Java Ranger: Static Regions for Efficient Symbolic Execution of Java

Anonymous Author(s)*

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.

ACM Reference Format:
Anonymous Author(s). 2019. Java Ranger: Static Regions for Efficient Symbolic Execution of Java. In *Proceedings of The 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 [**?** **?**], 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 [**?** **?**], equivalence checking [**?** **?**], vulnerability finding [**?** **?**], and protocol correctness checking [**?**]. Symbolic execution tools are available for many languages, including CREST [**?**] for C source code, KLEE [**?**] for C/C++ via LLVM, JDart [**?**] and Symbolic PathFinder (SPF) [**?**] for Java, and S2E [**?**], FuzzBALL [**?**], and `angr` [**?**] 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 [**?** **?**] 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 [**?**] 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 [**?**], 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 [**?**]. We call such regions *higher order* as they require a

ESEC/FSE 2019, 26–30 August, 2019, Tallinn, Estonia

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 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. The paper also proposes Single-Path Cases as an alternative to multiple transition points as defined by Avgerinos et al. [**?**].

1.1 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

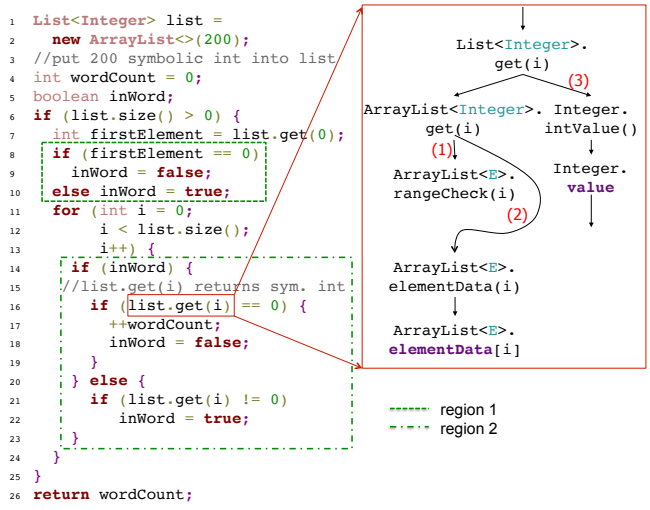


Figure 1: An example where Java Ranger generates a multi-path region summary for two regions

`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 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 3.

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. [**?**] explored this technique but found mixed results on its benefits. Kuznetsov et al. [**?**] 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. [**?**], 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. [**?**] 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 [**?**].

The veritesting approach has been integrated with another binary level symbolic execution engine named `angr` [**?**]. 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 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

To add path merging to SPF, we pre-compute static summaries of multi-path code regions and use them to also pre-compute summaries of methods. Algorithm 1 describes the core algorithm of Java Ranger for using such pre-computed summaries. Given a multi-path region summary in Ranger IR (described in Figure 2), Java Ranger runs two loops, one nested within the other, up to a fixed point in order to ensure that the type used for inlining higher-order method summaries is precise. Java Ranger ensures such precision of the higher-order region's type because it only inlines methods called using a concrete object reference for which type information can be recovered from the region's instantiation environment. In the following sections, we describe the semantics of Java Ranger's instantiation-time transformations and explain how each transformation is realized on the motivating example presented in Figure 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

```
Input: Region R;
somethingChanged = true;
execute early-return transformation(R);
execute alpha-renaming transformation(R);
repeat
    while (somethingChanged) do
        execute substitution transformation(R);
        execute higher-order method inlining
          transformation(R);
        execute field references SSA transformation(R);
        execute array references SSA transformation(R);
        execute simplification transformation(R);
    end
    execute higher-order transformation(R);
until !(somethingChanged);
execute single-path cases transformation(R);
execute linearlization transformation(R);
execute to-green transformation(R);
if R successfully transformed then
    populate outputs;
end
```

Algorithm 1: Ranger general pseudocode

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

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

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

⟨op_b⟩ ::= + | − | ∗ | ÷ | & | bitwise-or | ⊕ | % | == | ≠ | ≤ | ≥ | && | logical-or | >| <| ≪ | ≫ | ⋙

⟨op_u⟩ ::= − | ~

⟨*val*⟩ ::= ( ) | $\mathbb{Z}$ | $\mathbb{C}$

⟨*var*⟩ ::= ID_$\mathbb{N}$

Figure 2: Context Free Grammar for Java Ranger IR

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 class depth of 1 from C. We continue this method discovery process up to a class depth of 3. We found that summarizing arbitrary code regions with more than 3 steps deep did not lead to practically useful region summaries. Next, we attempt to summarize these methods and regions in them.

