Java Ranger: Static Regions for Efficient Symbolic Execution of Java

Anonymous Author(s)*

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, 16 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 [10, 20], 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 [16, 26], equivalence checking [25, 28], vulnerability finding [30, 31], and protocol correctness checking [32]. Symbolic execution tools are available for many languages, including CREST [5] for C source code, KLEE [6] for C/C++ via LLVM, JDart [22] and Symbolic PathFinder (SPF) [24] for Java, and S2E [9], FuzzBALL [3], and `angr` [30] for binary code.

Although symbolic analysis is a popular technique, scalability is a substantial challenge for many applications. In particular it can suffer from a *path explosion*: complex software has exponentially many execution paths, and baseline symbolic execution techniques that explore one path at a time are unable to cover all paths. Dynamic state merging [17, 21] 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 [2] 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 [2], constructing bounded static code regions was shown to allow symbolic execution to find more bugs, and achieve more node and path coverage, when implemented at the X86 binary level for compiled C programs. This motivates us to investigate using static regions for symbolic execution of Java software (at the Java bytecode level).

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

Not being able to summarize such dynamically dispatch methods can lead to poor performance for 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 region as an argument and can return a region that may need to be further interpreted. In our experiments, we demonstrate exponential speedups on benchmarks (in general, the more paths contained within a program, the larger the speedup) over the unmodified Java SPF tool using this approach.

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

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

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

on region summaries. (3) It makes the construction of region summaries more accessible to users of the symbolic execution tool, thereby making path merging more useful to end-users.
In this paper, we present Java Ranger, an extension of Symbolic PathFinder, that computes such region summaries over a representation we call Ranger IR. Ranger IR has support for inlining method summaries and for constructing SSA form for heap accesses. It also proposes Single-Path Cases as an alternative to multiple transition points as defined by Avgerinos et al. [2].

This paper extends our initial investigations of Java veritesting reported in a paper at the 2017 JPF workshop [29]. Our workshop paper motivated some of the ways in which Java veritesting is different from veritesting for binary code and the value that veritesting could provide, but it did not describe an end-to-end automated implementation. The present paper describes a number of conceptual improvements that are new since the workshop paper, including the Ranger IR, single-path cases, and the details of our higher-order region approach. We have also made a number of architectural changes, such as switching from Soot to Wala for static analysis and using Green for formula representation, and we have integrated Java Ranger as an extension to SPF and evaluated it in several case studies.

1.1 Motivating Example

Consider the example of Java code shown in Figure 1. The example is used to count number of words where characters are represented with 1 and 0 represents space. The `list` object refers to an `ArrayList` of 200 `Integer` objects which have an unconstrained symbolic integer as a field. Checking the first element to be a zero or one introduces a symbolic branch, that requires the symbolic execution to explore both paths. Therefore loop checking for each indexed entry in `list` introduces a branch, which has both sides feasible and which requires the symbolic execution to explore both paths again.
Performing this check over the entire `list` makes symbolic execution need 2^{100} execution paths to terminate (assuming `list` has 200 entries with every even-indexed entry pointing to a new unconstrained symbolic integer). A simple way to avoid this path explosion is to merge the two paths arising out of the `list.get(i) == 0` branch. Such path merging requires us to compute a summary of all behaviors arising on both sides of the branch of the if-statement inside the for loop. If we can construct such a summary beforehand, our symbolic executor can instantiate the summary by reading in inputs to the summary from the stack and/or the heap, and writing outputs of the summary to the stack and/or the heap. Unfortunately, constructing such a summary for this simple region is not straightforward due to the call to `list.get(int)` which is actually a call to `ArrayList<Integer>.get(int)` (`java.util.List<E>.get(int)` is abstract and does not have an implementation). `ArrayList<Integer>.get(int)` internally does the following:

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

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

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

2 RELATED WORK

Path explosion is a major cause of scalability limitations for symbolic execution, so an appealing direction for optimization is to combine the representations of similar execution paths, which we refer to generically as *path merging*. If a symbolic execution tool already concurrently maintains objects representing multiple execution states, a natural approach is to merge these states, especially ones with the same control-flow location. Hansen et al. [17] explored this technique but found mixed results on its benefits. Kuznetsov et al. [21] 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. [27], 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. [2] and dubbed "veritesting" because it pushes symbolic execution further along a continuum towards all-paths techniques used in verification. A veritesting-style technique is a convenient way to add path merging to a symbolic execution system that maintains only one execution state, which is one reason

