Java Ranger: Static Regions for Efficient Symbolic Execution of Java

***Abstract*—Merging related 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 statically translating a bounded control flow region into a single formula. This approach is a convenient way to achieve path merging as a modification to a pre-existing single-path symbolic execution engine. Avgerinos et al. evaluated their approach in a symbolic execution tool for binary code, but different design considerations apply when building tools for other languages. In this paper we explore the best way to use a veritesting approach in the 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 a *higher-order* path-merging technique to do so modularly. Java's typed memory structure is very different from a binary, but we show how the idea of static single assignment (SSA) form can be applied to object references to statically account for aliasing. We extend path merging to summarize multiple exit points that return control flow from a multi-path region into a single such exit point. We have implemented our algorithms in Java Ranger, an extension to the widely used Symbolic Pathfinder tool for Java bytecode. Our empirical evaluation shows that Java Ranger greatly reduces the search space of Java symbolic execution benchmarks with its expanded path-merging capabilities providing a significant improvement.

***Index Terms*—multi-path symbolic execution; veritesting; Symbolic PathFinder; static analysis**

I. 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 [1], [2], 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 [3], [4], equivalence checking [5], [6], vulnerability finding [7], [8], and protocol correctness checking [9]. Symbolic execution tools are available for many languages, including CREST [10] for C source code, KLEE [11] for C/C++ via LLVM, JDart [12] and Symbolic PathFinder (SPF) [13] for Java, and S2E [14], FuzzBALL [15], and `angr` [8] 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], [17] 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 [18] 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 [18], 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 will 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 dispatched methods can lead to poor performance for näive 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 [19]. 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 at 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.

A third common feature of Java at the frontier of path merging is the *return* instruction. For a region summary, a return instruction represents an *exit point* of the region. An exit point is a program location in the region at which paths in the region have been merged into a single path. If region has multiple exit points in the form of multiple return instructions, each predicated on a condition, the symbolic executor can construct a formula that represents all return values predicated on their corresponding conditions. Summarizing such multiple control-flow returning exit points of a region into a single exit point further reduces the branching factor of the region.

While summarizing higher-order regions, finding single-path cases, and converting multiple returning exit points into a single returning exit point 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.
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. The paper also proposes Single-Path Cases as an alternative to multiple transition points as defined by Avgerinos et al. [18].

A. Motivating Example

Consider the example shown in Figure 1. The code counts the number of words where the concrete value `0` acts as a delimiter for words. The `list` object refers to an `ArrayList` of 200 `Integer` objects which have an unconstrained symbolic integer as a field. Checking the number of words delimited by one or more `0`'s requires 2^{200} execution paths because vanilla symbolic execution needs to branch on comparing every entry in the list to `0`. However, we can avoid this path explosion by merge the two paths arising out of the `list.get(i) == 0` branch. Such path merging requires us to

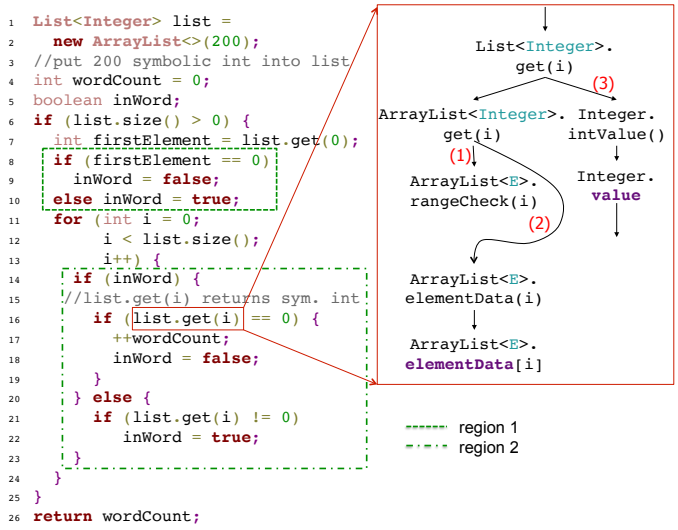

Fig. 1: An example where Java Ranger generates a multi-path region summary for two regions

compute a summary of all behaviors arising on both sides of the branch of the if-statement at line 16 in Figure 1. 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. The call to `list.get(int)` which is actually a call to `ArrayList<Integer>.get(int)` which 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 not only to include summaries of all these three methods but also to include the possibility of an exception being raised by the included summary of `ArrayList<E>.rangeCheck(i)`. 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 this example, the dynamic type of `list` is `ArrayList`, whereas it is declared statically as having the type `List`. Our contributions to path-merging uses the runtime type of `list` to inline such method summaries to merge path all 2^{200} paths into a single execution path in this example. We walk through the transformations that allow such path merging on this example in Section III.

II. 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. [16] explored this technique but found mixed results on its benefits. Kuznetsov et al. [17] 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. [20], 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. [18] 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 we chose it when extending 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 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 [21].

The veritesting approach has been integrated with another binary level symbolic execution engine named `angr` [8]. 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 in Section IV, Java Ranger performs similarly to a bounded symbolic model checker for Java. SPF and Java Ranger build on Java Pathfinder (JPF) [22], 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 [23], ESC/Java2 [24], and OpenJML [25], though these tools target static error checking and verification of annotated specifications.

Perhaps the most closely related Java model checking tool is JBMC [26], 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. [27] 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 [28]. 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.

III. Technique

Algorithm 1 describes how Java Ranger works. The main idea is that JR intercepts conditional branch instructions with symbolic operands and attempts to summarize instructions that lie along every possible path in the region, up until the program location where all paths in the region merge. We call this summarization a "region" or just R. It is important

Algorithm 1: Ranger general pseudocode

```
1  Input: (symbolic conditional branch instruction inst);
2  R_initial = jit-region-construction(inst);
3  R_no_return = return transformation(R_initial);
4  R_alpha = alpha-renaming transformation(R_no_return);
5  R_before = R_alpha;
6  R_after = null;
7  repeat
8      repeat
9          if (R_after ≠ null) then
10             R_before = R_after
11         R_sub = substitution transformation(R_before);
12         R_ord = higher-order transformation(R_sub);
13         R_field = field references transformation(R_ord);
14         R_array = array references transformation(R_field);
15         R_simple = simplification transformation(R_array);
16         R_after = R_simple;
17     until (R_before = R_after);
18     R_after = higher order transformation(R_after);
19 until (R_before = R_after);
20 R_single_path = single-path cases transformation(R_after);
21 R_linear = linearization transformation(R_single);
22 if is-fully-linearized(R_linear) then
23     R_green = to-green transformation(R_linear);
24     populate outputs // successful transformation
25     skip region execution;
26 else
27     abort // resume DSE from cond. branch
28 end
```