3.1.2 IR-Statement Recovery. The algorithms used in this step are similar to those used for those used for decompilation [**?**]. Starting from an initial basic block in a control-flow graph recovered by

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
Region-map	$\Psi : \tau \rightarrow$	$x.s$
path-subscript-map	$psm : v \rightarrow$	x
	$\Theta : var \rightarrow$	loc
Single-path-constraints	$\Sigma : \{exp\}$	
Early-return-constraints	$\Sigma_{ret} : x_{ret} \rightarrow$	exp

Figure 3: Environments used in Ranger.

Wala [**?**], 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 IR. 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.

```
1 if ((! (== x58 0 ) )) {
2   x44 = invokeinterface < Application, Ljava/util/List,
          get(I)Ljava/lang/Object; >[x9,x59]
3   x47 = invokevirtual < Application, Ljava/lang/Integer,
          intValue()I > x44
4   if ((! (!= x47 0 ) )) {
5     x48 := (+ x57 1 );
6   }
7 } else { ... }
8 x54 := (Gamma !(x58==0) (Gamma !(x47!=0) x48 x57) (Gamma
      !(x53==0) x57 x57));
9 x55 := (Gamma !(x58==0) (Gamma !(x47!=0) 0 x58) (Gamma
      !(x53==0) 1 x58));
```

We present the grammar that describes Ranger IR in Figure 2. It consists of different kinds of statements of which *exit* statements define single-path cases present in the region and *skip* statements allow simplifications to reduce the size of the region summary. Java Ranger expressions consist of constants aka 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 and unary operators over expressions, reads and writes from fields and arrays, and a special gamma expression, $gamma(exp, exp, exp)$, that is the ternary operator in Java. Gamma expressions allow Ranger IR to construct Gated Single Assignment (GSA) form for variables. Java Ranger defines 4 types of variables, a field or array variable (created to maintain fields and arrays ssa), an early return variable (to carry early return values),

variables that correspond to local variables, and variables used to represent intermediate computation.

3.2 Region Definition

Once the statement of a multi-path region has been recovered, its corresponding environment is populated. 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. The local variable inputs are Ranger IR variables in the region summary that make the first *use* of a stack slot. Similarly, local variable outputs are the last *def* of a Ranger IR variable in a region summary that maps to a stack slot. We use the instantiation environment to map Ranger IR variables to their corresponding stack slots, starting with the stack slot information given by Wala. We also assume that, if at least one variable used in a ϕ-expression of Wala maps to a stack slot, then all variables used in that ϕ-expression must belong to the same stack slot. We use this assumption to propagate stack slots across all ϕ-expressions. Formally we define the following structures:

value-map: Java Ranger and SPF value map Δ_r and Δ_s that maps 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 region's summary

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

Figure 4 shows the semantics of different transformations, with evaluation following judgment where each step on some statement s, with respect to the maps $\Delta_r, \Delta_s, \Gamma_r, \Gamma_s, \Psi, \Sigma, \Sigma_{ret}$, yields a new statement s' and possibly new maps $\Delta'_r, \Delta'_s, \Gamma'_r, \Gamma'_s, \Psi', \Sigma', \Sigma'_{ret}$ in some transformation T.

$$\Delta_r, \Delta_s, \Gamma_r, \Gamma_s, \Psi, \Sigma, \Sigma_{ret} \; ; \; s \; \mapsto_T \; \Delta'_r, \Delta'_s, \Gamma'_r, \Gamma'_s, \Psi', \Sigma', \Sigma'_{ret} \; ; \; s'$$

To avoid unnecessarily complicated rules and to promote clarity, we take the freedom of writing only maps relevant to a given rule.

3.3 Instantiation-time Transformations

Early Return Transformation: In this transformation, Java Ranger converts region summaries with return statements into region summaries that assign a x_{ret} variable a *gamma* expression that evaluates to different return values, each return value predicated on the condition that would cause that value to be returned. The `early-return`$_1$ and `early-return`$_2$ describe the semantics of this transformation.

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:

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

Local Variable Substitution Transformation: During this transformation, we substitute Ranger IR variables that are local variable inputs with their constant or symbolic values read for each local variable's stack slot. The `substitution` rule in Figure 4 describes the semantics of this transformation. 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 is encountered during local variable substitution. At this point, we perform three steps. (1) The called method's summary is retrieved and variables in it are alpha-renamed. (2) Ranger IR expressions that correspond to the method's 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 method summaries can be inlined, the resulting substituted method region is inlined into the outer region. Rules `high-order`$_1$ and `high-order`$_2$ describe the semantics of this transformation.

