Java Ranger: Static Regions for Efficient Symbolic Execution of Java

Vaibhav Sharma[1⋆], Soha Hussein[1⋆], Michael W. Whalen[2], Stephen McCamant[1], and Willem Visser[3]

[1] University of Minnesota, Minneapolis, MN, United States of America
vaibhav@umn.edu, husse200@umn.edu, mccamant@cs.umn.edu
[2] Amazon Web Services, mww@amazon.com
[3] University of Stellenbosch, Stellenbosch, South Africa
wvisser@cs.sun.ac.za

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 veritesting regions; we introduce a *higher-order* veritesting 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. More formally, we describe our veritesting algorithms as syntax-directed transformations of a structured intermediate representation, which highlights their logical structure. 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 veritesting greatly reduces the search space of Java symbolic execution benchmarks, while our expanded capabilities provided a significant further improvement.

1 Introduction

needs rewriting, assigned to anyone who can find the time to write it Our primary contributions to multi-path region summarization or path merging is (1) creation of an SSA-form IR to represent region summaries, (2) representing stack and heap accesses in Java programs in an IR, (3) extending path merging to include method summaries, (4) proposing the Single-Path case as an alternative to transition points as defined by the veritesting paper

⋆ These authors were equal contributors to this work

Symbolic execution is a popular analysis technique that performs non-standard execution of a program: data operations generate formulas over inputs, and the branch constrains along an execution path are combined into a predicate. Originally developed in the 1970s [11, 9], 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 many application of symbolic execution include test generation [10, 17], equivalence checking [16, 19], vulnerability finding [21, 20], and protocol correctness checking [22]. Symbolic execution tools are available for many languages, including CREST [5] for C source code, KLEE [6] for C/C++ via LLVM, JDart [13] and Symbolic PathFinder [15] for Java, and S2E [8], FuzzBALL [3], and angr [20] for binary code. More here...explain the 'ecosystem' - tools for different languages: KLEE, FuzzBall, Java Symbolic Pathfinder, ...

Although symbolic analysis is a very popular technique, scalability is a substantial challenge for symbolic execution. Dynamic state merging [12] provides one way to alleviate scalability challenges by opportunistically merging dynamic symbolic executors, which can be performed on paths Add std. cite or on environments FM paper from 2014 on Javascript?. Other techniques include CEGAR/-subsumption Add references from ASE 2017 paper: More Effective Interpolations in Software Model Checking.

Veritesting [2] is a different recently proposed technique that can dramatically improve the performance of symbolic execution. Rather than explicitly merge paths or check subsumption relationships, Veritesting simply encodes a local region of a program containing branches as a disjunctive region for symbolic analysis. If any path within the region meets an exit point, then the disjunctive formula is satisfiable. This often allows many paths to be collapsed into a single path involving the region. In previous work [2], bounded static code regions have been shown to find more bugs, and achieve more node and path coverage, when implemented at the X86 binary level for compiled C programs. This provides motivation for investigating integration of introducing static regions with symbolic execution at the Java bytecode level.

```
1  // x = ArrayList of symbolic integers with
2  // concrete length
3  for (int i = 0; i < x.size(); i++) {
4    // Begin region for static unrolling
5    if (x.get(i) < 0) sum += -1;
6    else if (x.get(i) > 0) sum += 1;
7    // End region for static unrolling
8  }
9  if (sum < 0) System.out.println("neg");
10 else if (sum > 0) System.out.println("pos");
11 else System.out.println("bug");
```

Listing 1.1: An example to loop through a symbolic array with three execution paths through the loop body

We present an example demonstrating the potential benefit of integrating static code regions with SPF in Listing 1.1. The example checks if positive or negative integers occur more frequently in the list x, and it contains a bug if x contains an equal number of positive and negative integers. The three-way branch on lines 5, 6 causes the total number of execution paths required to cover the *for* loop to be 3^{len}. However, this three-way branch can be combined into a multi-path region and represented as a disjunctive predicate. We present such predicates in SMT2 notation in Listing 1.2 assuming x to contain two symbolic integers named *x0* and *x1* (*len* equals 2). The updates to *sum* in the two loop iterations are captured by *sum0* and *sum1*. Using such predicates to represent the three-way branch on lines 5, 6 of Listing 1.1 allows us to have only one execution path through the loop body. Figure 1 shows a comparison of the number of execution paths explored to find the bug on line 11 of Listing 1.1. The exponential speed-up from our predicates, representing a multi-path region, allows us to find the bug using just three test cases.