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

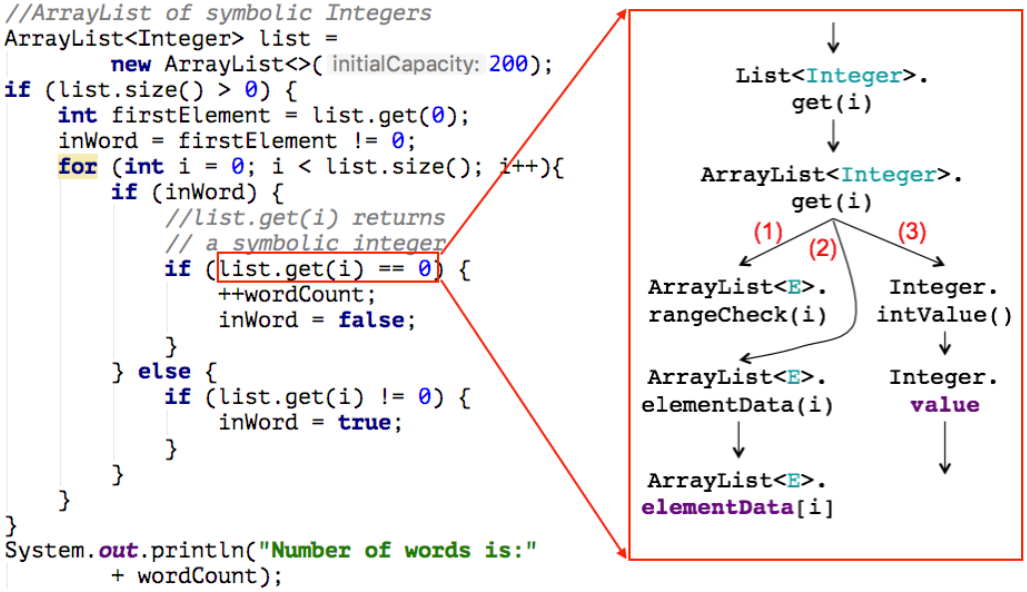


Figure 1: An example demonstrating the need for using a multi-path region summary

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

The veritesting approach has been integrated with another binary level symbolic execution engine named angr [30]. However angr's authors found that their veritesting implementation did not provide an overall improvement over their dynamic symbolic execution baseline: though veritesting allowed some new crashes to be found, they observed that giving more complex symbolic expressions slowed down the SMT solver enough that total performance was degraded. We have also observed complex expressions to be a potential cost of veritesting, but we believe that optimizations of the SMT solver interface and potentially heuristics to choose when to use static regions can allow them to be a net asset.

The way that Java Ranger and similar tools statically convert code regions into formulas is similar to techniques used in verification. In the limit where all relevant code in a program can be summarized, such as with WBS and TCAS-SR in Section 4, Java Ranger performs similarly to a bounded symbolic model checker for Java. SPF and Java Ranger build on Java Pathfinder (JPF) [34], 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 [14], ESC/Java2 [12], and OpenJML [11], though these tools target static error checking and verification of annotated specifications.

Perhaps the most closely related Java model checking tool is JBMC [13], 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. [4] 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 [15]. Like Java Ranger's function summaries, SMART summarizes the behavior of a function in isolation from its calling context so that the summary can be

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

3 TECHNIQUE

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

Algorithm 1: Ranger general pesudo-code

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

Figure **??** describes the main steps that Java Ranger deploys to summarize the region. Intuitively, Java Ranger starts with a symbolic multi-path region expressed in Ranger grammar 2 (described in section 3.1). Java Ranger then proceeds as follows, first it performs early return transformation, by eliminating eliminating return statements, this is followed by alpha-renaming on variables in the region. Then Java Ranger enters two fix-point loops, the inner loop terminates when no further changes is detected on the region after repeatedly applying substitution, high-order, fields, arrays, reference alpha-renaming and simplification transformations. can be made on the region. The outer loop ensures that at least one step of high order region was attempt without further changes before it declares reaching a fixed point.

After reaching a fixed point, single path cases are determined, then flattening the IR of the region to a sequence of assignment statements followed by a transformation that transforms the region into the corresponding solver syntax using Green. Finally if all transformations were successful then Java Ranger populates the output to SPF and advances it to the right position, otherwise it aborts.