⟨*stmt*⟩ ::= skip | λ x.⟨*stmt*⟩
| ⟨*stmt*⟩ ; ⟨*stmt*⟩ | ⟨*exp*⟩ := ⟨*exp*⟩
| ⟨*exit_stmt*⟩ | if ⟨*exp*⟩ then ⟨*stmt*⟩ else ⟨*stmt*⟩
| invoke(⟨*exp*⟩, ⟨*exp*⟩,⟨*exp*⟩) | exit

⟨*exit-stmt*⟩ ::= new τ | return ⟨*exp*⟩ | throw ⟨*exp*⟩

⟨*exp*⟩ ::= ⟨*val*⟩ | ⟨*var*⟩ | ⟨*exp*⟩ op_b ⟨*exp*⟩ | op_u ⟨*exp*⟩
| get_field(⟨*exp*⟩) | gamma(⟨*exp*⟩, ⟨*exp*⟩, ⟨*exp*⟩)
| put_field(⟨*exp*⟩, ⟨*exp*⟩) | array_load(⟨*exp*⟩, ⟨*exp*⟩)
| array_store(⟨*exp*⟩, ⟨*exp*⟩, ⟨*exp*⟩)

⟨op_b⟩ ::= $+$ | $-$ | $*$ | $\div$ | & | bitwise-or | $\oplus$ | % | $==$ | $\neq$ | $\leq$ | $\geq$ | && | logical-or | $>|$ $<|$ $\ll$ | $\gg$ | $\ggg$

⟨op_u⟩ ::= $-$ | ~ ⟨*val*⟩ ::= ( ) | $\mathbb{Z}$ | $\mathbb{C}$ ⟨*var*⟩ ::= ID_$\mathbb{N}$

Fig. 2: 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
	$\Gamma_s : val \rightarrow$	τ
Region-map	$\Psi : \tau \rightarrow$	$\lambda x.s$
Path-subscript-map	$psm : val \rightarrow$	var
Location-map	$\Theta : var \rightarrow$	loc
Path-constraints	$\Sigma : \{exp\}$	
Return-constraints	$\Sigma_{ret} : x_{ret} \rightarrow$	exp
Nominal-next-instruction	$Next :$	$\mathbb{Z}$
Return-next-instruction	$Next_{ret} :$	$\mathbb{Z}$

Fig. 3: Environments used in Ranger.

to note that, R contains not only the summarization in form of a Ranger IR statement, it also contains other environment-related information, such as types, input and outputs of the region, explained later in this section.

Java Ranger uses Just-In-Time (JIT) analysis and, with the help of Wala [29], constructs the control-flow graph (CFG) of the method containing the intercepted symbolic conditional branch instruction. Then JR translates a part of the CFG, that begins from the node containing the symbolic conditional branch and ends at this node's immediate post-dominator, into a Ranger IR statement and an environment (line 2). We call this step "JIT region construction". The region is said to "begin" at the symbolic conditional branch instruction and "end" at the address of the first instruction in the immediate post-dominator of the node that contained the symbolic conditional branch instruction.

Next, Java Ranger then runs different transformations on R (lines 2-22) with the goal of reducing the region's IR statement into a fully-linearized form that can be easily translated into a formula.

These transformations perform operations, like for example inlining method summaries and supporting field accesses, that will allow the Ranger IR statement to ultimately be translated to a formula. But, dependencies between these transformations are not pre-determined. For example, a region can contain field accesses and a method invocation, the summary of which contains other field accesses. Using a nested fixed-point computation, JR ensures that such dependencies are taken into consideration.

If the transformations produce a fully-linearized form of the region, then Java Ranger adds the translated solver constraint of the region to the path condition, populates the region's outputs, and sets the program counter (the next instruction to be executed) to the address of the instruction that is at the beginning of the immediate post-dominator. Otherwise, JR aborts and allows DSE to resume execution (lines 23-28).

In the following sections, we first describe Java Ranger grammar then we discuss JIT region construction then finally we talk about different JR transformations and their semantic meaning. We also show how each transformation is realized on the motivating example presented in Figure 1

A. Java Ranger Grammar

Figure 2 shows grammar used for constructing Ranger IR, which is a superset of Wala IR.

Ranger IR's grammar is composed of statements $stmt$ and expressions exp. Statements include, a region statement

$\lambda x.\langle stmt \rangle$, which defines the binding of the input parameter in the region statement. Other statements in Ranger IR also include: sequential composition ($stmt; stmt$), assignment ($exp := exp$), skip, exit ($exit\text{-}stmt$) (which determine single-path Cases, it includes $new\ \tau$, $throw\ exp$ and $return\ exp$ statements), if-then-else and finally $invoke$ statement to indicate call of a method.

Ranger IR expressions includes constants aka values (unit ( ) , positive and negative integers $\mathbb{Z}$, characters $\mathbb{C}$), variables (these are subscripted with integers to facilitate having SSA form for them ID_$\mathbb{N}$), binary and unary expressions as well as a special gamma expression, $gamma(exp, exp, exp)$. Gamma expressions allow Ranger IR to construct Gated Single Assignment (GSA) [30] form for variables which intuitively describes conditional assignment of variables. Ranger IR expressions also include reads and writes to fields and arrays. A fully-linearized statement in Ranger IR only includes composition and assignment over fully-linearized expressions. A fully linearized expression consists of unary, binary, and ternary operators (aka gamma expressions) over Ranger IR variables. Java Ranger defines 5 types of variables, variables used to construct SSA form for field and array accesses, a return variable used to capture return values in a region), variables that correspond to local variables, and variables used to represent intermediate computation. This distinction is used where necessary in the rest of this work.

B. JIT Region Construction

The construction of a region happens when Java Ranger detects a symbolic conditional branch during dynamic symbolic execution (DSE), and thus an opportunity for path-merging. Using the recovered CFG of the method where the intercepted conditional branch instruction resides, JR constructs two types of regions: *multi-path region* and *method-region*.

A *multi-path region* contains a statement in Ranger IR and a corresponding environment and it always begins on a conditional branch instruction. A Ranger IR statement represents the summarization of the instructions lying in the part of the CFG, that begins from the basic block containing the conditional branch and ends at this basic block's immediate post-dominator. The region environment that contains information about variables in the region, such as types of variables, variables stack slot information, inputs of the region and output of the region.

A *method-region* also contains a statement in Ranger IR and an environment. But, the IR statement represents the summarization of the whole method. The environment for a method-region is similar to the multi-path region, except that the input and the stack slot of variables are defined in terms of the parameters to the method. A method region begins at the beginning of the method that it is summarizing and ends on a `return` instruction. In the next subsections, we describe the two main steps in constructing a region, the IR statement creation, which we call *IR statement recovery* and the environment creation for regions.