Unfortunately, as originally proposed, Veritesting would be unable to create a static region for this loop because it involves non-local control jumps (the calls to the `get` methods). This is not an 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 resolved at runtime, so the compiler is unable to perform these optimizations.

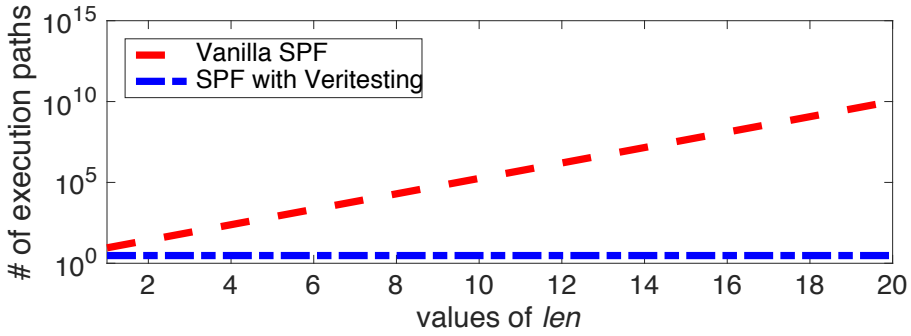
In Java, programs often consist of many small methods that are dynamically dispatched, leading to poor performance for näive implementations of bounded static regions. Thus, to be successful, we must be able to inject the static regions associated with the calls into the dispatching region. We call such regions *higher order* as they require a region as an argument and can return a region that may need to be further interpreted. Given support for such regions, we can make analysis of programs such as 1.1 trivial for large loop depths. In our experiments, we demonstrate 100x speedups on several models (in general, the more paths contained within a program, the larger the speedup) over the unmodified Java SPF tool using this approach.

```
1  ; one variable per array entry
2  (declare-fun x0 () (_ BitVec 32))
3  (declare-fun x1 () (_ BitVec 32))
4  ; a variable to represent 'sum'
5  (declare-fun sum () (_ BitVec 32))
6  ; one 'sum' variable per loop iteration
7  (declare-fun sum0 () (_ BitVec 32))
8  (declare-fun sum1 () (_ BitVec 32))
9  ; unrolled lines 5, 6 in Listing 1
10 (assert
11   (or (and (= x0 #x00000000) (= sum0 #x00000000))
12     (or (and (bvsgt x0 #x00000000) (= sum0 #x00000001))
13         (and (bvslt x0 #x00000000) (= sum0 #xffffffff)))))
14 ; second iteration of unrolling lines 5, 6
```

```
15 (assert
16   (or (and (= x1 #x10000000) (= sum1 #x10000000))
17     (or (and (bvsgt x1 #x10000000) (= sum1 #x10000001))
18         (and (bvslt x1 #x10000000) (= sum1 #xffffffff)))))
19 ; merge function for 'sum' variable
20 (assert (= sum (bvadd sum0 sum1)))
21 ; branch on line 9 of Listing 1
22 (assert (bvslt sum #x00000000))
```

Listing 1.2: SMT2 representation of multi-path execution in Listing 1.1 using $len = 2$

Fig. 1: Comparing number of execution paths from Listing 1.1 using vanilla SPF and SPF with static unrolling



1.1 Motivating Example

Consider the example of Java code shown in Figure 2. The `list` object refers to an `ArrayList` of 200 `Integer` objects which have an unconstrained symbolic integer as a field. The checking of each even-indexed entry in `list` introduces a branch, which has both sides feasible, and requires symbolic execution to explore two execution paths instead of the one it was at. 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 `i%2 == 0 && list.get(i) == 42` branch. Such path merging requires us to compute a summary of all behaviors arising on both sides of the branch from lines 11 to 13 until both sides of the branch merge at line 14. 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