$\langle stmt \rangle$ ::= $\langle stmt \rangle$; $\langle stmt \rangle$ | $\langle exp \rangle$:= $\langle exp \rangle$ | skip | x.$\langle stmt \rangle$
| $\langle exit\_stmt \rangle$ | if $\langle exp \rangle$ then $\langle stmt \rangle$ else $\langle stmt \rangle$
| invoke(τ, $\langle exp \rangle$, $\langle exp \rangle$) | exit

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

$\langle exp \rangle$ ::= $\langle val \rangle$ | $\langle var \rangle$ | $\langle exp \rangle$ op_b $\langle exp \rangle$ | op_u $\langle exp \rangle$
| get_field($\langle exp \rangle$, $\langle exp \rangle$) | put_field($\langle exp \rangle$, $\langle exp \rangle$)
| array_load($\langle exp \rangle$, $\langle exp \rangle$) | array_store($\langle exp \rangle$, $\langle exp \rangle$, $\langle exp \rangle$)
| gamma($\langle exp \rangle$, $\langle exp \rangle$, $\langle exp \rangle$) | $\langle exp \rangle$.$\langle exp \rangle$

$\langle op_b \rangle$ ::= + | − | ∗ | ÷ | & | bitwise-or | ⊕ | % | == | ≠ | ≤ | ≥ | && | logical-or | >| <| ≪ | ≫ | ⋙

$\langle op_u \rangle$ ::= − | ~

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

$\langle var \rangle$ ::= ID_$\mathbb{N}$

Figure 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
	$\Gamma_s : ref \rightarrow$	τ
Region-map	$\Psi : \tau \rightarrow$	$x.s$
	$\Theta : var \rightarrow$	loc
Single-path-constraints	$\Sigma : \{exp\}$	
Early-return-constraints	$\Sigma_{ret} : x_{ret} \rightarrow$	exp

Figure 3: Environments used in Ranger.

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

3.1 Statement Recovery

3.1.1 Statement Recovery Setup. To bound the set of code regions we analyze, we start by specifying a method M in a configuration file. Next, we construct a set containing only the class C that contains M. We then get another set of classes, C', such that every class in C' has at least one method that was called by a method in a class in C. This step which goes from C to C' discovers all the classes at a call depth of 1 from C. We continue this method discovery process up to a call depth of 2. While we can increase the call depth in our method discovery process, we found that summarizing arbitrary code regions with more than 2 calls deep, did not lead to practically useful region summaries. After obtaining a list of methods, we computed static summaries of regions in these methods and method summaries as explained in Section 3.1. After computing static region and method summaries, we process them as a sequence of transformations described in the next section 3.3 and summarized in Figure **??**.

3.1.2 IR-Statement Recovery. In this step, a Java Ranger is created from the corresponding CFG. Since regions of interest for our technique are bounded by the branch and meet of a given acyclic subgraph. The intuition is that path explosion during execution of loops is driven by conditional logic within the loop, rather than the loop itself. Starting from an SSA form, the first transformation recovers a tree-shaped AST for the subgraphs of interest. While this step is not strictly necessary, it substantially simplifies subsequent transformations.

The algorithms are similar to those used for those used for decompilation [37] but with slightly different goals:

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

Starting from an initial SSA node, the algorithm first finds the immediate post-dominator of all *normal* control paths, that is, paths that do not end in an exception or return instruction. It then looks for nested self-contained subgraphs. If for any graph, the post-dominator is also a predecessor of the node, we consider it a loop and discard the region.

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

```
1  if ((== x58 0 )) {
2     x50 = invoke(ArrayList.get(I)Ljava.lang.Object,x9, x59)
3     x51 = checkcast (java.lang.Integer, x50)
4     x53 = invoke(Integer.intValue()I, x51)
5     if ((! (== x53 0 ) )) {
6        skip;
7     } else { skip; }
8  } else {
9     .....
10 }
11 x54 := (Gamma x58==0 (Gamma !(x53==0) x57 x57) (Gamma
       !(x47!=0) x48 x57));
12 x55 := (Gamma x58==0 (Gamma !(x53==0) 1 x58) (Gamma
       !(x47!=0) 0 x58));
```

The outcome of this step is a statement in Java Ranger. For example our example 1 is recovered from the Java Bytecode to obtain the Ranger statement in In the following sections we will discuss the various parts of this statement to show the effect of various transformations.

The grammar of Java Ranger is defined in 2. Basically the grammar is composed of statements $stmt$ and expressions exp. Statements in Java Ranger contains: sequential composition ($stmt; stmt$), assignment ($exp = exp$), a skip, parametric statements $x.stmt$, exit-statements $exit$, which determine single path regions and they are $new\ \tau$, $throw\ exp$ and $return\ exp$, if-then-else, method invocation $invoke(exp, exp, exp)$ and especial $exit$ statement to bookmark exit point of a region/statement.

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

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

3.2 Region Definition

Once the statement of a multi-path region has been recovered, its corresponding environment is populated. This includes identifying region boundary and creating local variable inputs, outputs, type, and stack slot tables for the region. The region boundary is used to identify boundaries of the region w.r.t local variables. This is used later to constrain the computation and population of Ranger IR environment tables. For example, the local variable input table is populated with first *use* in the region boundary that map to a given stack slot. The output table is populated with the last *def* of a local variable at merge point of the region. The local variable type table is populated for all variables that lay within the boundaries of the region, this is initially done by inquiring the static analysis framework, WALA [35] but is later changed by inferring types of local variables at instantiation and during type propagation transformation 3.3.

