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

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 [13, 9], symbolic execution is a convenient building block for program analysis, since arbitrary query predicates can be combined

⋆ These authors were equal contributors to this work

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, 19], equivalence checking [18, 21], vulnerability finding [24, 23], and protocol correctness checking [25]. Symbolic execution tools are available for many languages, including CREST [5] for C source code, KLEE [6] for C/C++ via LLVM, JDart [15] and Symbolic PathFinder [17] for Java, and S2E [8], FuzzBALL [3], and `angr` [23] for binary code. Some of these tools, such as `angr` and Mayhem [7] that operate at the binary-level, are used for finding security bugs. Others, such as KLEE, are used for exploring system-level programs for software engineering purposes.

Although symbolic analysis is a very popular technique, scalability is a substantial challenge for symbolic execution. Dynamic state merging [14] provides one way to alleviate scalability challenges by opportunistically merging dynamic symbolic executors. Other techniques such as subsumption checking [26] make use of interpolants to find guarantees about safety properties on an execution path.

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.

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 [22]. 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. 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 increases the reduction in branching a symbolic executor has to perform. 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 has Static Single Assignment (SSA) form provides a few key advantages. (1) It allows region summaries to be constructed by using a sequence of transformations, with each transformation extending to add support for new features such as heap accesses, higher-order regions, and single-path cases. (2) It allows for simplifications such as constant propagation, copy propagation, constant folding to be performed on region summaries. (3) It makes the construction of region summaries more accessible to users of the symbolic execution tool, thereby making path merging more useful to end-users. In this paper, we present Java Ranger, an extension of Symbolic PathFinder, that computes such region summaries over Ranger IR. Ranger IR has support for inlining method summaries and constructs SSA form for heap accesses. It also proposing the Single-Path case as an alternative to transition points as defined by Avgerinos et al. [2].

1.1 Motivating Example

Consider the example of Java code shown in Figure 1. 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 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

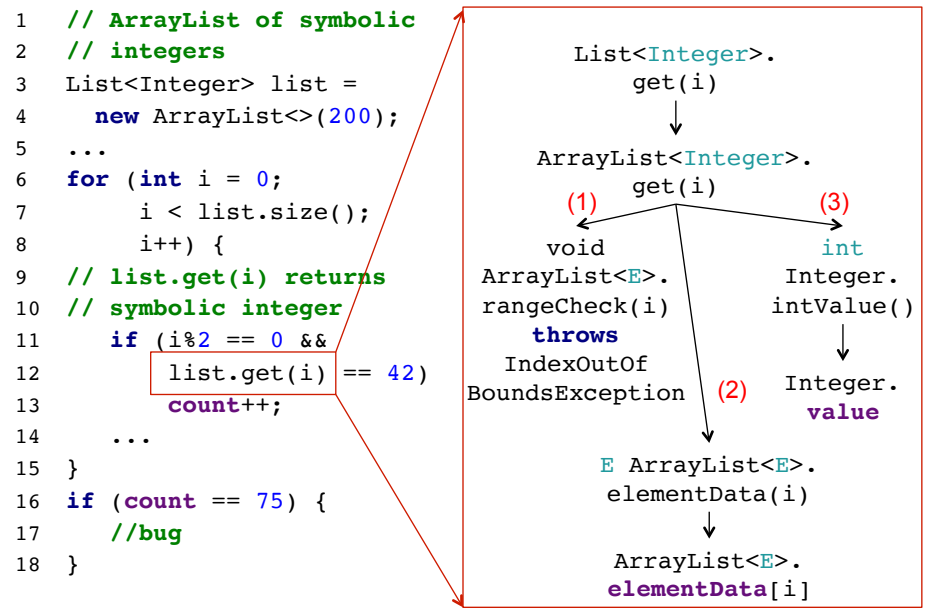


Fig. 1: An example demonstrating the need for using a multi-path region summary

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 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-

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

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

in progress

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. [11] explored this technique but found mixed results on its benefits. Kuznetsov et al. [14] developed new algorithms and heuristics to control when to perform such state merging profitably. A larger departure in the architecture of symbolic execution systems is the MultiSE approach proposed by Sen et al. [20], which represents values including the program counter with a two-level guarded structure, in which the guard expressions are optimized with BDDs. The MultiSE approach achieves effects similar to state merging automatically, and provides some architectural advantages such as in representing values that are not supported by the SMT solver.

Another approach to achieve path merging is to statically summarize regions that contain branching control flow. This approach was proposed by Avgerinos et al. [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 we chose it for use in 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 [7].

The veritesting approach has been integrated with another binary level symbolic execution engine named `angr` [23]. However angr's authors found that their veritesting implementation did not provide an overall improvement over their dynamic symbolic execution baseline: though veritesting allowed some new crashes to be found, they observed that giving more complex symbolic expressions slowed down the SMT solver enough that total performance was degraded.