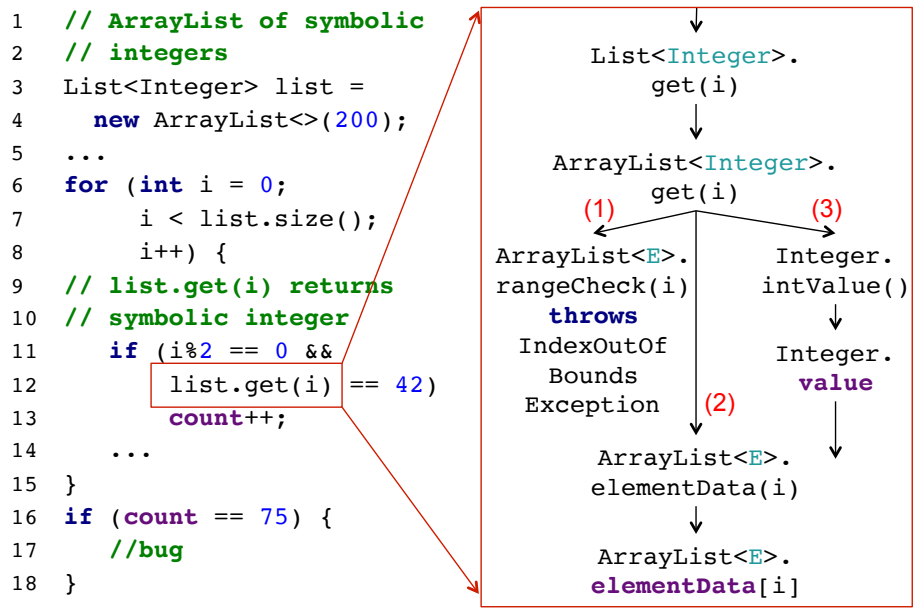


Fig. 2: An example demonstrating the need for using a multi-path region summary

summary to the stack and/or the heap. Unfortunately, constructing such a summary for this simple region from lines 11–13 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 [4]. 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

[4] Support of methods with multiple returns is a future work.

as having the type `List`. Therefore, the summary of `list.get(i)` pulls in the method summaryof `ArrayList<E>.get(i)`. Our 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.

2 Related Work

assigned to Stephen

The original idea for veritesting was presented by Avgerinos et al. [2]. They implemented veritesting on top of MAYHEM [7], a system for finding bugs at the X86 binary level which uses symbolic execution. Their implementation demonstrated dramatic performance improvements and allowed them to find more bugs, and have better node and path coverage. Veritesting has also been integrated with another binary level symbolic execution engine named `angr` [20]. Veritesting was added to `angr` with similar goals of statically and selectively merging paths to mitigate path explosion. However, path merging from veritesting integration with `angr` caused complex expressions to be introduced which overloaded the constraint solver. Using the Green [23] solver may alleviate such problems when implementing veritesting with SPF. Another technique named *MultiSE* for merging symbolic execution states incrementally was proposed by Sen et al. [18]. MultiSE computes a set of guarded symbolic expressions for every assignment and does not require identification of points where previously forked dynamic symbolic executors need to be merged. MultiSE complements predicate construction for multi-path regions beyond standard exit points (such as *invokevirtual*, *invokeinterface*, *return* statements). Combining both techniques, while a substantial implementation effort, has the potential to amplify the benefits from both techniques.