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

Formally we define the following structures:

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

type-map: Java Ranger and SPF value map Γ_r and Γ_s that maps

vars or references, to types τ.
region-map: Java Ranger defines a map Ψ from type τ to a parametric statement $x.s$ that defines the parameteric statement region in Java Ranger.
Single-Path-Constraint-List: A list Σ of single-path constraints.
Early-Return-Constraint-map: A map Σ_{ret} from early-return vars x_{ret} to early-return constraints.

3.3 Instantiation-time Transformations

Early Return Transformation: In this transformation, Java Ranger collects that conditions for branches that have early return in them. In figure 4 we show the rule of having an early return on the else side $_{\texttt{early-return}_1}$ where neither inner statements s_1 or s_2 are conditional with early return. In this case, the statement is transformed to appending an assignment statement to the early return result to a new early return var x_{ret}, also the condition under which the early return value occurs is added to the early return environment $\Sigma_{ret} \cup (x_{ret}, e_2)$

Rule $_{\texttt{early-return}_2}$ describes the situation where both the statements $s1$ and $s2$ contains inner conditions with early return statements. In this case a new result variable is created x_{ret} and assigned to the gamma expression $x_{ret} = gamma((c1 \wedge e_1), e_1', e_2')$ describing conditions and values assigned under them. In addition the context of early return variables-conditions is updated.

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

We do not define this transformation formally due to space, but it is basically straight forward by generating a unique prefix for each variable and updates the new name in the statement as well as all the contexts in which it appears. For example, in our example line 2, gets renamed to:

```
1   x50$1= invoke(ArrayList.get(I)Ljava.lang.Object,x9$1,
        x59$1);
```

Local Variable Substitution Transformation: During this transformation we eagerly bring in all dynamically known constant values, symbolic values and references from stack slots into the region for further processing.
More formally, rule $_{\texttt{substitution}}$ given a parametric statement $x.s$, and if x was mapped to a location l then its dynamic, i.e., concrete or symbolic (v, e), substitute x in s. For example, the reference of the arraylist can be resolved (in this case it was resolved to object reference 375). It is then substituted by the substitution transformation, thus line 2 becomes:

```
1   x50$1= invoke(ArrayList.get(I)Ljava.lang.Object,375,
        x59$1);
```

Higher-order Regions Transformation: This transformation is initiated when a method invocation is encountered during local variable substitution. At this point, we perform three steps. (1) the region that corresponds to the called method is retrieved and alpha renaming of Ranger IR variables corresponding to local variables is applied on it. (2) Ranger IR expressions that correspond to the actual parameters are evaluated and used to substitute the formal parameters by repeatedly applying local variable substitution transformation over the method region. (3) When no more higher-order regions can be inlined, the resulting substituted method region is inlined into the outer region.
Formally, rule $_{\texttt{high-order}_1}$ describes the inlining process for methods that has no return values for a concrete reference, and $_{\texttt{high-order}_2}$ describes inlining of methods that have a single return values also for concrete reference. For non-concerete reference, and after reaching a fixed point, the updated type table (using type propagation) is used to get the type of the variable and therefore the corresponding parametric region statement. It must be noted that methods with multiple returns, should have been normalized out with the early-return transformation discussed previously. Note the rules describes the support for the jvm invoke virtual instruction, however Java Ranger supports as well invoke static and invoke interface.

For $_{\texttt{higho}\texttt{rder-1}}$ initially after evaluating the reference of the invoke by either finding it concerte value through substitution or obtaining it from the updated type table, the type is used to retrieve the parametric region for the method being invoked. Substitution of the parameters then follows and direct inlining occurs. In rule $_{\texttt{high-order}_2}$, inlining of the return value translates to an assignment statement.

For example, regions that defines $ArrayList.get(I)java.lang.Object$ is discovered and inlined with the original region to become:

```
1  [] = invoke < Primordial, Ljava/util/ArrayList,
       rangeCheck(I)V >[w1$3, w2$3]
2
3  [w6$3] = invoke < Primordial, Ljava/util/ArrayList,
       elementData(I)Ljava/lang/Object; >[w1$3, w2$3]
4  skip;
5  ~earlyReturnResult$3 := w6$3;
6  \end{lslisting}
7
8  \begin{lstlisting}
9  w4 = get(w1.< Primordial, Ljava/util/ArrayList, size,
       <Primordial,I> >)
10 if ((! (< w2 w4 ) )) {
11   new Instruction
12
13 [w7] = invoke < Primordial, Ljava/util/ArrayList,
       outOfBoundsMsg(I)Ljava/lang/String; >[w1, w2]
14
15 [] = invoke < Primordial,
       Ljava/lang/IndexOutOfBoundsException,
       <init>(Ljava/lang/String;)V >[w5, w7]
16   Throw Instruction
17 } else {
18   skip;
19 }
20  return w-1
```