1) IR Statement Recovery: While Java bytecode operates as a register (used for local variables) and stack (used for instruction operands) machine, the recovered Ranger IR captures the region in Static Single Assignment (SSA) form with inputs corresponding to local variables and outputs corresponding to local variables or the stack. The algorithms used in this step are similar to those used for decompilation [31]. Starting from an initial basic block in a control-flow graph recovered by Wala [29], 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 for this 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. The outcome of this step is a statement in Java Ranger. This step does not compute loop summaries, it does summarize regions contained before, within, and after loops. For region 2 shown in the motivating example in Figure 1, the recovered statement is as follows where $x58$ corresponds to input from $inWord$, and $x54$, $x55$ correspond to outputs to $wordCount$, $inWord$ respectively.

```
if ((! (== x58 0 ) )) {
  x44 = invokeinterface < Application, Ljava/util/List,
       get(I)Ljava/lang/Object; >[x9,x59]
  x47 = invokevirtual < Application, Ljava/lang/Integer,
       intValue()I > x44
  if ((! (!= x47 0 ) )) {
    x48 := (+ x57 1 );
  }
} else { ... }
x54 := (Gamma !(x58==0) (Gamma !(x47!=0) x48 x57) (Gamma
     !(x53==0) x57 x57));
x55 := (Gamma !(x58==0) (Gamma !(x47!=0) 0 x58) (Gamma
     !(x53==0) 1 x58));
```

2) Environment Creation: Once the statement of a region has been recovered, i.e., a corresponding Ranger IR statement has been created, JR then tries to populate its corresponding environment. This includes identifying the region boundary, in terms of the region's local variable inputs and outputs, types, and stack slots of variables used in the region summary.

JR defines local variable inputs as variables in the region summary that make the first *use* of a given stack slot. These are input variables to the region because, when the DSE hits a symbolic region where there is a opportunity of path-merging, JR collects information (concrete/symbolic values) from the dynamic environment of DSE, and substitute them in the region's statements. JR reads values from stack slots and substitutes local input variables with their corresponding concrete or symbolic values. Similarly, local variable outputs are the last *def* of a Ranger IR variable in a region summary that maps to a given stack slot. JR uses this information

to populate symbolic expression or concrete values to local outputs of a region. JR relies on Wala to provide stack slot information for determining which variables in Ranger IR correspond to local variables. Since Ranger IR is a superset of Wala IR, and Wala IR contains ϕ expressions to represent the merging of variables assigned along different paths within the region to a single variable. In order to ensure correctness of Ranger-IR-variable-to-stack-slot mapping, JR makes use of an assumption about ϕ expressions: all variables used in a ϕ-expression must belong to the same stack slot.

Formally we define the following structures:

Value-map: Δ_r and Δ_s maps a location (i.e, stack slots, heap entry), to concrete val or symbolic exp for Java Ranger and DSE respectively. This is used to collect the input of the region as well as to populate the output of the region.

Type-map: Γ_r and Γ_s maps vars or references, to types τ for Java Ranger and DSE respectively.

Region-map: Java Ranger defines a map Ψ from type τ to a parametric statement $\lambda x.s$ that defines the region's summary. This is used to bind the region type to its statement. The actual form of the region type is abstracted here, but really it is defined by class-name, method-name, and instruction line number. This way, if a region is every hit again, i.e., executed again, then instead of re-running the jit-region construction, JR just pulls the region from the map.

Path-subscript-map: This is used to associate with every reference, the last subscripted variable.

Location-map: a map Θ that defines a location mapping for each variable in the Java Ranger IR statement, i.e, which stack-slot or part of the heap does it refer to.

Path-Constraint-List: A list Σ of path constraints. This is later used to guide the DSE execution to particular paths that JR did not summarize, more information is in next section.

Return-Constraint-map: A map Σ_{ret} from return vars x_{ret} to return constraints. This is used to set up the path condition for return case, more information is in next section.

Nominal-next-instruction: is a the instruction position that DSE should resume from, if path-merging was successful.

Return-Next-instruction: is a the instruction position that DSE should resume from, if path-merging was successful for return case.

Figure 4 shows the semantics of different transformations, where the judgment describes a transformation step on some statement s, w.r.t. the maps, which yield a new statement s' and changes in the map.

C. Transformations

Return Transformation: In this transformation, Java Ranger converts multi-path region with *one or more* return statements or a method region with *more than one* return statement, into a region that assigns a return variable x_{ret} as the region's return value. x_{ret} is assigned to a $gamma$ expression that evaluates to different return values, each predicated on the condition that would cause that value to be returned. The $\texttt{return}_1$ and $\texttt{return}_2$ rules describe the semantics of this transformation, where the first rule describes the outermost transformation of a conditional if with the setting of the next instruction to be executed, i.e., a return or continue with the next instruction following the if-condition. The second rule describes the event of multiple returns inside a conditional if, where all possible return values are summarized in the assignment of x depending on the gamma evaluation.

Alpha-renaming Transformation: In this transformation, all Ranger IR variables are renamed by adding a subscript unique to each instantiation of the region. This transformation helps distinguish variables across multiple instantiations of the same region. To give an example of this transformation, line 2 in the recovered Ranger IR statement gets changed to:

```
x44$2 = invoke < Application, Ljava/util/List,
     get(I)Ljava/lang/Object; >[x9$2, x59$2]
```

Local Variable Substitution Transformation: During this transformation, local variable inputs in JR are substituted with their concrete or symbolic values obtained from the environment of DSE. The `substitution` rule in Figure 4 describes the semantics of this transformation where the location of an input variable l is used to lookup its value in the DSE environment Δ_s and is substituted in the region statement. Running this transformation on the previous Ranger IR statement causes $x9\$2$ to be resolved to a concrete integer value 375 to change the statement to

```
x44$2= invoke(ArrayList.get(I)Ljava.lang.Object,375, x59$2);
```

Higher-order Regions Transformation: This transformation is initiated when a method invocation after local variable substitution. At this point, we perform three steps. (1) The called method's region is retrieved, depending on the dynamic type obtained from the baseline symbolic executor's environment, if the region is not found then JIT-region construction is called for it. If a method region was finally constructed/retrieved then its variables are alpha-renamed. (2) For concrete object references that JR was able to substitute for callee objects (if it is non-static invocation, otherwise no object reference is need), JR uses expressions of IR parameters to substitute the formal parameters by repeatedly applying local variable substitution transformation over the method region. (3) When no more higher-order method summaries can be inlined, the resulting substituted method region is inlined into the outer region.

The $\texttt{high-order}$ rule describes inlining of methods that have a single return values for concrete object reference. In this rule, e_1 reduces using substitution to a concrete reference v_1 type, then using the DSE environment method-region $\lambda x.s_2$ is pulled out from region map Ψ. Finally inlining of the method-region is down where values v_3 substitute formal parameters.