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

```
1  if ((! (== x40$1 0 ) )) {
2    x4$4 = get(375.< Primordial, Ljava/util/ArrayList, size,
         <Primordial,I> >)
3    if ((! (< 0 x4$4 ) )) {
4      Throw Instruction
5    }
6    x4$5 = get(375.< Primordial, Ljava/util/ArrayList,
         elementData, <Primordial,[Ljava/lang/Object> >)
7    x5$5 = x4$5[0:<Primordial,Ljava/lang/Object>]
8    x6$3 = x5$5;
9    x44$2 = x6$3;
10   [x47$2] = invoke < Application, Ljava/lang/Integer,
         intValue()I >[x44$2]
11    if ((! (!= x47$2 0 ) )) {
12      x48$2 = (+ 0 1 );
13    }
14 } else { ... }
15 x54$2 := (Gamma !(x40$1==0) (Gamma !(x47$2!=0) x48$2 0)
       (Gamma !(x53$2==0) 0 0));
16 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

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

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

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

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

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

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

$$\frac{psm \; ; \; e \mapsto_{any} psm \; ; \; v \quad \Theta(v) = l \quad \Delta_s(v) = (v, e') \quad psm(v) = \phi}{psm \; ; \; x := get\text{-}field(e) \mapsto_{field} psm \; ; \; x := (v, e')}\ \text{get-field}_1$$

$$\frac{psm \; ; \; e \mapsto_{any} psm \; ; \; v \quad psm(v) = x_r}{psm \; ; \; x := get\text{-}field(e) \mapsto_{field} psm \; ; \; x := x_r}\ \text{get-field}_2$$

$$\frac{\Gamma_r, psm \; ; \; e \mapsto_{any} \Gamma_r, psm \; ; \; v \quad \Theta(v) = l \quad psm(v) = \phi}{\Gamma_r, psm \; ; \; x := put\text{-}field(e1, e2) \mapsto_{field} (\Gamma_r \cup (x_0, \Gamma_s v)), (psm \cup (v, x_0)) \; ; \; (x := x_0); (x_0 := e2)}\ \text{put-field}_1$$

$$\frac{psm \; ; \; e \mapsto_{any} psm \; ; \; v \quad \Theta(v) = l \quad psm(v) = x_{ref} \quad gen\_id() = x'_{ref}}{\Gamma_r, psm \; ; \; x := put\text{-}field(e1, e2) \mapsto_{field} (\Gamma_r \cup (x'_{ref}, \Gamma_r(x_{ref}))), (psm \cup (v, x'_{ref})) \; ; \; (x := x_r); (x'_{ref} := e2)}\ \text{put-field}_2$$

$$\frac{\Gamma, psm \; ; \; s_1 \mapsto_{field} \Gamma', psm' \; ; \; s'_1 \quad \Gamma', psm' \; ; \; s_2 \mapsto_{field} \Gamma'', psm'' \; ; \; s'_2 \quad gen\_id() = x_{ref}}{\Gamma, psm \; ; \; if\ e\ then\ s_1\ else\ s_2 \mapsto_{field} \Gamma''), psm'' \; ; \; if\ e\ then\ s'_1\ else\ s'_2; x_{ref} = create\text{-}gamma(s'_1, s'_2, psm)}\ \text{gamma-generation}_1$$

$$\frac{gen\_id() = x_{ret} \quad \text{no\_return}(s_1) \quad \text{no\_return}(s_2)}{\Sigma_{ret} \; ; \; if\ e_1\ then\ s_1\ else\ (s_2; return\ e_2) \mapsto \Sigma_{ret} \cup (x_{ret}, e_2) \; ; \; (if\ e_1\ then\ (s_1; return\ e_2)\ else\ s_2); x_{ret} := e_2}\ \text{early-return}_1$$

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

Figure 4: Evaluation Rules for Ranger Transformations

Assignment (GSA) [**?**]. 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.