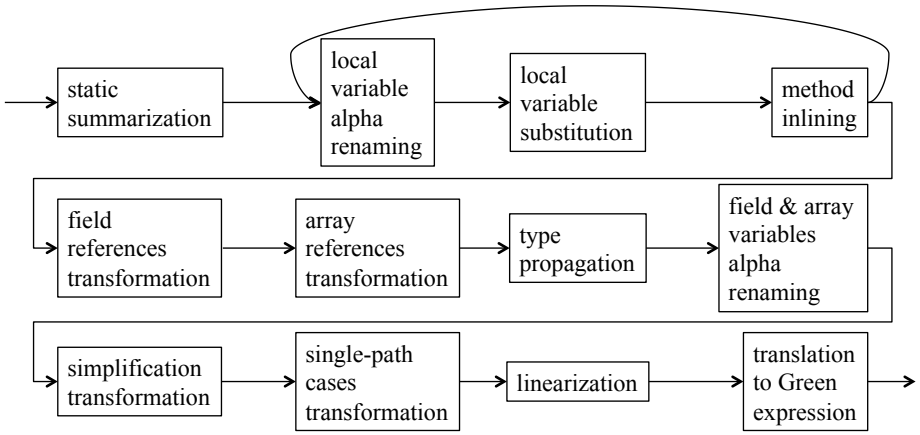
Finding which reflective method call is being used, or handling dynamic class loading are known problems for static analysis tools. TamiFlex [4] provides an answer that is sound with respect to a set of previously seen program runs. Integrating veritesting runs into similar problems, and using techniques from TamiFlex would allow static predicate construction beyond exit points caused by reflection or dynamic class loading.

3 Technique

To add path merging to SPF, we first pre-compute static summaries of arbitrary code regions with more than one execution path and methods. 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 atleast 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 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. 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 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. The instantiation of the region summary is performed as a sequence of transformations described below and summarized in Figure 3.

Fig. 3: Overview of transformations on Ranger IR to create and instantiate multi-path region summaries with higher-order regions



3.1 Statement Recovery

The 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 [**?**] 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.

3.2 Region Definition

Once the statement of a Ranger region has been recovered, its corresponding environment is populated. This includes identifying region boundary and creating input, output, type and stack slot tables for the region.
The region boundary is used to identify boundaries of the region as opposed to exploring the whole IR. This is used later to constrain the computation and population of different environment tables. For example, 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, if a converging statement exits. 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 [] but later undergoes some changes during type propagation transformation **??**. @Vaibhav: type propagation is mentioned in the figure but has no reference later in the transformation or in its text.
The stack slot table on the other hand, does not use region boundary for its population. The reason for this is that, the static analysis framework we use, WALA, sometime does not provide information about the stack slots of intermediate variables. This is particularly problematic in our case because the def of a phi is both an intermediate variable, and so we do not know its stack slot, yet it is also an output of the region for which we want to populate its symbolic representation onto the stack slot. Therefore, our stack slot table uses stack slot inference by propagating the stack slots of vars used in a phi onto the def of the phi. This requires visiting all variables and phi statements of the IR to maximize the inference of the stack slot, this is repeatedly done until a fix point is reached.

3.3 Instantiation-time Transformations

¡¡¡¡¡¡¡ HEAD **Alpha Renaming Transformation**: In Alpha renaming transformation, all Ranger IR variables are renamed to ensure their uniqueness before further processing takes place. ======= **Alpha Renaming Transformation**: In Alpha renaming transformation, all Ranger IR variables are renamed to ensure their uniqueness before further processing takes place. ¿¿¿¿¿¿¿ 1006ac5a6a3622f8ab31a1405909d8d8c5a72f6a 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.

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

Higher-order Regions Transformation: This transformation happens as a part of the substitution transformation. It is initiated when a method invocation is encountered.

At that point, three things happen. First, the region that corresponds to the called method, *method region*, is retrieved and basic alpha renaming is applied on it. Second, Ranger IR expressions that correspond to the actual parameters are evaluated and used to substitute the formal parameters by repeatedly applying substitution transformation over the method region. Then finally, when no more high-order regions could be discovered, the resulting substituted method region is inlined into the outer region.

If the method region has a single return value, then the original method invocation is replaced with an assignment to the returned expression. Otherwise, inlining of the method region takes place. The case where the method region has multiple returns, is currently not supported as candidate for high order region. It requires another transformation, we talk more about this in future work in section 7.