To show the effect of this transformation on the motivating example, regions that defines $ArrayList.get(I)java.lang.Object$ are inlined in the original region to get the following Ranger IR statement. Please note that $x58$ from the originally recovered static summary has been substituted by $x40\$1$ because it was read

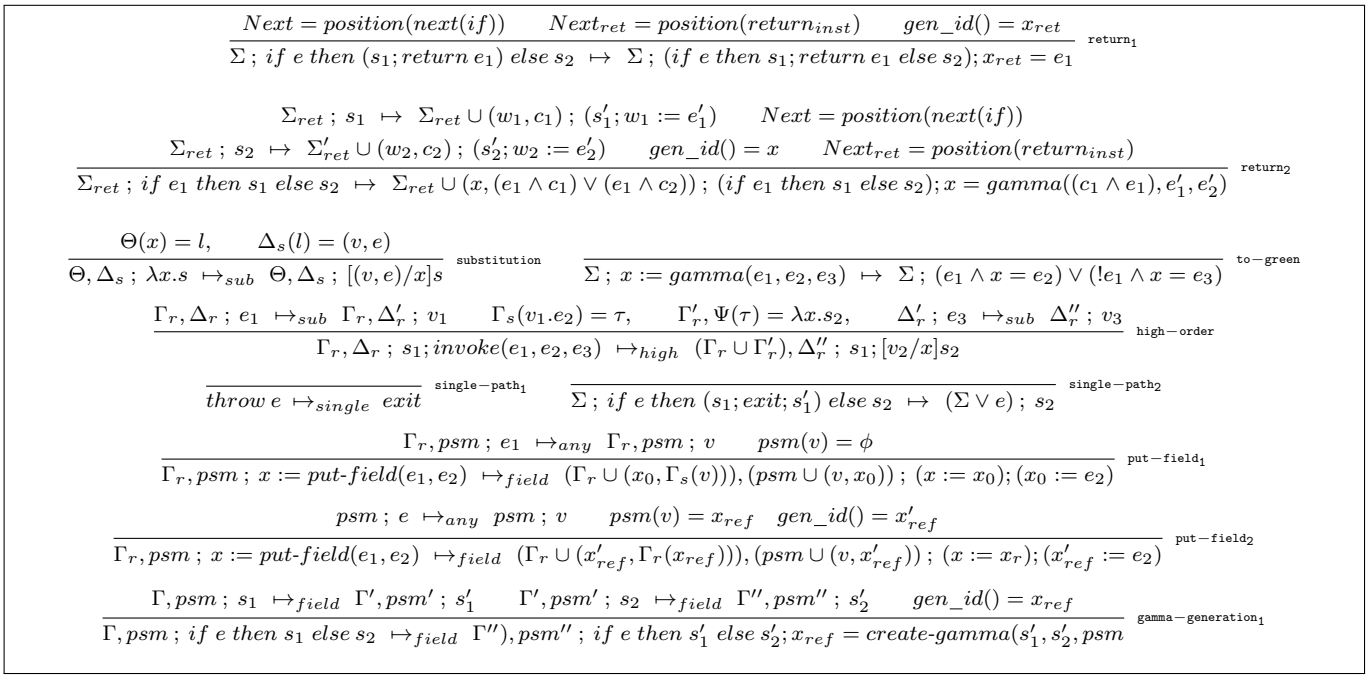

Fig. 4: Subset of the Evaluation Rules for Ranger Transformations

as a local variable input ($inWord$) that had the symbolic value $x40\$1$ (a consequence of summarizing region 1 in Figure 1).

```
if ((! (== x40$1 0 ) )) {
  x4$4 = get(375.< Primordial, Ljava/util/ArrayList, size,
       <Primordial,I> >)
  if ((! (< 0 x4$4 ) )) {
    Throw Instruction
  }
  x4$5 = get(375.< Primordial, Ljava/util/ArrayList,
       elementData, <Primordial,[Ljava/lang/Object> >)
  x5$5 = x4$5[0:<Primordial,Ljava/lang/Object>]
  x6$3 = x5$5;
  x44$2 = x6$3;
  [x47$2] = invoke < Application, Ljava/lang/Integer,
       intValue()I >[x44$2]
   if ((! (!= x47$2 0 ) )) {
      x48$2 = (+ 0 1 );
   }
} else { ... }
x54$2 := (Gamma !(x40$1==0) (Gamma !(x47$2!=0) x48$2 0) (Gamma
     !(x53$2==0) 0 0));
x55$2 := (Gamma !(x40$1==0) (Gamma !(x47$2!=0) 0 x40$1) (Gamma
     !(x53$2==0) 1 x40$1));
```

Field References SSA Transformation: This 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) [30].

Rule $\texttt{put-field}_1$ and $\texttt{put-field}_2$ describes the operation of the field transformation over a put-field instruction and rule $\texttt{gamma-generation}_1$ describes the generation of a gamma expression for field SSA vars. The first rule describes a put-field for a concrete reference v that has not been mapped to SSA variable. In this case, JR defines a new var with initial subscript x_0 and adds that to the psm map. The second rule describes a situation where the reference has an occurance in psm map, and therefore a new subscript variable is being created x'_{ref} that would be assigned the put field value. Finally the $\texttt{gamma-generation}$ rule, describes the situation where on different sides of an if instruction psm had different subscripted variables appear for the same referenced variable, in this case JR creates a gamma statement.

To continue expanding the motivating example, the field references transformation on the above Ranger IR statement changes the assignments to $x4\$4$ and $x4\$5$ to assign them the concrete values 200 and 397 respectively. Since Java Ranger runs a simplification transformation (which has constant propagation and if-then-else statement simplification) within the same fixed point iteration, this Ranger IR statement is simplified to the following statement. Please note that the simplification transformation sets variables assigned to a constant value in a constant values map maintained in the region's instantiation environment. This causes uses of $x48\$2$ to be substituted by the constant 1.

```
if ((!= x40$1 0 )) {
```

```
  x5$5 = 397[0:<Primordial,Ljava/lang/Object>]
  x45$2 := x5$5;
  [x47$2] = invoke < Application, Ljava/lang/Integer,
       intValue()I >[x5$5]
} else { ... }
x54$2 := (Gamma x40$1!=0 (Gamma x47$2==0 1 0) 0);
x55$2 := (Gamma x40$1!=0 (Gamma x47$2==0 0 x40$1) (Gamma
     x53$2!=0 1 x40$1));
```

Array References SSA Transformation: This 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 copy of the array. All array copies are merged at the merge point of the region. The merged array copy represents array outputs of the region. The effect of this transformation on the last shown Ranger IR statement produces the following statement where the value at index 0 in array reference 397 was 380 and the length of array at reference 397 was 200.

```
if ((!= x40$1 0 )) {
  if ((&& (< 0 200 ) (>= 0 0 ) )) {
    x5$5 := 380;
  } else {
    Throw Instruction
  }
  x45$2 := x5$5;
  [x47$2] = invoke < Application, Ljava/lang/Integer,
       intValue()I >[x5$5]
} else { ... }
x54$2 := (Gamma x40$1!=0 (Gamma x47$2==0 1 0) 0);
x55$2 := (Gamma x40$1!=0 (Gamma x47$2==0 0 x40$1) (Gamma
     x53$2!=0 1 x40$1));
```