Field References SSA form: The field references transformation translates reads and writes of fields in Java bytecode into corresponding Ranger IR statements. In order to translate all field

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

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

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

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

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

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

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

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

Figure 4: Evaluation Rules for Ranger Transformations

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) [23]. Each merged field access variable has its own path and global subscripts and represents the output of the region into its field. The path subscript helps us resolve read-after-write operations on the same execution path and find the latest write into a field on an execution path. The global subscript helps us distinguish between field accesses across multiple execution paths.

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

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

Simplification of Ranger IR: The Ranger IR constructed by earlier transformations computes exact semantics of all possible behaviors in the region. Representation of such semantics as a formula can often lead to unnecessarily large formulas, which has the potential to reduce the benefits seen from path merging [30]. For example, if an entry in an array is never written to inside a region, the array reference transformation can still have an array output for that entry that writes a new symbolic variable into it. The region summary would then need to have an additional constraint that makes the new symbolic variable equal the original value in that array entry. Such conjuncts in the region summary can be easily eliminated with constant propagation, copy propagation, and constant folding [1]. Ranger IR also has statement and expression classes that use a predicate for choosing between two statements (similar to an `if` statement in Java) and two sub-expressions (similar to the C ternary operator) respectively. When both choices are syntactically equal, the predicated statement and expression objects can be substituted with the statement or expression on one of their two choices. Such statements and expressions were simplified away to use one of their two choices. Ranger IR performs these two simplifications on such predicated statements and expressions along with constant folding, constant propagation, and copy propagation.

Single Path Cases: This transformation collects path predicates inside a region that lead to *non-nominal* exit point. This is an alternative approach to that was presented in [2]. In our work we define non-nominal exit point to be points inside the region that either define exceptional behavior or involve behavior that we cannot summarize, i.e, object creation and throw instructions. The intuition here is that, we want to maximize regions that Java Ranger can summarize, even if the summarization is only partial. We use this pass of transformation to identify such points, collect their path

predicates and prune them away from the Ranger IR statement. The formal rules of this transformation is defined in rule $\texttt{single\_path}_1$ 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, which Java Ranger had not summarized.

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

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

3.4 Checking Correctness Of Region Summaries

The Ranger IR computed as a result of performing the transformations described in Figure **??** should correctly represent the semantics of the summarized region. If it does not, then using the instantiated region summary can cause symbolic exploration to explore the wrong behavior of the subject program. We checked the correctness of our instantiated region summaries by using equivalence-checking as defined by Ramos et al. [25]. We designed a test harness that first executes the subject program with a set of symbolic inputs using SPF and capture the outputs of the subject program. Next, the test harness executes the same subject program with the same set of symbolic inputs using Java Ranger and capture the outputs of the subject program once again. Finally, the test harness compares outputs returned by symbolic execution with SPF and Java Ranger. If the outputs do not match, then a region summary used by Java Ranger did not contain all the semantics of the region it summarized. We symbolically execute all execution paths through this test harness. If no mismatch is found between outputs on any execution path, we conclude that all region summaries used by Java Ranger must correctly represent the semantics of the regions they summarized. We performed correctness-checking on all results reported in this paper.

4 EVALUATION

4.1 Experimental Setup

We implemented the above mentioned transformations as a wrapper around the Symbolic PathFinder [24] 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 enabled.

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 multi-path regions to have more than one exit point. These exit points take the form of new object allocations, exceptional behavior present in the region, and method invocations for which no summaries are present.Soha to double-check this statement

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 along with using single-path cases to allow multi-path regions to have more than one non-returning exit point.

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

We used the control-flow graph recovered by Wala [35] 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 [19]. To avoid these unnecessary 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. [36] which also includes a translation of the Siemens suite to Java. (1) Wheel Brake System (WBS) [38] 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 [18]. 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 [7] 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 [33] 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. It was originally modeled in Simulink/Stateflow and automatically translated into Java using the Polyglot framework. 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.