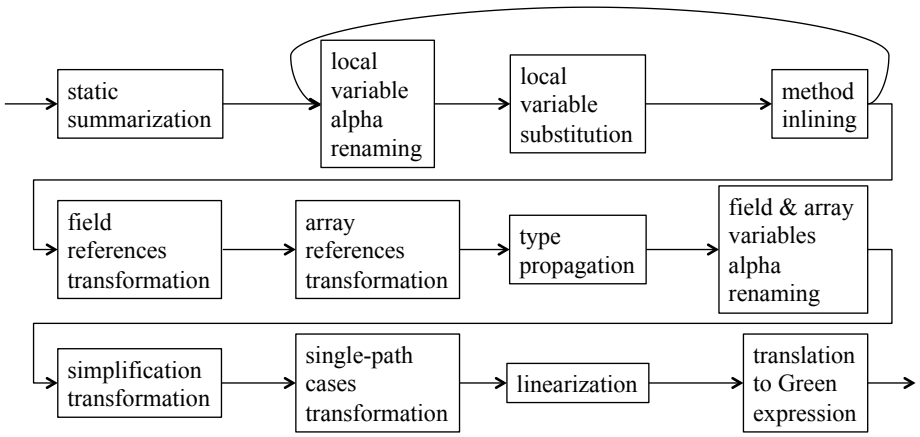
We have also observed complex expressions to be a potential cost of veritesting, but we believe that optimizations of the SMT solver interface and potentially heuristics to choose when to use veritesting can allow it to be a net asset.

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. After computing static region and method summaries, we instantiate them as a sequence of transformations described below and summarized in Figure 2.

Fig. 2: 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 [28] but with slightly different goals:

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

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

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

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 described below.
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

Alpha 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.
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 6.
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) [16]. 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 [23]. For example, if an entry in an array is never written to inside a region, the array reference transformation can still have an array output for that entry that writes a new symbolic variable into it. The region summary would then need to have an additional constraint that makes the new symbolic variable equal the original value in that array entry. Such conjuncts in the region summary can be easily eliminated with constant propagation, copy propagation, and constant folding [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 cre-

ation 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: 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 2 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. [18]. 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 [17] tool. To make use of region summaries in Symbolic PathFinder, we use an existing feature of SPF named *listener*. A listener is a method defined within SPF that is called for every bytecode instruction executed by SPF. Java Ranger adds a path merging listener to SPF that, on every instruction, checks (1) if the instruction involves checking a symbolic condition, and (2) if Java Ranger has a pre-computed static summary that begins at that instruction's bytecode offset. If both of these conditions are satisfied, Java Ranger instantiates the multi-path region summary corresponding to that bytecode offset by reading inputs from and writing outputs to the stack and the heap. It then conjuncts the instantiated region summary with the path condition and resumes symbolic execution at the bytecode offset of the end of the region. Our implementation, named Java Ranger, wraps around SPF and can be configured to run in four modes. (1) In mode 1, it runs vanilla SPF without any path merging enabled. (2) In mode 2, it summarizes multi-path regions only and instantiates them if they are encountered. (3) In mode 3, it summarizes and instantiates methods along with multi-path regions as done in mode 2. (4) In mode 4, it uses single-path cases along with multi-path region and method summaries used in mode 3.

4.2 Evaluation

In order to evaluate the performance of Java Ranger, we used the following benchmarking programs commonly used to evaluate symbolic execution performance. (1) Wheel Brake System (WBS) [29] is a synchronous reactive component developed to make aircraft brake safely when taxing, landing, and during a rejected take-off. (2) Traffic Collision Avoidance System (TCAS) is part of a suite of programs commonly referred to as the Siemens suite [12]. TCAS is a system that maintains altitude separation between aircraft to avoid mid-air collisions. (3) Replace is another program that's part of the Siemens suite. Replace searches for a pattern in a given input and replaces it with another input string. We used the translation of the Siemens suite to Java as made available by Wang et al. [27]. We also manually created a variant of TCAS by converting regions of code with return statements on every execution path to regions of code with a single return statement at the merge point of the region. We refer to this variant of TCAS as TCAS-SR (TCAS with Single Returns). We also created variants of WBS, TCAS, and TCAS-SR by running them for multiple steps.

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. The "# exec paths" column is the number of execution paths required to completely explore the benchmark. "dyn. time (msec)" is the time taken for dynamic analysis (excludes static analysis time). "inst. regions" is the total number of regions that were instantiated at least once when running each benchmark. "# higher-order regions" is the number of higher-order regions that were used and "inst." is the total number of instantiations that was done when running the benchmark.