Simplification of Ranger IR: This transformation uses constant propagation, copy propagation, and constant folding [32] to shorten the summary by representing constant assignments and copies in the region's instantiation environment. This transformation also simplifies if-then-else expressions and if-then-else statements where the choices are syntactically equal. Running simplification on the previous Ranger IR statement yields the following statement.

```
if ((!= x40$1 0 )) {
  [x47$2] = invoke < Application, Ljava/lang/Integer,
       intValue()I >[380]
} else { ... }
x54$2 := (Gamma x40$1!=0 (Gamma x47$2==0 1 0) 0);
x55$2 := (Gamma x40$1!=0 (Gamma x47$2==0 0 x40$1) (Gamma
     x53$2!=0 1 x40$1));
```

Termination of the fixed-point computation loop: The above Ranger IR statement is not fully-linearized because we still need to inline the method summary for `Integer.intValue()` and perform simplification on it. Therefore, at this point, Java Ranger would run one more iteration of the nested fixed-point computation. Every iteration through the fixed-point loop modifies the Ranger IR summary so that it is closer to having fully-linearized form. Once Java Ranger cannot modify the Ranger IR summary any further, it proceeds with the following transformations. Otherwise, it aborts using this region and transfers control back to DSE.

Single-Path Cases: This transformation collects path predicates inside a region that lead to *non-nominal* exit point as an alternative to transition points [18]. We define non-nominal exit points to be program locations inside the region that either cause exceptional behavior or use behavior that we cannot summarize, i.e, object creation. This feature allows path-merging by exploring unexceptional behavior in the region separately from exceptional behavior and potentially reduces the branching factor in the multi-path region.

In this transformation, Java Ranger identifies non-nominal exit points, collects their path predicates and prunes them away from the Ranger IR statement. This transformation is defined in the $\texttt{single-path}_1$ and $\texttt{single-path}_2$ rules, where the former transforms statements that are considered exit points, then the latter 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.

Linearization: At the stage when this transformation is run, all GSA expressions have been computed, and so, Ranger IR statement 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 behavior happening on both sides of the branch. Running this transformation on the last shown Ranger IR statement (after another simplification and inlining of the summary of $Integer.intValue()$) produces the following statement. Please note that the $value$ field accessed by $Integer.intValue()$ in the object referenced by the value 380 was set to the symbolic integer $x1$. The variables $x54\$2$ and $x55\$2$ correspond to outputs to $wordCount$, $inWord$ in Figure 1 respectively.

```
x54$2 := (Gamma x40$1!=0 (Gamma x1==0 1 0) 0);
x55$2 := (Gamma x40$1!=0 (Gamma x1==0 0 x40$1) (Gamma x1!=0 1
     x40$1));
```

Translation to Green: At this point, the region summary contains only compositional statements with assignment statements that contain GSA expressions. Compositional statements are translated into conjunction, assignment statements are translated into Green [33] equality expressions. For assignment statements that have GSA expressions (shown in the $\texttt{to-green}$ rule in Figure 4), these are translated into two disjunctive formulas that describes the assignment if the GSA condition or its negation were satisfied. We translate region summaries from Ranger IR to Green because we found Green constraints to be a more pliable interface for translation from Ranger IR to the solver than SPF's existing constraints.

Maintaining soundness: All the Ranger IR transformations described above are semantics-preserving transformations and therefore do not adversely affect the soundness of the baseline symbolic executor. One transformation in which this may

not be obvious is the array references transformation. On encountering an array access with a symbolic index, a baseline symbolic executor can choose to explore behavior caused due to the index being less than 0, equal to and greater than the size of the array, and being within bounds of the array length. The array references transformation preserves these semantics of out-of-bounds exploration of each array access by adding a `if` statement into the region summary for each array access. This `if` statement contains a `throw` Ranger IR statement on the then-side and uses a condition that is a disjunction of the aforementioned out-of-bounds conditions that check the symbolic index. Since `throw` Ranger IR statements are explored as single-path cases, Java Ranger explores the possibility of the same out-of-bounds array accesses as would be explored without path-merging of array accesses.

IV. Evaluation

A. Experimental Setup

We implemented the above mentioned transformations as a wrapper around the Symbolic PathFinder [13] 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 program location to be executed, checks if the next instruction is a conditional branch instruction with at least one symbolic operand. On finding such a symbolic branch condition, Java Ranger attempts to compute a static summary of all multi-path regions in the method which contains the current program location. After this Just-In-Time static analysis, if Java Ranger summarized the multi-path region that begins at the current program location, Java Ranger computes the fully-linearized region corresponding to the current program location by reading inputs from and writing outputs to the stack and the heap. Finally, it conjuncts the fully-linearized region summary with the path condition and resumes symbolic execution at the bytecode offset of the end of the region. Java Ranger wraps around SPF and can be configured to either run SPF without any path merging or run SPF with the following four path-merging features enabled.

(1) Java Ranger only transforms multi-path regions with a single exit point to their fully-linearized form. This includes multi-path regions that have local, stack, field, or array outputs. Java Ranger substitutes local inputs into the Ranger IR representation of the multi-path region, and constructs SSA form Ranger IR for all field and array accesses in the region using its runtime context. The SSA form representation of all field and array accesses allows the Ranger IR to be simplified uniformly across all variable types which reduces the size of the region summary and improves performance. While our current implementation of Java Ranger cannot transform summaries with symbolic object and array references, it is capable of summarizing reads and writes to arrays with symbolic indices. (2) Java Ranger uses multi-path region summaries that make method calls which have also been statically summarized. Method summaries are inlined into the multi-path region summary based on the dynamic type of the method. (3) Java Ranger uses single-path cases to allow regions to have more than one exit point. These exit points take the form of new object allocations and exceptional behavior present in the region. (4) Java Ranger converts multiple exit points that return control flow from the region to its caller into a single control flow-returning exit point. This feature allows summarization of multiple `return` instructions in a region into a single control-flow returning exit point. Instead of a single return value, such an exit point returns a formula to its caller, which predicates each return value on its corresponding condition in the region.

We used the control-flow graph recovered by Wala [29] to bootstrap our static statement recovery process. We ran the above implementation using the incremental solving mode of Z3 using the bitvector theory. The incremental solving mode provides only the last constructed constraint to the solver instead of passing the entire path condition every time a query is to be solved. Since path-merging can create large formulas in the path condition, the incremental solving mode provided a crucial benefit in reducing the number of times large formulas had to be passed to the solver. Finally, we also built in a heuristic that estimates the number of paths through a fully-linearized region summary and compares it to the number of exit points present in the region. The fully-linearized region summary is used only if the estimated number of paths is greater than the number of exit points in the region.