```
1 if ((!= x40$1 0 )) {
2   x5$5 = 397[0:<Primordial,Ljava/lang/Object>]
3   x45$2 := x5$5;
4   [x47$2] = invoke < Application, Ljava/lang/Integer,
        intValue()I >[x5$5]
5 } else { ... }
6 x54$2 := (Gamma x40$1!=0 (Gamma x47$2==0 1 0) 0);
7 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 multi-path regions. 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.

```
1  if ((!= x40$1 0 )) {
2    if ((&& (< 0 200 ) (>= 0 0 ) )) {
3      x5$5 := 380;
4    } else {
5      Throw Instruction
6    }
7    x45$2 := x5$5;
8    [x47$2] = invoke < Application, Ljava/lang/Integer,
         intValue()I >[x5$5]
9  } else { ... }
10 x54$2 := (Gamma x40$1!=0 (Gamma x47$2==0 1 0) 0);
11 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 [**?**] 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.

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

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

Single Path Cases: This transformation collects path predicates inside a region that lead to *non-nominal* exit point as an alternative to transition points [**?**]. 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 helps the region instantiation proceed ahead with exploring unexceptional behavior in the region separately from exceptional behavior and potentially reduces the branching factor in the multi-path region. We use this pass of transformation to identify non-nominal exit points, collect their path predicates and prune them away from the Ranger IR statement. The formal rules of this transformation is defined in rule $\texttt{single\_path}_1$ and $\texttt{single\_path}_2$, 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.

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.

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

Translation to Green: At this point Ranger region contains only compositional statements as well as assignment statements that might contain GSA expressions in them. Compositional statements are translated into conjunction, assignment statements are translated into Green 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.

4 EVALUATION

4.1 Experimental Setup

We implemented the above mentioned transformations as a wrapper around the Symbolic PathFinder [**?**] 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 the following five modes.

Mode 1: Java Ranger runs vanilla SPF without any path merging.

Mode 2: Java Ranger summarizes multi-path regions with a single exit point and instantiates them if they are encountered. 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 instantiation-time 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 the performance of Java Ranger. While our current implementation of Java Ranger cannot instantiate summaries with symbolic object and array references, it is capable of summarizing reads and writes to arrays with symbolic indices.

Mode 3: In addition to using summaries instantiated in mode 2, Java Ranger instantiates all multi-path region summaries that make method calls which have also been statically summarized. Method summaries, that have a single exit point in the form of a control-flow returning instruction, are inlined into the multi-path region summary based on the instantiation-time type of the method.

Mode 4: In addition to using summaries instantiated in mode 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.