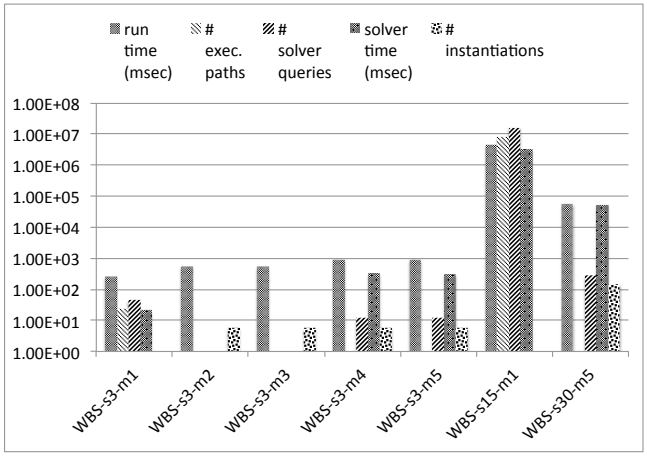
We first ran each of these benchmarks using SPF with increasing number of symbolic inputs and obtained the most number of symbolic inputs with which SPF finished complete exploration of each benchmark within a 12 hour time budget. We then ran each benchmark with this number of symbolic inputs with Java 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. 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 in each benchmark.

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.

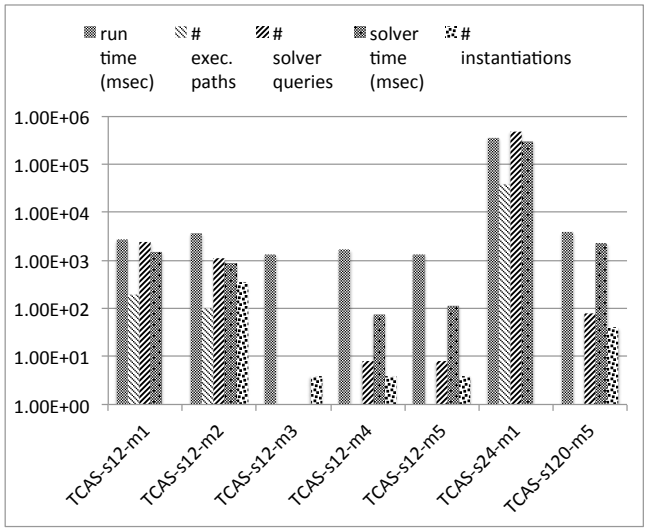
Java Ranger has the same performance as SPF on Siena and Schedule2 with a 3% overhead in running time that comes from Java Ranger's lookup of a region summary for every executed branch instruction and recording of instantiation-time metrics. We restrict our results with Siena to 6 symbolic inputs because 6 is the most number of symbolic inputs for which SPF finishes complete path exploration within a 12 hour time budget. On running Siena with 6 symbolic inputs, the overhead of Java Ranger results from it running into (1) 3,800,343 symbolic branch instructions that required Java Ranger to lookup a region summary for it 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 feasible return values of a method 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.

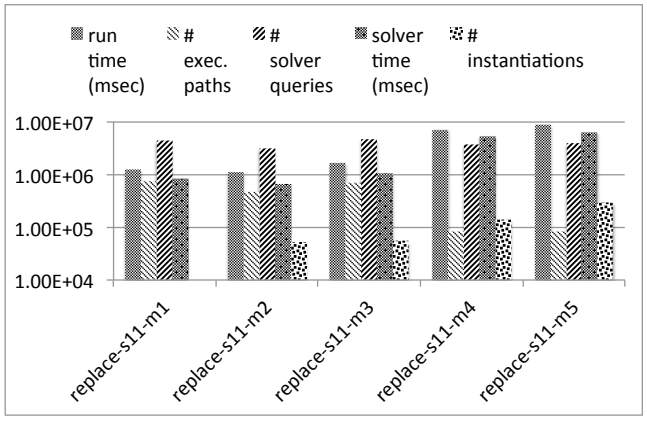
On the NanoXML benchmark, Java Ranger finds several opportunities for execution path count reduction on account of the NanoXML benchmark having several methods with multi-path



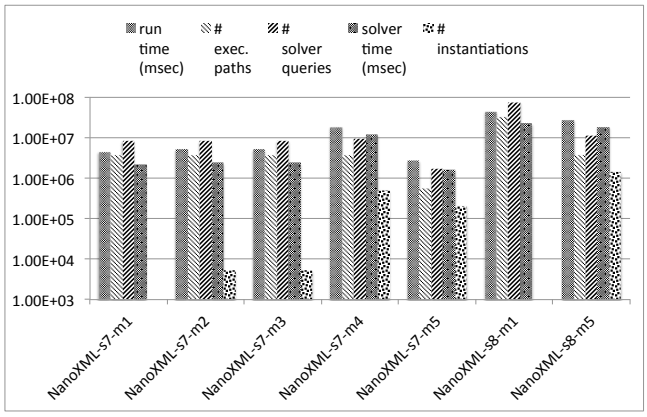
(a) Java Ranger explores WBS with one execution path in modes 2, 3, 4, and 5 regardless of the number of symbolic inputs. Java Ranger doesn't need to query the solver in modes 2, 3. WBS-s15-m1 times out in 12 hours.