B. Evaluation

In order to evaluate the performance of Java Ranger, we used the following nine benchmarking programs commonly used to evaluate symbolic execution performance. Eight of these programs were provided by Wang et al. [34] which also includes a translation of the Siemens suite to Java. (1) Wheel Brake System (WBS) [35] 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 [36]. 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. (4) NanoXML is an XML Parser written in Java which consists of 129 procedures and 4608 lines of code. (5) Siena (Scalable Internet Event Notification Architecture) is an Internet-scale event notification middleware for distributed event-based applications [37] which consists of 94 procedures and 1256 lines of code. (6) Schedule2 is a priority scheduler which consists of 27 procedures and 306 lines of code. (7) PrintTokens2 is a lexical analyzer which consists of 30 procedures and 570 lines of code. (8) ApacheCLI [38] provides an API for parsing command lines options passed to programs. It consists of 183 procedures and 3612 lines of code. (9) MerArbiter models a flight software component of NASA JPL's Mars Exploration Rovers. We used the version made available by Yang et al. [39]. This benchmark consists of

268 classes, 553 methods, 4697 lines of code including the Polyglot framework. While none of our existing benchmarks performed multi-threaded execution, path-merging can interfere with symbolic execution of multi-threaded programs. We plan to explore this extension of path-merging for symbolic execution of multi-threaded programs in the future.

We first ran each of these benchmarks using SPF with increasing number of symbolic inputs and obtained the most number of symbolic inputs with which SPF explored all execution paths in each benchmark within a 24 hour time budget. We then ran each benchmark with this number of symbolic inputs with Java Ranger. This evaluation allowed us to check if Java Ranger is faster than SPF at exploring all feasible paths through each benchmark. We report results from this evaluation for each benchmark in Table I. Table I shows that Java Ranger achieves a significant speed-up over SPF with 5 (WBS, TCAS, NanoXML, ApacheCLI, MerArbiter) of the 9 benchmarks in terms of both running time and number of execution paths. Java Ranger also achieves a modest 22% reduction in running time and 34% reduction in the number of execution paths with the PrintTokens2 benchmark.

Java Ranger is able to summarize the entire step function of WBS and TCAS into a single execution path. This step functions of WBS and TCAS take 3 and 12 symbolic inputs respectively. In WBS, Java Ranger summarizes multi-path regions with deeply nested `if` bytecode instructions, in one case summarizing a region that consisted of 9 branches nested within one another. In TCAS, Java Ranger inlines 28 method summaries in each step of TCAS, many of which summarize multiple return values into a single formula that represents all the return values of the method. While SPF does not finish more than 5 steps of WBS and 2 steps of TCAS within 24 hours, Java Ranger finishes 10 steps of both benchmarks within 2.81 seconds and 1.41 seconds respectively.

In the replace benchmark, while Java Ranger reduces the number of execution paths by about 88%, it incurs an increase in execution time. This increase occurs even when Java Ranger transforms 20 distinct regions into 7334 fully-linearized region summaries. On manually investigating the set of instantiated regions in replace, we found that the outputs of most of these regions were being branched on later in the code causing the benefit from path-merging to be lost. In order to test this hypothesis, we ran an automated evaluation where we restricted Java Ranger to only a subset of regions. First, we determined the most number of symbolic inputs with which Java Ranger could explore all paths in replace in less than a minute. We found this number to be 6 symbolic inputs which took Java Ranger about 37 seconds while using 14 regions. Next, we constructed a list of region subsets sorted in increasing order of the subset size. This list began with 14 region subsets, each containing one region, and ended with a single set containing all 14 regions. The total number of region subsets present in this list was $2^{14} - 1$. Finally, we ran Java Ranger with every region subset in this list, where Java Ranger was allowed to only fully-linearize regions within a given subset. After running this evaluation for 96 hours, we found that all possible region subsets containing up to 4 regions had been tested. But, no region subset up to 4 regions in size resulted with a reduction in the running time and number of execution paths. While it is possible that there is a larger set of regions in the replace benchmark that provides the most benefit from path-merging, our automated evaluation and a manual investigation did not reveal what such a set of regions might be. In the future, we plan to integrate a query count estimated heuristic of the kind proposed by Kuznetsov et al. [17] to fully-linearize only a beneficial set of regions in the replace benchmark.

On the NanoXML benchmark, Java Ranger finds several opportunities for execution path count reduction on account of the NanoXML benchmark having several methods with multi-path regions that have a control-flow returning instruction on every branch side. Since Java Ranger can convert such multiple control-flow returning exit points regions into a single control-flow returning exit point, Java Ranger is able to inline more than 8,000 method summaries when running NanoXML with 8 symbolic inputs and finish in about 8 hours while vanilla SPF needs about 18 hours to explore all feasible paths.

Java Ranger has the same performance as SPF on Siena and Schedule2 with a minor overhead (about 5% in running time) in running time that comes from Java Ranger's lookup of a region summary for every executed conditional branch instruction, recording of metrics, and from doing JIT static analysis to recover region statements. We restrict our results with Siena to 7 symbolic inputs because 7 is the most number of symbolic inputs for which SPF finishes exploring all feasible paths within a 24 hour time budget. We do not attempt to construct a fully linearized region if the region does not begin on a symbolic conditional branch. In the Siena benchmark, Java Ranger begins to transform four different multi-path regions. But, at the point where it is about to fully linearize the region summary, it realizes that the number of estimated paths through the region is equal to the number of single-path cases it would need to execute through the region. Since using the fully linearized form of these regions does not lead to a reduction in the number of paths through the region, Java Ranger does not use these region summaries. On the Schedule2 benchmark, Java Ranger begins to transform six different multi-path regions. But, when attempting to inline method summaries, it fails to summarize a crucial method, `Schedule2.get_current()` that is invoked in all six regions. This method contains a loop whose bound is symbolic. Summarizing this method would require Java Ranger to also have the ability to summarize loops with symbolic bounds (loops that are executed for a symbolic number of iterations). Since Java Ranger currently does not summarize loops, it fails to summarize `Schedule2.get_current()` which has a cascading effect of not being able to convert summaries for these six regions into a fully linearized form.

In the PrintTokens2 benchmark, Java Ranger uses 4 distinct regions, 2 of which involve summarizing multi-path regions which have a `return` instruction at the end of every path in the region. Being able to summarize multiple control-flow

TABLE I: Comparing the performance of Java Ranger with Symbolic PathFinder on 9 benchmarks. "# sym inputs' indicates the total number of symbolic inputs that were given to each benchmark. For WBS and TCAS, we ran WBS and TCAS with path-merging using symbolic inputs corresponding to 10 steps of each. "path-merging enabled" indicates a comparison of no path-merging with path-merging with all the features turned on.

