with TCAS-SR where Java Ranger in mode 1 can only run 2 steps of TCAS-SR in 6 hours,

but in mode 4, it can easily run up to 10 steps of TCAS-SR in about 1 second.

However, the performance of Java Ranger when running Replace is quite different from that seen when running WBS, TCAS, and TCAS-SR. While Java Ranger reduces the total number of execution paths with every increase in mode, the path reduction isn't as significant. More importantly, it also makes an order of magnitude more solver calls thereby increasing the total runtime.

We further investigated this drop in performance of Java Ranger when running Replace by exploring the space of potential regions to instantiate. We obtained the list of regions that are instantiated at least once when running Replace with Java Ranger in mode 3 (using

Final Results Table - 2

Benchmark Name	# sym inputs	mode	run time (msec)	static ana. time (msec)	#exec. paths	#solver queries	solver time (msec)	#inst. reg.	#inst. methods	#inst.
PrintTokens2	3	1	4.2E+04	0.0E+00	9.1E+03	1.9E+05	2.3E+04	0.0E+00	0.0E+00	0.0E+00
PrintTokens2	3	2	4.8E+04	8.0E+02	8.2E+03	1.8E+05	2.3E+04	2.0E+00	0.0E+00	2.3E+03
PrintTokens2	3	3	5.2E+04	1.2E+03	8.2E+03	1.8E+05	2.6E+04	2.0E+00	0.0E+00	2.3E+03
PrintTokens2	3	4	5.0E+04	1.2E+03	8.2E+03	1.9E+05	2.5E+04	2.0E+00	0.0E+00	2.3E+03
PrintTokens2	3	5	7.7E+04	1.1E+03	6.6E+03	1.7E+05	3.4E+04	1.5E+01	0.0E+00	1.8E+04
ApacheCLI	6	1	1.3E+07	0.0E+00	2.5E+06	2.7E+07	4.4E+06	0.0E+00	0.0E+00	0.0E+00
ApacheCLI	6	2	1.7E+06	6.1E+02	9.3E+04	2.1E+05	6.9E+05	4.0E+00	0.0E+00	4.2E+05
ApacheCLI	6	3	1.8E+06	9.1E+02	9.3E+04	2.1E+05	7.4E+05	4.0E+00	0.0E+00	4.2E+05
ApacheCLI	6	4	5.9E+06	9.1E+02	9.3E+04	1.1E+06	4.1E+06	4.0E+00	0.0E+00	4.2E+05
ApacheCLI	6	5	5.0E+06	1.1E+03	9.3E+04	1.1E+06	3.3E+06	5.0E+00	0.0E+00	4.4E+05
ApacheCLI	7	2	1.3E+07	5.5E+02	5.6E+05	1.3E+06	5.5E+06	4.0E+00	0.0E+00	3.6E+06
ApacheCLI	6+1	1	3.2E+07	0.0E+00	4.8E+06	6.4E+07	9.5E+06	0.0E+00	0.0E+00	0.0E+00
ApacheCLI	6+1	2	3.8E+06	5.9E+02	1.7E+05	7.6E+05	1.8E+06	4.0E+00	0.0E+00	8.1E+05
ApacheCLI	6+1	3	3.7E+06	8.0E+02	1.7E+05	7.6E+05	1.7E+06	4.0E+00	0.0E+00	8.1E+05
ApacheCLI	6+1	4	1.0E+07	8.4E+02	1.7E+05	2.4E+06	7.3E+06	4.0E+00	0.0E+00	8.1E+05
ApacheCLI	6+1	5	1.2E+07	7.4E+02	1.7E+05	2.4E+06	8.0E+06	5.0E+00	0.0E+00	8.5E+05
ApacheCLI	7+1	2								
MerArbiter-6steps	24	1	5.5E+06	0.0E+00	7.1E+05	2.7E+05	3.8E+04	0.0E+00	0.0E+00	0.0E+00
MerArbiter-6steps	24	2	1.7E+06	1.2E+04	4.1E+04	1.3E+05	5.1E+04	3.4E+01	0.0E+00	6.7E+04
MerArbiter-6steps	24	3	1.8E+06	2.1E+04	4.1E+04	1.3E+05	4.9E+04	3.4E+01	0.0E+00	6.7E+04
MerArbiter-6steps	24	4	1.9E+06	2.0E+04	4.1E+04	2.6E+05	1.1E+05	3.5E+01	0.0E+00	6.8E+04
MerArbiter-6steps	24	5	2.0E+06	2.1E+04	4.1E+04	2.6E+05	1.1E+05	3.5E+01	0.0E+00	6.8E+04
MerArbiter-7steps	28	2	1.2E+07	1.3E+04	2.8E+05	8.7E+05	4.4E+05	3.4E+01	0.0E+00	4.6E+05

