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 [10, 8], 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 [9, 15], equivalence checking [14, 17], vulnerability finding [19, 18], and protocol correctness checking [20]. Symbolic execution tools are available for many languages, including CREST [4] for C source code, KLEE [5] for C/C++ via LLVM, JDart [12] and Symbolic PathFinder [13] for Java, and S2E [7], FuzzBALL [2], and angr [18] 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 [11] 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 [1] 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 [1], 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.

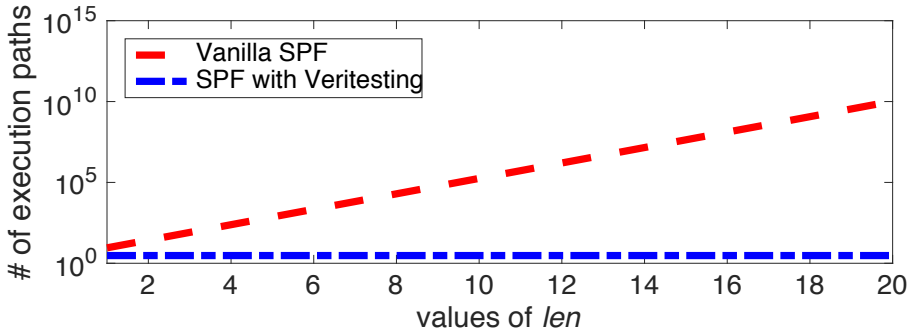
In Java, programs often consist of many small methods that are dynamically dispatched, leading to poor performance for näive implementations of bounded static regions. Thus, to be successful, we must be able to inject the static regions associated with the calls into the dispatching region. We call such 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

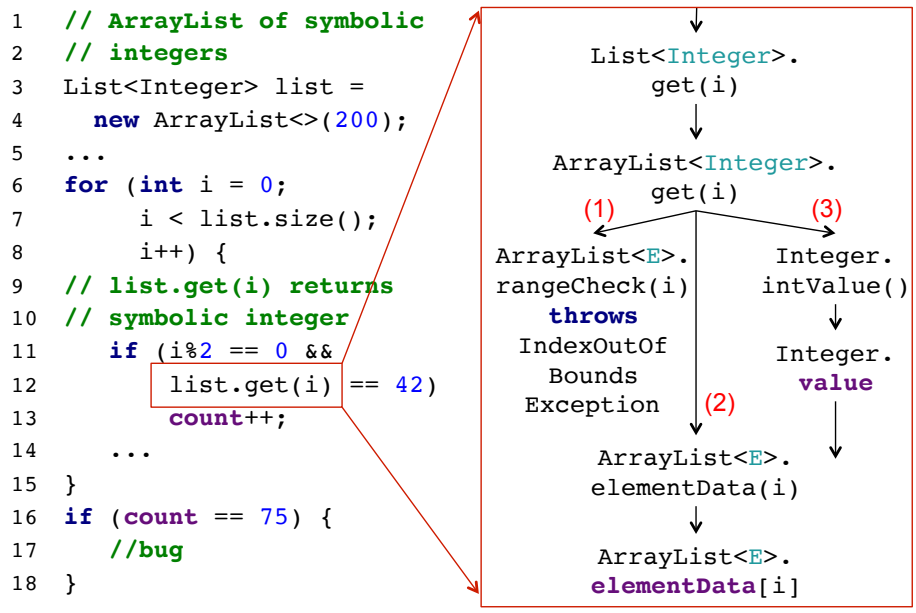


Fig. 2: An example demonstrating the need for using a multi-path region summary

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 as part of region summaries that have method calls. 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`. 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. [1]. They implemented veritesting on top of MAYHEM [6], 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` [18]. 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 [21] 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. [16]. 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.

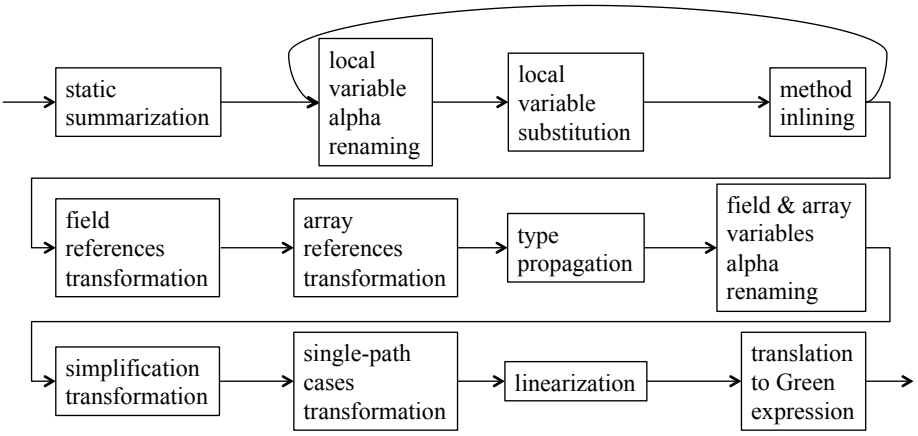
Finding which reflective method call is being used, or handling dynamic class loading are known problems for static analysis tools. TamiFlex [3] provides an answer that is sound with respect to a set of previously seen program runs. Integrating veritesting runs into similar problems, and using techniques 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 Representation of Static Regions

assigned to Mike MWW: - we should provide an AST of the constraint language

3.2 Instantiation-time Transformations

Alpha Renaming: assigned to Soha
Local Variable Substitution: assigned to Soha
Higher-order Regions: assigned to Soha
Field References SSA form: assigned to Vaibhav
Array References SSA form: assigned to Vaibhav
Simplification of Ranger IR: assigned to Vaibhav

Single Path Cases: assigned to Soha
Linearization: assigned to Mike or Soha (whoever grabs it first)
Translation to Green: assigned to Soha

3.3 Checking Correctness

assigned to Vaibhav

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

assigned to Vaibhav

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

5.1 Performance

We consider the performance of Java Ranger against Java Symbolic Pathfinder in Table 1.

Table 1: 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

5.2 Number of Paths

We consider the number of paths explored by Java Ranger against Java Symbolic Pathfinder in Table 2.

Table 2: 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

5.3 Number of SMT Calls

We consider the number of SMT calls explored by Java Ranger against Java Symbolic Pathfinder in Table 3. 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.

Table 3: 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)

Table 4: 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

5.4 Region Size

We consider the region size produced by each configuration in Table 4

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

assigned to the one who gets to it first

Future work:

Extensions:

1. Simplification of static regions at instantiation time using constant propagation, code specialization, etc.
2. Adding support for parallel Java programs with static regions
3. Adaptation of metric-based test-case generation for static regions: statement, branch, MCDC, observable metrics.
4. Parallelization of the analysis process (similar to Staats work in 2011)
5. Adding support for bypass of complex expressions: *symcrete* execution involving regions.
6. Integration with the Green constraint solver

Other improvements:

1. Simplification of the Symbolic PathFinder constraint mechanism
2. Interpolation-based path subsumption checks (c.f.: "More Effective Interpolations in Software Model Checking" - ASE 2017")

References

1. 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
2. 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
3. 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)
4. 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
5. 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
6. 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)
7. 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
8. 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
9. 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

10. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976), http://doi.acm.org/10.1145/360248.360252
11. 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)
12. 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)
13. 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
14. 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
15. 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
16. 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
17. 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)
18. 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
19. 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)
20. 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

21. 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