Benchmark name	# sym inputs	path-merging enabled?	total runtime (sec)	static analysis time (sec)	# exec. paths	# solver queries	solver time (sec)	# distinct regions used	# inlined method summaries	# fully-linearized region summaries
WBS	15	NO	4427.66	0	7962624	15925246	3121.44	0	0	0
WBS	30	YES	2.81	3.73	1	0	0	16	0	140
TCAS	24	NO	353.07	0	39204	465262	296.23	0	0	0
TCAS	120	YES	1.41	2.29	1	0	0	4	560	40
replace	11	NO	1176.83	0	757261	4317642	749.77	0	0	0
replace	11	YES	4581.13	7.97	91983	1441674	3563.57	20	360	7334
NanoXML	8	NO	63554.90	0	33944190	77831662	30071.39	0	0	0
NanoXML	8	YES	28803.23	13.31	3637075	10363980	19742.50	10	8136	990022
Siena	7	NO	55027.94	0	35831808	91208236	26689.72	0	0	0
Siena	7	YES	57782.26	23.66	35831808	91208236	26945.8	0	0	0
Schedule2	3	NO	1.48	0	343	684	0.33	0	0	0
Schedule2	3	YES	2.52	3.42	343	684	0.32	0	0	0
PrintTokens2	4	NO	1716.27	0	166644	3491916	1305.59	0	0	0
PrintTokens2	4	YES	1338.85	4.01	109727	2404854	634.34	4	0	122504
ApacheCLI	6+1	NO	31886.46	0	4839812	64432586	9470.83	0	0	0
ApacheCLI	6+1	YES	5560.94	10.12	171818	756030	1904.21	5	0	977972
MerArbiter	28	NO	70876.22	0	2429568	6816406	3391.12	0	0	0
MerArbiter	28	YES	11087.11	5.96	276144	910023	475.49	16	0	411227

returning exit points into a single such exit point proves to be a crucial feature in Java Ranger for this benchmark.

The ApacheCLI benchmark takes 9 inputs, the first 8 are 1-byte inputs used to construct command-line options and the last input controls whether ApacheCLI should stop on encountering an invalid option input. Since the 9th input is different from the first 8, we ran ApacheCLI with the first 6 inputs and the 9th input made symbolic. Java Ranger finishes exploration with this setup of ApacheCLI in about 1.5 hours whereas SPF finishes exploration in about 9 hours as shown in Table I When we ran SPF on ApacheCLI with the first 7 inputs and the 9th input made symbolic, it was unable to explore all feasible paths in ApacheCLI within 24 hours. But, Java Ranger can complete this exploration in about 15 hours.

In the MerArbiter benchmark, the most significant benefit from Java Ranger comes from its ability to summarize multi-path regions that put their output on the stack directly. Such regions are common in Java bytecode since the JVM is both a stack machine and a register machine. All the multi-path regions that SPF (which is mode 1 in Java Ranger) needs to branch on but are summarized by Java Ranger are regions that compute a boolean value based on a symbolic branch and write it to the stack as an operand to be used by the following return instruction. Most of these multi-path regions lie inside several levels of nested classes and dynamically bound field references that necessitate a fixed-point computation over the field substitution and constant propagation transformations. Java Ranger summarizes such multi-path regions and does 411,000 instantiations with 7 steps of MerArbiter (each step takes 4 symbolic inputs), with more than 282,000 instantiations needing more than 8 iterations of the fixed-point computation.

Java Ranger has different path-merging features such as summarizing multi-path regions with a single non-returning exit point, inlining method summaries, exploring unsummarized behavior using single-path cases, and summarizing multiple control flow-returning exit points into a single such exit point. We wanted to evaluate the effect each feature has when it is added to a previous set of features. We set up an experiment where beginning with no path-merging (aka baseline SPF), we added path-merging features in the aforementioned order. For every benchmark where any path-merging was performed, we computed the ratio of three metrics with a set of path-merging features enabled to the same metrics without path-merging (aka when only baseline SPF was used). These three metrics were: the running time, the number of execution paths explored, and the number of solver calls made. We present the results of this comparison in Table II. Table II shows that merging of paths for regions that have a single non-returning exit point is most often useful. This observation matches our intuition that regions are most commonly present in Java. The addition of method summary inlining provides a major reduction in all three metrics in TCAS. This observation matches a 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 provides a major reduction in the number of solver queries in the

Benchmark name	multi-path region summ.	+ higher-order regions	+ single-path cases	+returns summ.
WBS	0.0002	0.0002	0.0002	0.0002
TCAS	0.347	0.002	0.002	0.002
replace	1.36	3.23	3.93	3.89
NanoXML	1.28	1.27	1.36	0.45
PrintTokens2	0.67	0.66	0.7	0.78
ApacheCLI	0.16	0.13	0.18	0.19
MerArbiter	0.15	0.16	0.17	0.16

(a) Comparing running time

Benchmark name	multi-path region summ.	+ higher-order regions	+ single-path cases	+returns summ.
WBS	1.30E-07	1.30E-07	1.30E-07	1.30E-07
TCAS	0.245	2.60E-05	2.60E-05	2.60E-05
replace	0.63	0.9	0.12	0.12
NanoXML	0.9998	0.9994	0.2655	0.1071
PrintTokens2	0.87	0.87	0.87	0.66
ApacheCLI	0.0035	0.0035	0.0035	0.0035
MerArbiter	0.11	0.11	0.11	0.11

(b) Comparing number of execution paths

Benchmark name	multi-path region summ.	+ higher-order regions	+ single-path cases	+returns summ.
WBS	0	0	0	0
TCAS	0.234	0	0	0
replace	0.74	1.13	0.33	0.33
NanoXML	0.9999	0.9995	0.2778	0.1332
PrintTokens2	0.92	0.92	0.92	0.69
ApacheCLI	0.05	0.05	0.05	0.05
MerArbiter	0.13	0.13	0.13	0.13

(c) Comparing number of solver queries

TABLE II: Presenting the ratio of three metrics, running time, number of execution paths, and number of solver queries, with path-merging to the same three metrics without path-merging for the 7 benchmarks where any path-merging was done. A ratio less than 1 indicates path-merging was useful in reducing the value of the metric (smaller is better). The "multi-path region summ." column represents path-merging was enabled only for multi-path regions that did not have a method invocation and had a single non-returning exit point. Every column in the next three columns adds a feature to the features enabled in the previous column. The "+ higher-order regions" column adds path-merging for regions containing method invocations. The "+ single-path cases" column adds path-merging for regions containing new object allocations, throwing of exceptions, and calls to unsummarized methods. The "+ returns summ." column adds path-merging for regions that have exit points that return control flow to the caller of the method in which the region is present.

replace and NanoXML benchmarks. But, it also causes a minor increase in the running time of both of these benchmarks. We attribute this increase to the fact that the reduction in the number of solver queries is not enough to compensate for the increase in complexity of the solver queries introduced with single-path cases. In the future, we plan to address this issue by transforming region summaries with more sophisticated term rewriting and context-specific simplification of fully-linearized regions to reduce the complexity of solver queries. The addition of summarizing multiple control flow-returning exit points into a single such exit point provides a significant reduction in the number of execution paths and number of solver queries in the NanoXML, PrintTokens2 benchmarks. The benefit from this feature matches our observation that both of these benchmarks contain frequently-executed regions that contain multiple control-flow returning exit points. In conclusion, Table II shows that every path-merging feature present in Java Ranger has a beneficial impact on atleast one benchmark in our set. It demonstrates the benefits from our re-interpretation and extension of the original veritesting approach [18] for symbolic execution of Java bytecode.