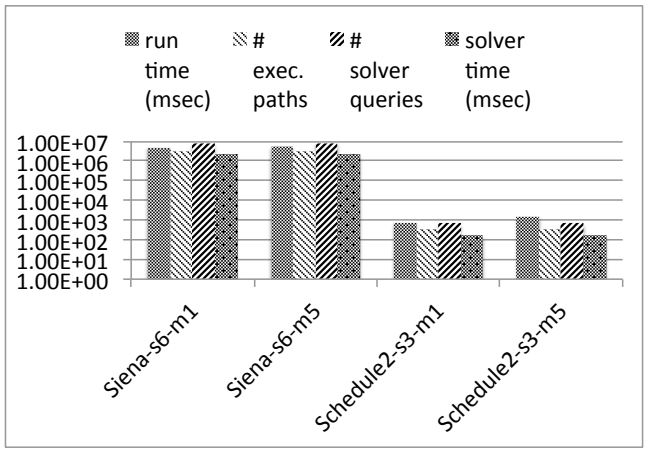
(b) Java Ranger summarizes TCAS in one execution path starting with mode 3 regardless of the number of symbolic inputs. TCAS-s24-m1 times out in 12 hours. In mode 3, Java Ranger uses 28 method summaries in every step of TCAS.



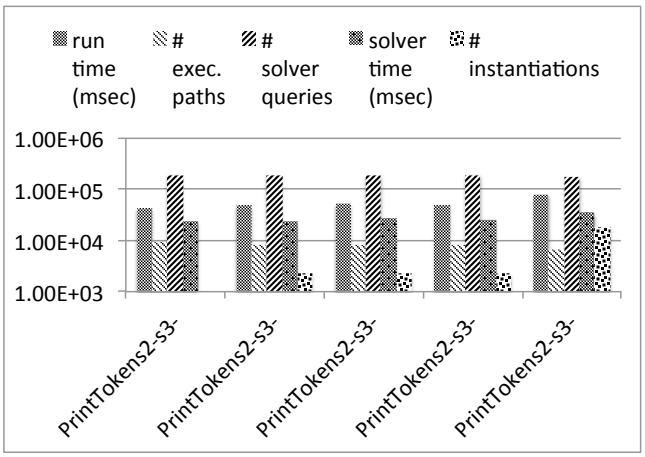
(c) In mode 2, Java Ranger reduces the execution path count in Replace by 37% from mode 1 with a modest decrease in running time of 13%. In modes 4, 5, Java Ranger reduces the execution path count by an overall 89% at the expense of more a larger path condition caused due to more region instantiations.



(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.

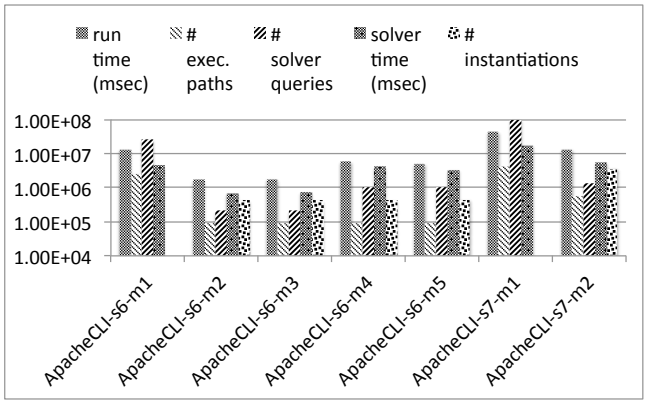


(e) Java Ranger introduces a 3% overhead in running time with Siena. The overhead is more with Schedule2 because of its short runtime in mode 1 and because Java Ranger needs about the same amount of time for its offline static analysis in mode 5.

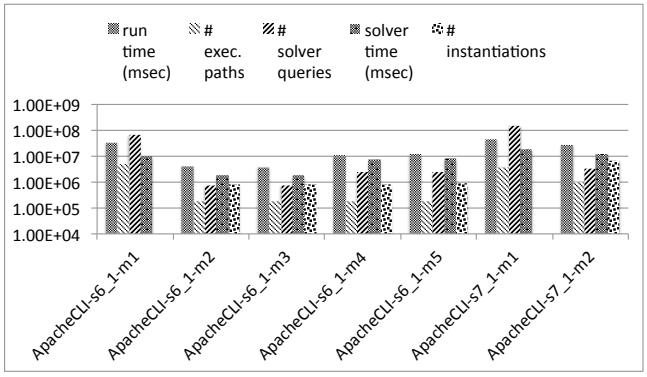


(f) While Java Ranger reduces the execution path count by 10% in modes 2, 3, 4, and by 27% in mode 5, this reduction comes at the expense of longer formulas sent to the solver without achieving a significant reduction in the number of solver queries.