Mode 5: In addition to using summaries instantiated in mode 4, Java Ranger converts multiple exit points that return control flow into a single control flow-returning exit point to allow multi-path regions to have more than one non-returning exit point.

We used the control-flow graph recovered by Wala [**?**] to bootstrap our static statement recovery process. While our static statement recovery was able to summarize thousands of regions in Java library code, many of these summaries could not be instantiated due to JPF's use of native peers [**?**]. To avoid these unnecessary

WBS	TCAS	replace	NanoXML	
1234	4509	3980	871	
Siena	Schedule2	PrintTokens2	ApacheCLI	MerArbiter
1551	722	1180	910	20639

Table 1: Static analysis time (msec) for all 9 benchmarks

instantiation failures and target our static statement recovery towards the benchmark code, we turned off statement recovery across a few Java library packages in Wala on all benchmarks.

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 was beneficial in reducing the number of times large formulas had to be passed to the solver.

4.2 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. [**?**] which also includes a translation of the Siemens suite to Java. (1) Wheel Brake System (WBS) [**?**] 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 [**?**]. 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 [**?**] 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 [**?**] 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. [**?**]. This benchmark consists of 268 classes, 553 methods, 4697 lines of code including the Polyglot framework.

We first ran each of these benchmarks using SPF with increasing number of symbolic inputs and obtained the most number of inputs with which SPF finished complete exploration of each benchmark within a 12 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 achieving complete path exploration of each benchmark. Next, we used the fastest mode of Java Ranger to check if it could explore the benchmark with even more symbolic inputs within the same 12 hour time budget. We report results from both of these evaluations for each benchmark in Figures 5, 6. Detailed experimental results are reported in Tables 2, 3 in a file named "results-tables.pdf" in the supplementary material included in our submission. A even more detailed set of results can be found in a file named "results-all.csv". Since Java Ranger relies on a prior static analysis to construct region summaries, we report the time taken to statically analyze all the code in each of the benchmarks in Table 1. We found that the total time required for static analyis is roughly proportional to the size of the benchmark. While such static analysis performance can cause Java Ranger to be slower on benchmarks with a small number of execution paths, we found that the cost of static analysis gets amortized as the number of execution paths increased.

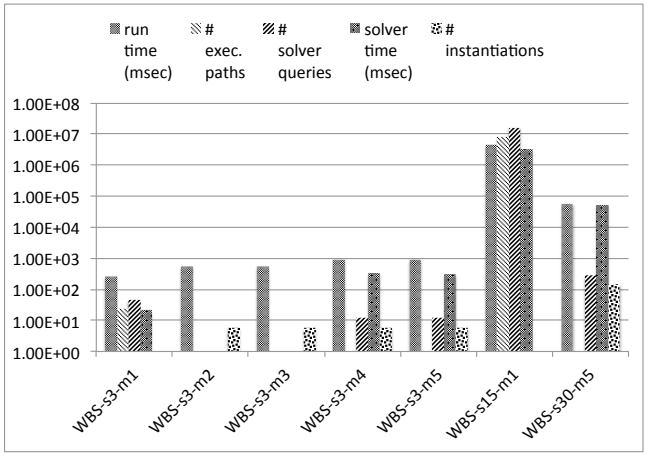
Figures 5, 6 show that Java Ranger achieves a significant speed-up over SPF with 5 (WBS, TCAS, NanoXML, ApacheCLI, MerArbiter) of the 9 benchmarks in terms of both running time and number of execution paths. Of the remaining four benchmarks, in the replace benchmark, Java Ranger achieves a modest reduction in execution time of 13% while reducing the number of execution paths by 37% in mode 2. In mode 5, while it reduces the number of execution paths by about 89%, it incurs an increase in execution time due to an increase in formula size caused due to an increase in the number of region instantiations in mode 5. On manually investigating the set of instantiated regions in replace, we found that the outputs of most regions were being branched on later in the code causing the benefit from more instantiations to be lost. Similar reasons cause Java Ranger to lose out on a reduction in running time while reducing the execution path count with the PrintTokens2 benchmark.