Field References SSA form: The field references transformation translates reads and writes of fields in Java bytecode into corresponding Ranger IR statements. In order to translate all field accesses to SSA form, this transformation creates a summary of the semantics represented by the field accesses in the region. This transformation constructs a new field access variable for every field assignment on every path within the region. This new field access variable construction makes use of two monotonically increasing subscripts. It uses a path subscript to distinguish between assignments to the same field on the same execution path. It uses a global subscript to distinguish between assignments to the same field across execution paths. At the merge point of the region, field assignments done on the same field are merged using Gated Single Assignment (GSA) [14]. 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.

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 [20]. 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 the Ranger 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 requires more involvement with the dynamic symbolic execution engine, i.e, object creation and throw instructions. The intuition here is that, we want to maximize regions that 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 statement. 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 Ranger had not summarized.

Linearization: Soha, Mike, please feel free to rewrite. 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 its if-

then-else statements anymore. In other words, the γ functions introduced by GSA are a functional representation of branching, which is what lets us capture the semantics of everything happening on both sides of the branch. Since the linearization transformation 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, 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 3 should correctly represent the semantics of the summarized region. If it doesn't, 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. [16]. 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 Experiment

assigned to Vaibhav

We would like to measure the performance of Java Ranger against the baseline of single-path exploration using Java Symbolic PathFinder. This can be examined in several dimensions: the wall-clock time of the solving process, the number of paths explored. In addition, we would also like to gather metrics about

the regions themselves, in order to better understand where static regions are effective.

Therefore, we investigate the following research questions:

RQ1: How much do higher-order static regions (HOSR) improve the performance of symbolic execution?
RQ2: How much do HOSRs reduce the number of paths explored?
RQ3: How do HOSRs affect the number and expense of calls to the SMT solver?
RQ4: How does the size of computed HOSRs affect the performance of the approach?

For each question, we examine three different configurations of Java Ranger: a version that creates simple regions (one branch) only (Java Ranger-SR), one that creates complex regions with multiple branches but no non-local jumps (Java Ranger-CR), and one that operates over higher-order complex regions containing non-local jumps (Java Ranger-CR+HO). This allows us to examine (at a coarse level) how each feature impacts the experimental results.

4.1 Experimental Setup

Information here about benchmark models and machine configuration

The benchmarks should be a superset of at least one previous paper, and better yet, multiple papers.

5 Results

5.1 Experimental Setup

We implemented all of the above mentioned transformations as a

We present our results in Table 1. Our static analysis results are shown in Table 2.

In this section, we examine experimental results from the perspective of each research question.

5.2 Performance

We consider the performance of Java Ranger against Java Symbolic Pathfinder in Table 3.

5.3 Number of Paths

We consider the number of paths explored by Java Ranger against Java Symbolic Pathfinder in Table 4.