Table 2: Java Ranger Performance on PrintTokens2, ApacheCLI with the last input made made concrete and symbolic respectively, and MerArbiter

high order and multi-path region summaries). This step produced a list of 24 regions. Next, we sampled the space of all possible subsets of this set of 24 regions by (1) randomly enumerating the space up to 11500 subsets of all possible subsets of 24 regions, (2) enumerating the space in the order of set size up to all subsets of 3 regions from a set of 24 regions. With both methods of enumeration, we searched for the least amount of time Java Ranger needs to instantiate at least one region with Replace. We found this least time to be 26.3 seconds which is still more than the 3.47 seconds Java Ranger needs to run Replace in mode 1 (running vanilla SPF with no path merging). In the fastest runs with at least one region instantiation in both methods of enumeration, Java Ranger finished exploration with the same number of execution paths as that explored during mode

1. The result of this analysis, combined with a significant increase in the number of solver queries with Java Ranger's path-merging modes, points toward the conclusion that every instantiation of a multi-path region in Replace causes another branch to be symbolic. This conclusion is also in alignment with the observation made by Kuznetsov et al. [15] that path merging can sometimes have a net negative effect on the performance of symbolic execution. We plan to integrate heuristics for estimating the side-effects of introducing new symbolic state as a result of path merging on symbolic execution performance with Java Ranger in the future.

5 FUTURE WORK

While Java Ranger has the potential to scale to large real-world Java programs, there are a number of directions along which it can be further improved.

- Java Ranger attempts to perform path merging as aggressively as possible. This path merging strategy doesn't optimize 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.
- Java Ranger is most useful when it can merge multiple execution paths into one summary. But such merging causes an increase in the size of the summary, and consequently, the path condition. Once a summary has been communicated to the solver, it need not be sent again as long as it remains the same. This requires us to use the solver in an incremental mode, where we use previously-constructed state in the solver for future solving. Our current implementation of Java Ranger sends the entire path condition to the solver with every query, making every query even more expensive in the presence of large multi-path region summaries. We plan to integrate incremental solving with Java Ranger in the future.
- Multiple return instructions that appear on all execution paths inside a multi-path region can be simplified to a single return value of the region. We implemented this transformation manually in TCAS but we plan to automate it in the future. Such automation would allow us to summarize regions with return instructions.
- While statically summarizing regions gives dynamic symbolic execution a performance boost to explore more paths efficiently, generating test cases that covers all summarized branches is one of the fundamental roles of dynamic symbolic execution that is currently unsupported. This is an extension that we intend to investigate in our future work.
- While path merging can potentially allow symbolic execution to explore interesting parts of a program sooner, the effect of path merging on search strategies, such as depth-first search and breadth-first search commonly used with symbolic execution, remains to be investigated. We plan to explore the integration of such guidance heuristics with path merging in the future.
- Java Ranger can summarize methods and regions in Java standard libraries. This creates potential for automatically constructing summaries of standard libraries so that Java symbolic execution engines can prevent path explosion originating from standard libraries.

6 CONCLUSION

We presented Java Ranger as a path merging tool for Java. It works by systematically applying a series of transformations over a statement recovered from the CFG. This representation provides the benefit of modularity and makes path-merging for Java symbolic execution more accessible to end users. Java Ranger has its own IR statement and it supports the construction of SSA for fields and arrays. Java Ranger provides evidence that inlining summaries of higher-order regions can lead to a further reduction in the number of execution paths that need to be explored with path merging. It supports the exploration of exceptional behavior in multi-path regions via Single Path Cases. It provides an alternative method using which a symbolic executor can summarize unexceptional behavior of a multi-path region while simultaneously capturing exceptional behavior in the region. Java Ranger reinterprets path merging for symbolic execution of Java bytecode and has the potential to allow symbolic execution to scale up to exploration of real-world Java programs.