Java Ranger has the same performance as SPF on Siena and Schedule2 with a 3% overhead in running time that comes from Java Ranger's lookup of a region summary for every executed branch instruction and recording of instantiation-time metrics. We restrict our results with Siena to 6 symbolic inputs because 6 is the most number of symbolic inputs for which SPF finishes complete path exploration within a 12 hour time budget. On running Siena, the overhead of Java Ranger results from running into (1) 3,800,343 symbolic branch instructions that a region summary lookup and (2) 8,959,692 concrete branch instructions that Java Ranger could have summarized, had these branches been symbolic.

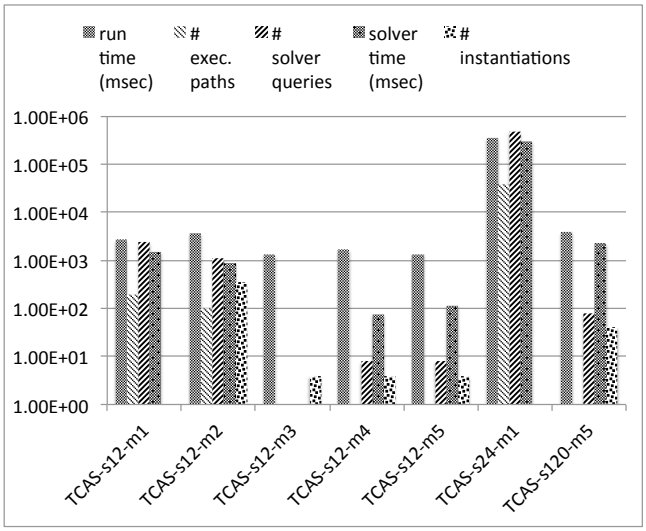
Java Ranger is able to summarize the entire step function of WBS and TCAS into a single execution path. In case of WBS, Java Ranger summarizes a multi-path region that consists of 9 branches. In case of TCAS, Java Ranger uses 28 method summaries in each step of TCAS, many of which summarize multiple return values into a single formula that represents all the feasible return values of the method. While SPF does not finish more than 5 steps of WBS and 2 steps of TCAS within 12 hours, Java Ranger finishes 10 steps of both benchmarks within 59 seconds and 5.6 seconds respectively.

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. With mode 5 of Java Ranger converting such multiple control-flow returning exit points regions into a single control-flow returning exit point, Java Ranger is able to use more than 27,000 method summaries when running NanoXML with 8 symbolic inputs and finish in 7.3 hours while vanilla SPF (which is the same as mode 1 of Java Ranger) times out in 12 hours.

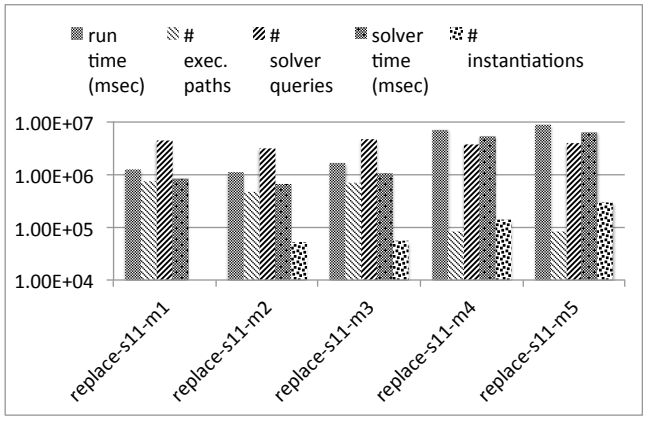
On the ApacheCLI benchmark, the most significant benefit is introduced in mode 2 of Java Ranger. Modes 4, 5 cause an increase in execution path count and running time but still give a significant reduction over mode 1. The increase in running time in modes 4 and 5 results from Java Ranger's use of the solver to check if the single-path cases in modes 4 and 5 are feasible. In the future, we plan to use a counter-example cache to reduce the use of a solver to check the feasibility of these single-path cases.



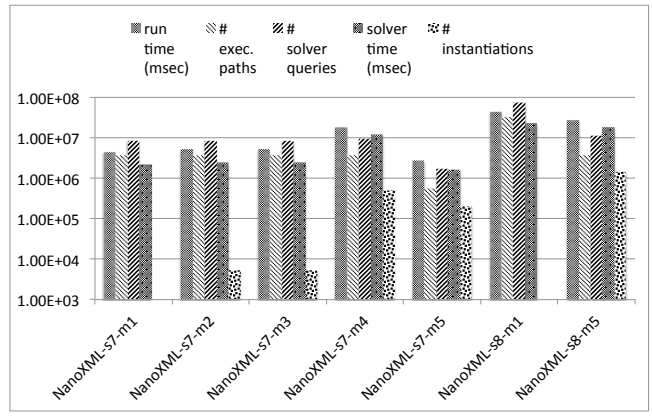
(a) Java Ranger explores WBS with one execution path in modes 2, 3, 4, and 5 regardless of the number of symbolic inputs. Java Ranger doesn't need to query the solver in modes 2, 3. WBS-s15-m1 times out in 12 hours.



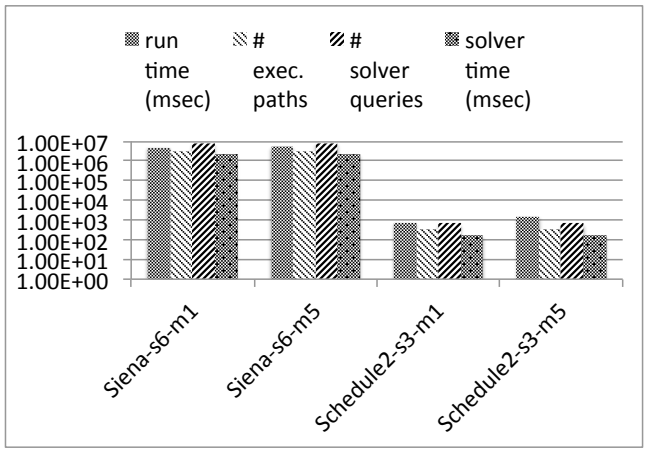
(b) Java Ranger summarizes TCAS in one execution path starting with mode 3 regardless of the number of symbolic inputs. TCAS-s24-m1 times out in 12 hours. In mode 3, Java Ranger uses 28 method summaries in every step of TCAS.



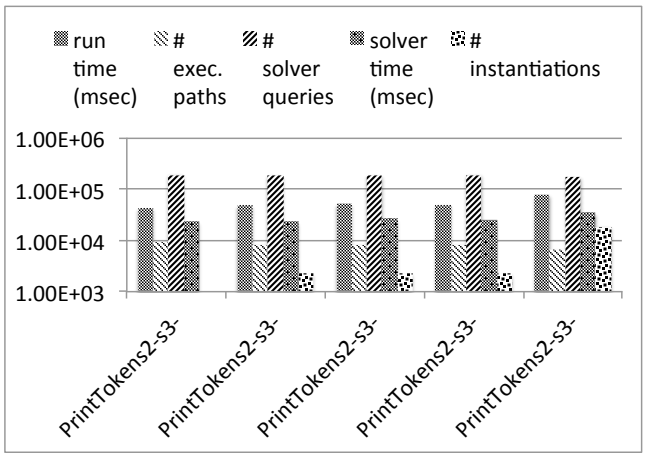
(c) In mode 2, Java Ranger reduces the execution path count in Replace by 37% from mode 1 with a modest decrease in running time of 13%. In modes 4, 5, Java Ranger reduces the execution path count by an overall 89% at the expense of more a larger path condition caused due to more region instantiations.