Bench mark Name	Java Ranger mode	# exec paths	dyn. time (msec)	# solver queries	total solver time (msec)	# inst. regions	# higher order regions	# inst.
WBS-1step	mode 1	24	273	46	53	0	0	0
WBS-1step	mode 2	1	548	0	0	6	0	6
WBS-1step	mode 3	1	582	0	0	6	0	6
WBS-1step	mode 4	1	663	6	65	6	0	6
TCAS-SR-1step	mode 1	392	4115	2798	3792	0	0	0
TCAS-SR-1step	mode 2	18	999	74	428	19	0	107
TCAS-SR-1step	mode 3	1	512	0	0	4	28	4
TCAS-SR-1step	mode 4	1	619	4	72	4	28	4
replace_5	mode 1	1715	3470	5482	2763	0	0	0
replace_5	mode 2	1080	3.04E+04	1.30E+04	2.47E+04	17	0	977
replace_5	mode 3	632	36614	10259	2.88E+04	24	113	806
replace_5	mode 4	345	1.06E+05	7020	1.02E+05	26	122	501
TCAS-1step	mode 1	392	4698	2798	4287	0	0	0
TCAS-2steps	mode 1	1.54E+05	2.16E+06	1.10E+06	2.10E+06	0	0	0
TCAS-1step	mode 4	178	6848	1719	5349	13	0	445
TCAS-2steps	mode 4	3.17E+04	1.36E+06	3.08E+05	1.15E+06	13	0	7.97E+04
TCAS-SR-2steps	mode 1	1.54E+05	1.74E+06	1.10E+06	1.68E+06	0	0	0
TCAS-SR-2steps	mode 4	1	643	8	106	4	56	8
TCAS-SR-3steps	mode 4	1	704	12	116	4	84	12
TCAS-SR-4steps	mode 4	1	739	16	150	4	112	16
WBS-2steps	mode 1	576	775	1150	418	0	0	0
WBS-3steps	mode 1	13824	9806	27646	8364	0	0	0
WBS-4steps	mode 1	3.32E+05	2.18E+05	6.64E+05	1.86E+05	0	0	0
WBS-5steps	mode 1	7.96E+06	5.29E+06	1.59E+07	4.44E+06	0	0	0
WBS-2steps	mode 4	1	971	20	401	15	0	20
WBS-3steps	mode 4	1	1344	35	769	16	0	35
WBS-4steps	mode 4	1	1919	50	1235	16	0	50
WBS-5steps	mode 4	1	2165	65	1523	16	0	65
WBS-6steps	mode 4	1	3239	80	2479	16	0	80
WBS-7steps	mode 4	1	3326	95	2392	16	0	95

Table 1: The results from running Java Ranger on the WBS, TCAS, and Replace benchmarks. We created variants of WBS and TCAS by running them for multiple steps and present results from running Java Ranger on these variants too. mode 1 refers to Java Ranger running without any path-merging. mode 2 refers to Java Ranger running with path-merging for multi-path regions. mode 3 adds support for higher-order regions to mode 2. mode 4 adds support for single-path cases to mode 3.

Benchmark name	Static analysis time (msec)	# regions summarized	# methods summarized	max. branch depth
WBS	13118	3507	2203	5
TCAS	13830	3505	2209	4
Replace	7020	119	11	10

Table 2: Results of applying our static analysis on a few benchmarks

Table 3: Performance of Java Ranger vs. Java Symbolic Pathfinder

Program	SPF	JR CR+HO	JR SR	JR CR
Program1	0.005	2.335	0.192	0.355
Program2	0.014	13.297	0.589	1.473

Table 4: Number of Paths for Java Rangervs. Java Symbolic Pathfinder

Program	SPF	JR CR+HO	JR SR	JR CR
Program1	XXX	YYY	ZZZ	AAA
Program2	XXX	YYY	ZZZ	AAA

Table 5: Number and Aggregate Time of SMT Solver Calls for Java Ranger vs. Java Symbolic Pathfinder

Program	SPF	JR CR+HO	JR SR	JR CR
Program1	XXX (YY)	YYY (YY)	ZZZ (YY)	AAA (YY)
Program2	XXX (YY)	YYY (YY)	ZZZ (YY)	AAA (YY)

5.4 Number of SMT Calls

We consider the number of SMT calls explored by Java Ranger against Java Symbolic Pathfinder in Table 5. Note that this does not directly correspond to the number of paths, because static regions tend to make more of the variables symbolic, leading to larger numbers of solver calls per path.

5.5 Region Size

We consider the region size produced by each configuration in Table 6

Table 6: Size of Regions for Java Ranger vs. Java Symbolic Pathfinder

Program	SPF	JR CR+HO	JR SR	JR CR
Program1	XXX	YYY	ZZZ	AAA
Program2	XXX	YYY	ZZZ	AAA

6 Discussion

assigned to anyone who has something to say about future work we want do to on Java Ranger