V. Future Work

While Java Ranger was able to significantly outperform SPF in a majority of our benchmarks, there are a few directions along which it can further be extended. 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 currently lacks support for symbolic object and array references. Supporting these would require integrating our implementation with SPF's lazy initialization [13] to allow symbolic object references to be part of a region.

While path-merging gives dynamic symbolic execution a performance boost to explore more paths efficiently, generating test cases that cover all branches is one of the most useful applications of dynamic symbolic execution. This is an example of an application of symbolic execution that, if applied as-is, would have to undo the benefits of path-merging. We intend to extend Java Ranger towards test generation for merged paths in the future by targeting test generation towards a coverage criterion such as Modified Condition/Decision Coverage.

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.

VI. Conclusion

We presented an extension to veritesting implemented in a tool named Java Ranger. It works by systematically applying a

series of transformations over a statement recovered from the CFG. Java Ranger's use of fixed-point computation over multiple transformations demonstrates a significant improvement over SPF. 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 as shown by the TCAS benchmark. Java Ranger's use of return value summaries also demonstrates a significant benefit over SPF with the NanoXML benchmark. Java Ranger reinterprets path merging for symbolic execution of Java bytecode and allows symbolic execution to scale to exploration of real-world Java programs.

REFERENCES

[1] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976. [Online]. Available: http://doi.acm.org/10.1145/360248.360252

[2] L. A. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Trans. Software Eng.*, vol. 2, no. 3, pp. 215–222, 1976. [Online]. Available: https://doi.org/10.1109/TSE.1976.233817

[3] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 213–223. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065036

[4] K. Sen, D. Marinov, and G. Agha, "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*, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 263–272. [Online]. Available: http://doi.acm.org/10.1145/1081706.1081750

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

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

[7] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *NDSS*, vol. 16, 2016, pp. 1–16.

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

[9] W. Sun, L. Xu, and S. Elbaum, "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*, ser. ISSTA 2017. New York, NY, USA: ACM, 2017, pp. 79–89. [Online]. Available: http://doi.acm.org/10.1145/3092703.3092706

[10] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2008, pp. 443–446. [Online]. Available: https://doi.org/10.1109/ASE.2008.69

[11] C. Cadar, D. Dunbar, and D. R. Engler, "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*, 2008, pp. 209–224. [Online]. Available: http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[12] K. Luckow, M. Dimjašević, D. Giannakopoulou, F. Howar, M. Isberner, T. Kahsai, Z. Rakamarić, and V. Raman, "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)*, ser. Lecture Notes in Computer Science, M. Chechik and J.-F. Raskin, Eds., vol. 9636. Springer, 2016, pp. 442–459.

[13] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta, "Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis," *Automated Software Engineering*, vol. 20, no. 3, pp. 391–425, Sep 2013. [Online]. Available: https://doi.org/10.1007/s10515-013-0122-2

[14] V. Chipounov, V. Kuznetsov, and G. Candea, "The S2E platform: Design, implementation, and applications," *ACM Trans. Comput. Syst.*, vol. 30, no. 1, pp. 2:1–2:49, 2012. [Online]. Available: http://doi.acm.org/10.1145/2110356.2110358

[15] D. Babic, L. Martignoni, S. McCamant, and D. Song, "Statically-directed dynamic automated test generation," in *Proceedings of the 20th International Symposium on Software Testing and Analysis (ISSTA)*, 2011, pp. 12–22. [Online]. Available: http://doi.acm.org/10.1145/2001420.2001423

[16] T. Hansen, P. Schachte, and H. Søndergaard, "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*, 2009, pp. 76–92.

[17] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 193–204.

[18] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 1083–1094. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568293

[19] V. Sharma, M. W. Whalen, S. McCamant, and W. Visser, "Veritesting challenges in symbolic execution of Java," *SIGSOFT Softw. Eng. Notes*, vol. 42, no. 4, pp. 1–5, Jan. 2018. [Online]. Available: http://doi.acm.org/10.1145/3149485.3149491

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

[21] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *2012 IEEE Symposium on Security and Privacy*, May 2012, pp. 380–394.

[22] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Automated Software Engineering*, vol. 10, no. 2, pp. 203–232, Apr 2003. [Online]. Available: https://doi.org/10.1023/A:1022920129859

[23] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for Java," in *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, 2002, pp. 234–245.

[24] D. R. Cok and J. Kiniry, "ESC/Java2: Uniting ESC/Java and JML," in *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, International Workshop, CASSIS 2004, Marseille, France, March 10-14, 2004, Revised Selected Papers*, 2004, pp. 108–128.

[25] D. R. Cok, "OpenJML: JML for Java 7 by extending OpenJDK," in *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, 2011, pp. 472–479.

[26] L. C. Cordeiro, P. Kesseli, D. Kroening, P. Schrammel, and M. Trtík, "JBMC: a bounded model checking tool for verifying Java bytecode," in *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, 2018, pp. 183–190.

[27] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, 2018.

[28] P. Godefroid, "Compositional dynamic test generation," in *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, 2007, pp. 47–54.

[29] "WALA," http://wala.sourceforge.net/wiki/index.php/Main_Page, accessed: 2018-11-16.

[30] K. J. Ottenstein, R. A. Ballance, and A. B. MacCabe, "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, ser. PLDI '90. New York, NY, USA: ACM, 1990, pp. 257–271. [Online]. Available: http://doi.acm.org/10.1145/93542.93578

[31] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith, "No more gotos: Decompilation using pattern-independent control-flow structuring and semantics-preserving transformations," in *The 2015 Network and Distributed System Security Symposium*, 02 2015.

[32] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*. Addison-wesley Reading, 2007, vol. 2.

[33] W. Visser, J. Geldenhuys, and M. B. Dwyer, "Green: Reducing, reusing and recycling constraints in program analysis," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 58:1–58:11. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393665

[34] H. Wang, T. Liu, X. Guan, C. Shen, Q. Zheng, and Z. Yang, "Dependence guided symbolic execution," *IEEE Transactions on Software Engineering*, vol. 43, no. 3, pp. 252–271, March 2017.

[35] G. Yang, S. Person, N. Rungta, and S. Khurshid, "Directed incremental symbolic execution," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 1, p. 3, 2014.

[36] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria," in *Proceedings of 16th International Conference on Software Engineering*, May 1994, pp. 191–200.

[37] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Achieving scalability and expressiveness in an internet-scale event notification service," in *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '00. New York, NY, USA: ACM, 2000, pp. 219–227. [Online]. Available: http://doi.acm.org/10.1145/343477.343622

[38] The Apache Software Foundation, "Commons CLI – Homepage," https://commons.apache.org/proper/commons-cli/, accessed: 2018-11-16.

[39] G. Yang, C. S. Păsăreanu, and S. Khurshid, "Memoized symbolic execution," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012. New York, NY, USA: ACM, 2012, pp. 144–154. [Online]. Available: http://doi.acm.org/10.1145/2338965.2336771