(d) In mode 5, Java Ranger reduces the execution path count in NanoXML with 7 symbolic inputs by 85% while reducing running time by 40%. This improvement is due to conversion of multiple control-flow returning exit points in multi-path regions into a single control-flow returning exit point in mode 5.

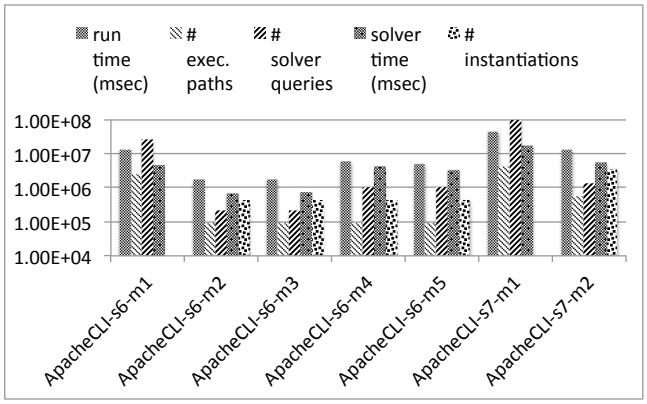


(e) Java Ranger introduces a 3% overhead in running time with Siena. The overhead is more with Schedule2 because of its short runtime in mode 1 and because Java Ranger needs about the same amount of time for its offline static analysis in mode 5.

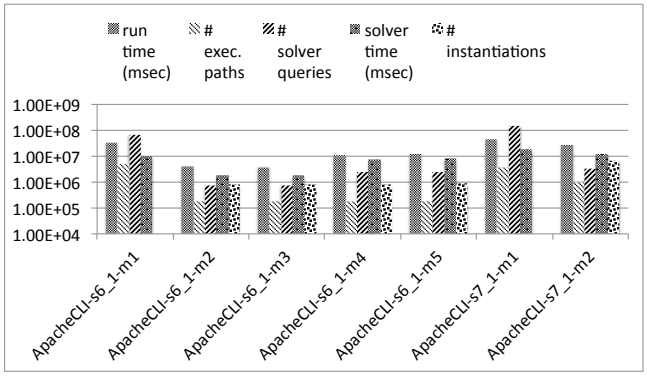


(f) While Java Ranger reduces the execution path count by 10% in modes 2, 3, 4, and by 27% in mode 5, this reduction comes at the expense of longer formulas sent to the solver without achieving a significant reduction in the number of solver queries.

Figure 5: Performance of Java Ranger on WBS, TCAS, Replace, NanoXML, Siena, Schedule, PrintTokens2 as shown by using a <benchmark-name>-s<number of symbolic inputs>-m<Java Ranger mode> format. Java Ranger runs vanilla SPF in mode 1.



(a) In mode 2 in ApacheCLI with 6 symbolic inputs, Java Ranger reduces execution path count by 96% while reducing running time by 86%. While vanilla SPF (mode 1 in Java Ranger) times out in 12 hours with 7 symbolic inputs, Java Ranger manages to finish in 3.6 hours in mode 2.



(b) The last input to ApacheCLI is different from all the other inputs because it controls whether ApacheCLI should stop on encountering a non-option input. The benefits of using Java Ranger with ApacheCLI are still evident if this input is made symbolic. In mode 2, Java Ranger reduces execution path count by 96% and running time by 88%. Vanilla SPF (mode 1 in Java Ranger) still times out in 12 hours with 8 symbolic inputs (including the `stopOnNonOption` input while Java Ranger manages to finish in 7.2 hours in mode 2.)

Figure 6: Performance of Java Ranger on ApacheCLI

In the MerArbiter benchmark, while the most significant benefit is introduced in mode 2 of Java Ranger, the benefit in terms of both execution path count and running time remains the same in modes 3, 4, and 5. All the multi-path regions that SPF (or mode 1 in Java Ranger) needs to branch on but are summarized by Java Ranger are small 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 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 464,000 instantiations with 7 steps of MerArbiter, with more than 319,000 instantiations needing more than 8 iterations of the fixed-point computation.

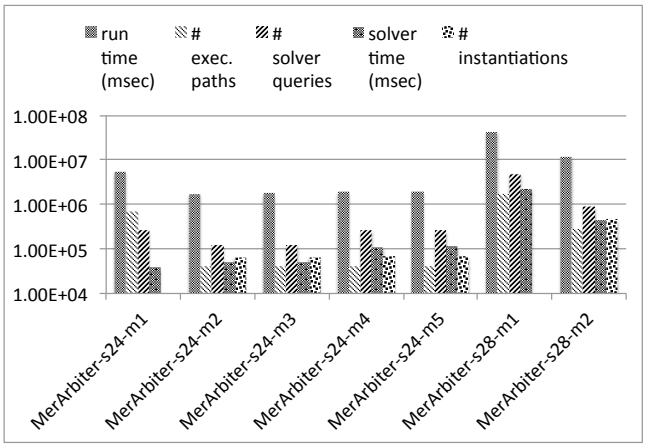


Figure 7: With 6 steps (24 symbolic inputs) of MerArbiter, Java Ranger reduces execution path count by 94% and running time by 68%. SPF times out in 12 hours with 7 steps (28 inputs), Java Ranger finishes in 3.4 hours in mode 2.

5 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 aggresively 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.

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 most useful applications of dynamic symbolic execution that is currently unsupported. We intend to extend Java Ranger towards test generation for merged paths in the future.

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 symbolic execution can prevent path explosion originating from standard libraries.

6 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.

7 APPENDIX

We report results from our evaluations for each benchmark in Tables 2 and 3