- Small regions cause performance problems, especially when they make previously concrete information symbolic. This can lead to many more solver calls, even when the number of paths is reduced.
- In some models, it is possible to reduce the number of paths to one. The static region approach essentially constructs a unrolled version of the program, similar to what tools like CBMC construct. This can only happen on relatively static models that do not have a lot of object construction leading to multiple dispatch paths. HOSRs are more flexible for these situations and allow specialization depending on dispatch type, which we believe will lead to better performance for highly-dynamic models.
- In general, the solver time does not rise dramatically for disjunctive paths. Since (in the limit) we reduce the number of paths exponentially by removing branches, we can perform relatively expensive analyses as preprocessing steps and at instantiation if we are able to instantiate a static region, and still end up with much better performance.
- Talk about the need to have incremental solving or other optimizations to region summaries that can reduce formula complexity and make fewer solver calls
- Talk about predicting the number of execution paths in regions so that our constructed region summaries can be used by other symbolic executors with a knowledge of how much path reduction is being performed by the summary for a given number of symbolic inputs

7 Conclusion and Future Work

In this paper we presented Java Ranger as a path merging tool for Java. It works systematically by applying a series of transformations over a recovered statement from the CFG. This representation provides the benefit of modularity and visibility that promotes extension.
Java Ranger has its own IR statement and its supports the construction of SSA for fields and arrays. in addition Java Ranger supports inlining summaries of high order regions and it support the exploration of exceptional behavior via Single Path Cases. This maximizes portions of regions that could be summarized by Java Ranger.
There are five main directions for future work we intend to do:

- Support Multiple Return Regions: Here we want to support regions with multiple returns. We want to do this by supporting another transformation that would take a multiple return region and turns it into a single return region.
- Support of Test Case Generation: while statically summarizing regions gives dynamic symbolic execution a performance boost to explore more paths efficiently, generating test cases that covers all summarized branches is one of the fundamental roles of dynamic symbolic execution that is currently unsupported. This is one of the extension that we intend to investigate in our future work.
- Heuristics for Instantiation: as shown in the experiments section, instantiating regions might not always be the best option in terms of performance. For instance, region instantiation could in fact turn concrete values into symbolic values thus increasing the number of explored paths. This is particularly problematic if that happened for a small region on an early branch point, and the symbolic variable is later used in multiple conditions. This calls for a study of the best heuristics that would provide a decision procedure as to when instantiation a region.
- Using Incremental Solver: this seems intuitive for two main reasons: firstly because as we go down one path the query sent to the solver is incrementally defined for every branch and yet it is repeatedly being sent to the solver. And secondly because using Java Ranger tend to have relatively complex formulas to solve due to summarization. Supporting incremental solver is one direction for our future work.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: principles, techniques, and tools, vol. 2. Addison-wesley Reading (2007)
2. Avgerinos, T., Rebert, A., Cha, S.K., Brumley, D.: Enhancing symbolic execution with veritesting. In: Proceedings of the 36th International Conference on Software Engineering. pp. 1083–1094. ICSE 2014, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2568225.2568293, http://doi.acm.org/10.1145/2568225.2568293