REFERENCES

[1] WALA. http://wala.sourceforge.net/wiki/index.php/Main_Page, accessed: 2018-11-16
[2] Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: principles, techniques, and tools, vol. 2. Addison-wesley Reading (2007)
[3] Avgerinos, T., Rebert, A., Cha, S.K., Brumley, D.: Enhancing symbolic execution with veritesting. In: Proceedings of the 36th International Conference on Software Engineering. pp. 1083–1094. ICSE 2014, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2568225.2568293, http://doi.acm.org/10.1145/2568225.2568293
[4] Babic, D., Martignoni, L., McCamant, S., Song, D.: Statically-directed dynamic automated test generation. In: Proceedings of the 20th International Symposium on Software Testing and Analysis (ISSTA). pp. 12–22 (2011), http://doi.acm.org/10.1145/2001420.2001423
[5] Bodden, E., Sewe, A., Sinschek, J., Oueslati, H., Mezini, M.: Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In: Proceedings of the 33rd International Conference on Software Engineering. pp. 241–250. ICSE '11, ACM, New York, NY, USA (2011)
[6] Burnim, J., Sen, K.: Heuristics for scalable dynamic test generation. In: 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 443–446 (2008), https://doi.org/10.1109/ASE.2008.69
[7] Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)s. pp. 209–224 (2008), http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf
[8] Cha, S.K., Avgerinos, T., Rebert, A., Brumley, D.: Unleashing mayhem on binary code. In: 2012 IEEE Symposium on Security and Privacy. pp. 380–394 (May 2012)
[9] Chipounov, V., Kuznetsov, V., Candea, G.: The S2E platform: Design, implementation, and applications. ACM Trans. Comput. Syst. **30**(1), 2:1–2:49 (2012), http://doi.acm.org/10.1145/2110356.2110358
[10] Clarke, L.A.: A system to generate test data and symbolically execute programs. IEEE Trans. Software Eng. **2**(3), 215–222 (1976), https://doi.org/10.1109/TSE.1976.233817
[11] Godefroid, P., Klarlund, N., Sen, K.: Dart: Directed automated random testing. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 213–223. PLDI '05, ACM, New York, NY, USA (2005). https://doi.org/10.1145/1065010.1065036, http://doi.acm.org/10.1145/1065010.1065036
[12] Hansen, T., Schachte, P., Søndergaard, H.: State joining and splitting for the symbolic execution of binaries. In: Runtime Verification, 9th International Workshop, RV 2009, Grenoble, France, June 26-28, 2009. Selected Papers. pp. 76–92 (2009)
[13] Hutchins, M., Foster, H., Goradia, T., Ostrand, T.: Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In: Proceedings of 16th International Conference on Software Engineering. pp. 191–200 (May 1994). https://doi.org/10.1109/ICSE.1994.296778
[14] King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976), http://doi.acm.org/10.1145/360248.360252
[15] Kuznetsov, V., Kinder, J., Bucur, S., Candea, G.: Efficient state merging in symbolic execution. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 193–204. PLDI '12, ACM, New York, NY, USA (2012)
[16] 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.) Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science, vol. 9636, pp. 442–459. Springer (2016)
[17] Ottenstein, K.J., Ballance, R.A., MacCabe, A.B.: 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. pp. 257–271. PLDI '90, ACM, New York, NY, USA (1990). https://doi.org/10.1145/93542.93578, http://doi.acm.org/10.1145/93542.93578
[18] 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, https://doi.org/10.1007/s10515-013-0122-2

[19] Ramos, D.A., Engler, D.R.: Practical, low-effort equivalence verification of real code. In: Proceedings of the 23rd International Conference on Computer Aided Verification. pp. 669–685. CAV'11, Springer-Verlag, Berlin, Heidelberg (2011), http://dl.acm.org/citation.cfm?id=2032305.2032360

[20] Sen, K., Marinov, D., Agha, G.: 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. pp. 263–272. ESEC/FSE-13, ACM, New York, NY, USA (2005). https://doi.org/10.1145/1081706.1081750, http://doi.acm.org/10.1145/1081706.1081750

[21] Sen, K., Necula, G., Gong, L., Choi, W.: Multise: Multi-path symbolic execution using value summaries. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. pp. 842–853. ESEC/FSE 2015, ACM, New York, NY, USA (2015). https://doi.org/10.1145/2786805.2786830, http://doi.acm.org/10.1145/2786805.2786830

[22] Sharma, V., Hietala, K., McCamant, S.: Finding Substitutable Binary Code By Synthesizing Adaptors. In: 11th IEEE Conference on Software Testing, Validation and Verification (ICST) (Apr 2018)

[23] Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., Vigna, G.: Sok: (state of) the art of war: Offensive techniques in binary analysis. In: 2016 IEEE Symposium on Security and Privacy (SP). pp. 138–157 (May 2016). https://doi.org/10.1109/SP.2016.17

[24] Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: Augmenting fuzzing through selective symbolic execution. In: NDSS. vol. 16, pp. 1–16 (2016)

[25] Sun, W., Xu, L., Elbaum, S.: 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. pp. 79–89. ISSTA 2017, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3092703.3092706, http://doi.acm.org/10.1145/3092703.3092706

[26] Tian, C., Duan, Z., Duan, Z., Ong, C.H.L.: More effective interpolations in software model checking. In: Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering. pp. 183–193. ASE 2017, IEEE Press, Piscataway, NJ, USA (2017), http://dl.acm.org/citation.cfm?id=3155562.3155589

[27] Wang, H., Liu, T., Guan, X., Shen, C., Zheng, Q., Yang, Z.: Dependence guided symbolic execution. IEEE Transactions on Software Engineering 43(3), 252–271 (March 2017). https://doi.org/10.1109/TSE.2016.2584063

[28] Yakdan, K., Eschweiler, S., Gerhards-Padilla, E., Smith, M.: No more gotos: Decompilation using pattern-independent control-flow structuring and semantics-preserving transformations (02 2015). https://doi.org/10.14722/ndss.2015.23185

[29] Yang, G., Person, S., Rungta, N., Khurshid, S.: Directed incremental symbolic execution. ACM Transactions on Software Engineering and Methodology (TOSEM) 24(1), 3 (2014)