We ran WBS, TCAS, TCAS-SR, and Replace using Java Ranger. We present our results from instantiating region summaries in Table 1. Table 1 shows that we achieve a significant improvement using path-merging with WBS. We outperform SPF when running TCAS too but our improvement doesn't allow path merging to summarize the entire program to a single execution path, as is the case with WBS. We analyzed the source code of TCAS and found the only barrier to our path-merging was the presence of several code regions containing a return instruction on every execution path. We manually performed a semantics-preserving transformation to the source code of TCAS to create another version, TCAS-SR, that has a single return value at the end of all such multi-path regions. We plan to automate this transformation in the future. Table 1 shows the benefit of such a *multiple-returns-to-single-return* transformation. This transformation allows Java Ranger to summarize the entire TCAS-SR program into a single execution path. The performance of Java Ranger when running Replace is quite different from that seen when running WBS, TCAS, and TCAS-SR. While Java Ranger reduces the total number of execution paths with every increase in mode, the path reduction isn't as significant. More importantly, it also makes an order of magnitude more solver calls thereby increasing the total runtime.

We further investigated this drop in performance of Java Ranger when running Replace by exploring the space of potential regions to instantiate. We obtained the list of regions that are instantiated at least once when running Replace with Java Ranger in mode 3 (using method and multi-path region summaries). This step produced a list of 24 regions. Next, we sampled the space of all possible subsets of this set of 24 regions by (1) randomly enumerating the space up to 4000 subsets of all possible subsets of 24 regions, (2) enumerating the space in the order of subset sizes up to all subsets of 3 regions from a set of 24 regions. With both methods of enumeration, we searched for the least amount of time Java Ranger needs to instantiate at least one region with Replace. We found this least time to be 26.3 seconds which is still more than the 3.47 seconds Java Ranger needs to run Replace in mode 1 (running vanilla SPF with no path merging).update these numbers before submission In the fastest runs with at least one region instantiation in both methods of enumeration, Java Ranger finished exploration with the same number of execution paths as that explored during mode 1. The result of this analysis, combined with a significant increase in the number of solver queries with Java Ranger's path-merging modes, points toward the conclusion that every instantiation of a multi-path region in Replace causes another branch to be symbolic. This conclusion is also in alignment with the observation made by Kuznetsov et al. [14] that path merging can sometimes have a net negative effect on the performance of symbolic execution. We plan to integrate heuristics for estimating the side-effects of introducing new symbolic state as a result of path merging on symbolic execution performance with Java Ranger in the future.

5 Future Work

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

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

6 Conclusion

We presented Java Ranger as a path merging tool for Java. It works by systematically applying a series of transformations over a statement recovered from the CFG. This representation provides the benefit of modularity and makes path-merging for Java symbolic execution more accessible to end users. Java Ranger

has its own IR statement and its 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. 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. Hansen, T., Schachte, P., Søndergaard, H.: State joining and splitting for the symbolic execution of binaries. In: Runtime Verification, 9th International Workshop, RV 2009, Grenoble, France, June 26-28, 2009. Selected Papers. pp. 76–92 (2009)
12. Hutchins, M., Foster, H., Goradia, T., Ostrand, T.: Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In: Proceedings of 16th International Conference on Software Engineering. pp. 191–200 (May 1994). https://doi.org/10.1109/ICSE.1994.296778
13. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976), http://doi.acm.org/10.1145/360248.360252
14. 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)
15. 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)
16. 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
17. 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
18. 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
19. 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
20. 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
21. 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)
22. Sharma, V., Whalen, M.W., McCamant, S., Visser, W.: Veritesting challenges in symbolic execution of java. SIGSOFT Softw. Eng. Notes **42**(4), 1–5 (Jan 2018). https://doi.org/10.1145/3149485.3149491, http://doi.acm.org/10.1145/3149485.3149491

23. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., Vigna, G.: Sok: (state of) the art of war: Offensive techniques in binary analysis. In: 2016 IEEE Symposium on Security and Privacy (SP). pp. 138–157 (May 2016). https://doi.org/10.1109/SP.2016.17
24. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: Augmenting fuzzing through selective symbolic execution. In: NDSS. vol. 16, pp. 1–16 (2016)
25. Sun, W., Xu, L., Elbaum, S.: Improving the cost-effectiveness of symbolic testing techniques for transport protocol implementations under packet dynamics. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 79–89. ISSTA 2017, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3092703.3092706, http://doi.acm.org/10.1145/3092703.3092706
26. Tian, C., Duan, Z., Duan, Z., Ong, C.H.L.: More effective interpolations in software model checking. In: Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering. pp. 183–193. ASE 2017, IEEE Press, Piscataway, NJ, USA (2017), http://dl.acm.org/citation.cfm?id=3155562.3155589
27. Wang, H., Liu, T., Guan, X., Shen, C., Zheng, Q., Yang, Z.: Dependence guided symbolic execution. IEEE Transactions on Software Engineering **43**(3), 252–271 (March 2017). https://doi.org/10.1109/TSE.2016.2584063
28. Yakdan, K., Eschweiler, S., Gerhards-Padilla, E., Smith, M.: No more gotos: Decompilation using pattern-independent control-flow structuring and semantics-preserving transformations (02 2015). https://doi.org/10.14722/ndss.2015.23185
29. Yang, G., Person, S., Rungta, N., Khurshid, S.: Directed incremental symbolic execution. ACM Transactions on Software Engineering and Methodology (TOSEM) **24**(1), 3 (2014)