3. Babic, D., Martignoni, L., McCamant, S., Song, D.: Statically-directed dynamic automated test generation. In: Proceedings of the 20th International Symposium on Software Testing and Analysis (ISSTA). pp. 12–22 (2011), http://doi.acm.org/10.1145/2001420.2001423
4. Bodden, E., Sewe, A., Sinschek, J., Oueslati, H., Mezini, M.: Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In: Proceedings of the 33rd International Conference on Software Engineering. pp. 241–250. ICSE '11, ACM, New York, NY, USA (2011)
5. Burnim, J., Sen, K.: Heuristics for scalable dynamic test generation. In: 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 443–446 (2008), https://doi.org/10.1109/ASE.2008.69
6. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)s. pp. 209–224 (2008), http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf
7. Cha, S.K., Avgerinos, T., Rebert, A., Brumley, D.: Unleashing mayhem on binary code. In: 2012 IEEE Symposium on Security and Privacy. pp. 380–394 (May 2012)
8. Chipounov, V., Kuznetsov, V., Candea, G.: The S2E platform: Design, implementation, and applications. ACM Trans. Comput. Syst. **30**(1), 2:1–2:49 (2012), http://doi.acm.org/10.1145/2110356.2110358
9. Clarke, L.A.: A system to generate test data and symbolically execute programs. IEEE Trans. Software Eng. **2**(3), 215–222 (1976), https://doi.org/10.1109/TSE.1976.233817
10. Godefroid, P., Klarlund, N., Sen, K.: Dart: Directed automated random testing. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 213–223. PLDI '05, ACM, New York, NY, USA (2005). https://doi.org/10.1145/1065010.1065036, http://doi.acm.org/10.1145/1065010.1065036
11. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976), http://doi.acm.org/10.1145/360248.360252
12. Kuznetsov, V., Kinder, J., Bucur, S., Candea, G.: Efficient state merging in symbolic execution. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 193–204. PLDI '12, ACM, New York, NY, USA (2012)
13. Luckow, K., Dimjašević, M., Giannakopoulou, D., Howar, F., Isberner, M., Kahsai, T., Rakamarić, Z., Raman, V.: JDart: A dynamic symbolic analysis framework. In: Chechik, M., Raskin, J.F. (eds.) Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science, vol. 9636, pp. 442–459. Springer (2016)
14. Ottenstein, K.J., Ballance, R.A., MacCabe, A.B.: The program dependence web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. In: Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation. pp. 257–271. PLDI '90, ACM, New York, NY, USA (1990). https://doi.org/10.1145/93542.93578, http://doi.acm.org/10.1145/93542.93578
15. Păsăreanu, C.S., Visser, W., Bushnell, D., Geldenhuys, J., Mehlitz, P., Rungta, N.: Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. Automated Software Engineering **20**(3), 391–425 (Sep 2013). https://doi.org/10.1007/s10515-013-0122-2, https://doi.org/10.1007/s10515-013-0122-2

16. Ramos, D.A., Engler, D.R.: Practical, low-effort equivalence verification of real code. In: Proceedings of the 23rd International Conference on Computer Aided Verification. pp. 669–685. CAV'11, Springer-Verlag, Berlin, Heidelberg (2011), http://dl.acm.org/citation.cfm?id=2032305.2032360
17. Sen, K., Marinov, D., Agha, G.: Cute: A concolic unit testing engine for c. In: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 263–272. ESEC/FSE-13, ACM, New York, NY, USA (2005). https://doi.org/10.1145/1081706.1081750, http://doi.acm.org/10.1145/1081706.1081750
18. Sen, K., Necula, G., Gong, L., Choi, W.: Multise: Multi-path symbolic execution using value summaries. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. pp. 842–853. ESEC/FSE 2015, ACM, New York, NY, USA (2015). https://doi.org/10.1145/2786805.2786830, http://doi.acm.org/10.1145/2786805.2786830
19. Sharma, V., Hietala, K., McCamant, S.: Finding Substitutable Binary Code By Synthesizing Adaptors. In: 11th IEEE Conference on Software Testing, Validation and Verification (ICST) (Apr 2018)
20. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., Vigna, G.: Sok: (state of) the art of war: Offensive techniques in binary analysis. In: 2016 IEEE Symposium on Security and Privacy (SP). pp. 138–157 (May 2016). https://doi.org/10.1109/SP.2016.17
21. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: Augmenting fuzzing through selective symbolic execution. In: NDSS. vol. 16, pp. 1–16 (2016)
22. Sun, W., Xu, L., Elbaum, S.: Improving the cost-effectiveness of symbolic testing techniques for transport protocol implementations under packet dynamics. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 79–89. ISSTA 2017, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3092703.3092706, http://doi.acm.org/10.1145/3092703.3092706
23. Visser, W., Geldenhuys, J., Dwyer, M.B.: Green: Reducing, reusing and recycling constraints in program analysis. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. pp. 58:1–58:11. FSE '12, ACM, New York, NY, USA (2012). https://doi.org/10.1145/2393596.2393665, http://doi.acm.org/10.1145/2393596.2393665