Benchmark Name	# sym inputs	mode	run time (msec)	static analysis time (msec)	#exec. paths	#solver queries	solver time (msec)	#inst. regions	#inst. methods	#inst.
WBS	3	1	263	0	24	46	21	0	0	0
WBS	3	2	569	1009	1	0	0	6	0	6
WBS	3	3	532	1234	1	0	0	6	0	6
WBS	3	4	920	1333	1	12	328	6	0	6
WBS	3	5	917	1355	1	12	311	6	0	6
WBS	15	1	4427660	0	7962624	15925246	3121444	0	0	0
WBS	30	5	57532	2230	1	280	54072	16	0	140
TCAS	12	1	2656	0	198	2338	1509	0	0	0
TCAS	12	2	3648	2988	98	1098	862	13	0	361
TCAS	12	3	1326	4509	1	0	0	4	28	4
TCAS	12	4	1674	4470	1	8	75	4	28	4
TCAS	12	5	1357	3364	1	8	114	4	28	4
TCAS	24	1	353066	0	39204	465262	296228	0	0	0
TCAS	120	5	3853	1840	1	80	2212	4	280	40
replace	11	1	1241296	0	757261	4317642	818462	0	0	0
replace	11	2	1080048	3390	477315	3125750	667165	27	0	51069
replace	11	3	1622018	3980	681673	4623560	1030076	25	17210	53938
replace	11	4	7027542	4062	82320	3757873	5338884	34	6339	139813
replace	11	5	8529211	4377	82320	3904498	6340147	40	6339	286767
NanoXML	7	1	4408426	0	3606607	8318716	2195506	0	0	0
NanoXML	7	2	5027498	680	3606476	8318577	2351930	2	0	5262
NanoXML	7	3	5021002	871	3606476	8318577	2336583	2	0	5262
NanoXML	7	4	17852489	1111	3599980	9292835	11787938	7	77156	492289
NanoXML	7	5	2663570	1406	553917	1680328	1619964	8	3241	203507
NanoXML	8	1	43200000	0	31835683	72954300	22040428	0	0	0
NanoXML	8	5	26365485	680	3642830	11024807	18055761	8	27273	1393422
Siena	6	1	4883384	0	2985984	7600684	2087250	0	0	0
Siena	6	5	5033783	1551	2985984	7600684	2092015	0	0	0
Schedule2	3	1	773	0	343	684	172	0	0	0
Schedule2	3	5	1557	722	343	684	172	0	0	0

Table 2: Java Ranger Performance on WBS, TCAS, Replace, NanoXML, Siena, and Schedule2

Benchmark Name	# sym inputs	mode	run time (msec)	static analysis time (msec)	#exec. paths	#solver queries	solver time (msec)	#inst. regions	#inst. methods	#inst.
PrintTokens2	3	1	41973	0	9074	188002	23080	0	0	0
PrintTokens2	3	2	47858	800	8184	182366	23179	2	0	2287
PrintTokens2	3	3	51571	1180	8184	182366	26022	2	0	2287
PrintTokens2	3	4	49842	1222	8184	186940	24538	2	0	2287
PrintTokens2	3	5	77124	1129	6612	168350	33783	15	0	18230
ApacheCLI	6	1	12987647	0	2515866	26577992	4393629	0	0	0
ApacheCLI	6	2	1731243	611	93328	208980	693509	4	0	420736
ApacheCLI	6	3	1803509	910	93328	208980	744353	4	0	420736
ApacheCLI	6	4	5858874	907	93328	1050452	4090707	4	0	420736
ApacheCLI	6	5	5031006	1083	93328	1050452	3307654	5	0	439072
ApacheCLI	7	1	43181000	0	4114753	98058899	17579650	0	0	0
ApacheCLI	7	2	13194514	546	555143	1301110	5480163	4	0	3610342
ApacheCLI	6+1	1	31886455	0	4839812	64432586	9470833	0	0	0
ApacheCLI	6+1	2	3838606	594	171818	756030	1770804	4	0	813694
ApacheCLI	6+1	3	3677573	803	171818	756030	1716274	4	0	813694
ApacheCLI	6+1	4	10474434	841	171818	2383418	7287933	4	0	813694
ApacheCLI	6+1	5	11636593	742	171818	2385043	8010246	5	0	847837
ApacheCLI	7+1	1	43191000	0	3725921	144990567	17875860	0	0	0
ApacheCLI	7+1	2	26186160	466	986522	3095212	11608660	4	0	6854478
MerArbiter	24	1	5456648	0	707972	271571	38138	0	0	0
MerArbiter	24	2	1729347	11734	40752	126384	50502	34	0	66514
MerArbiter	24	3	1762671	20639	40752	126384	48857	34	0	66514
MerArbiter	24	4	1949940	19997	40752	258672	109488	35	0	68191
MerArbiter	24	5	1990565	21110	40752	258672	114509	35	0	68191
MerArbiter	28	1	43191000	0	1731221	4852988	2258472	0	0	0
MerArbiter	28	2	12074874	13361	276144	871304	439035	34	0	463868

Table 3: Java Ranger Performance on PrintTokens2, ApacheCLI with the last input made made concrete and symbolic respectively, and MerArbiter