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

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 achieve complete path exploration in 7.3 hours while vanilla SPF (which is the same as mode 1 of Java Ranger) times out.

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 use of the solver to check if the single-path cases that may be present in the multi-path region 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.

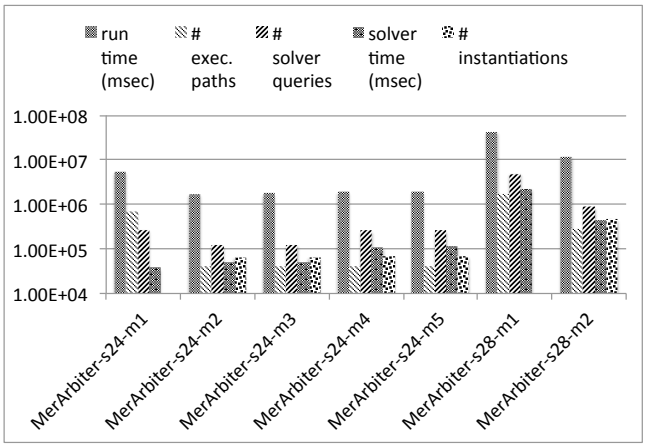


Figure 7: With 6 steps (24 symbolic inputs) of MerArbiter, Java Ranger reduces execution path count by 94% and running time by 68%. While vanilla SPF (mode 1 in Java Ranger) times out in 12 hours with 7 steps (28 symbolic inputs), Java Rangerfinishes in 3.4% hours in mode 2.

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.

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

REFERENCES

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

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

[3] Domagoj Babic, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. 2011. Statically-directed dynamic automated test generation. In *Proceedings of the 20th International Symposium on Software Testing and Analysis (ISSTA)*. 12–22. http://doi.acm.org/10.1145/2001420.2001423

[4] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3, Article 50 (2018).

[5] Jacob Burnim and Koushik Sen. 2008. Heuristics for Scalable Dynamic Test Generation. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 443–446. https://doi.org/10.1109/ASE.2008.69

[6] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. 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. 209–224. http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[7] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. 2000. Achieving Scalability and Expressiveness in an Internet-scale Event Notification Service. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '00)*. ACM, New York, NY, USA, 219–227. https://doi.org/10.1145/343477.343622

[8] Sang Kil Cha, T. Avgerinos, A. Rebert, and D. Brumley. 2012. Unleashing Mayhem on Binary Code. In *2012 IEEE Symposium on Security and Privacy*. 380–394.

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

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

[11] David R. Cok. 2011. 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*. 472–479.

[12] David R. Cok and Joseph Kiniry. 2004. 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*. 108–128.

[13] Lucas C. Cordeiro, Pascal Kesseli, Daniel Kroening, Peter Schrammel, and Marek Trtík. 2018. 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*. 183–190.

[14] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2002. 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*. 234–245.

[15] Patrice Godefroid. 2007. 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*. 47–54.

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

[17] Trevor Hansen, Peter Schachte, and Harald Søndergaard. 2009. 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*. 76–92.

[18] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. 1994. Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In *Proceedings of 16th International Conference on Software Engineering*. 191–200. https://doi.org/10.1109/ICSE.1994.296778

[19] Java PathFinder team. [n. d.]. Model Java Interface. https://github.com/javapathfinder/jpf-core/wiki/Model-Java-Interface. ([n. d.]). Accessed: 2019-01-18.

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

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

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

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

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

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

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

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

[28] Vaibhav Sharma, Kesha Hietala, and Stephen McCamant. 2018. Finding Substitutable Binary Code By Synthesizing Adaptors. In *11th IEEE Conference on Software Testing, Validation and Verification (ICST)*.

[29] Vaibhav Sharma, Michael W. Whalen, Stephen McCamant, and Willem Visser. 2018. Veritesting Challenges in Symbolic Execution of Java. *SIGSOFT Softw. Eng. Notes* 42, 4 (Jan. 2018), 1–5. https://doi.org/10.1145/3149485.3149491

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

[31] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution.. In *NDSS*, Vol. 16. 1–16.

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

[33] The Apache Software Foundation. [n. d.]. Commons CLI – Homepage. https://commons.apache.org/proper/commons-cli/. ([n. d.]). Accessed: 2018-11-16.

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

[35] Wala [n. d.]. WALA. http://wala.sourceforge.net/wiki/index.php/Main_Page. ([n. d.]). Accessed: 2018-11-16.

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

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

[38] Guowei Yang, Suzette Person, Neha Rungta, and Sarfraz Khurshid. 2014. Directed incremental symbolic execution. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 1 (2014), 3.

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

7 APPENDIX

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

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

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