Veritesting Challenges in Symbolic Execution of Java

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

[1]University of Minnesota, Minneapolis, MN, United States of America

[2]University of Stellenbosch, Stellenbosch, South Africa

[1]{*vaibhav,mwwhalen*}*@umn.edu*, *mccamant@cs.umn.edu*

[2]*wvisser@cs.sun.ac.za*

ABSTRACT

Scaling symbolic execution to industrial-sized programs is an important open research problem. Veritesting is a promising technique that improves scalability by combining the advantages of static symbolic execution with those of dynamic symbolic execution. The goal of veritesting is to reduce the number of paths to explore in symbolic execution by creating formulas describing regions of code using disjunctive formulas. In previous work, veritesting was applied to binary-level symbolic execution.

Integrating veritesting with Java bytecode presents unique challenges: notably, incorporating non-local control jumps caused by runtime polymorphism, exceptions, native calls, and dynamic class loading. If these language features are not accounted for, we hypothesize that the static code regions described by veritesting are often small and may not lead to substantial reduction in paths. We examine this hypothesis by running a Soot-based static analysis on six large open-source projects used in the Defects4J collection. We find that while veritesting can be applied in thousands of regions, allowing static symbolic execution involving non-local control jumps amplifies the performance improvement obtained from veritesting. We hope to use these insights to support efficient veritesting in Symbolic PathFinder in the near future. Toward this end, we briefly address some engineering challenges to add veritesting into SPF.

Keywords

multi-path symbolic execution; veritesting; Symbolic PathFinder; static analysis

1. INTRODUCTION

Symbolic execution is a popular testing technique that performs non-standard execution of a program. Having been originally proposed in the 1970s, it has many applications like test generation [Godefroid et al. 2005, Sen et al. 2005], equivalence checking [Ramos and Engler 2011, Sharma et al. 2017], finding vulnerabilities [Stephens et al. 2016, Shoshitaishvili et al. 2016].However, scalability continues to remain a challenge for symbolic execution. Dynamic state merging [Kuznetsov et al. 2012] provides one way to alleviate scalability challenges by opportunistically merging dynamic symbolic executors. Veritesting [Avgerinos et al. 2014] is a different recently proposed technique that is more suitable to symbolic execution tools that execute one path at a time. It has been shown to find more bugs, and achieve more node and path coverage, when implemented at the X86 binary level. This provides motivation for investigating integration of veritesting with symbolic execution at the Java bytecode level.

Symbolic Pathfinder (SPF) [Păsăreanu et al. 2013] is a tool that performs symbolic execution of Java bytecode. We present an example demonstrating the potential benefit of integrating veritesting with SPF in Listing 1. It checks if positive or negative integers occur more frequently in the array x. But, Listing 1 contains a hypothetical bug if x contains an equal number of positive and negative integers. The three-way branch on lines 6, 7 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 predicate by using disjunctions. Each iteration through the loop can be represented by such a predicate.

```
1  // x = array of symbolic integers
2  // len = concrete length of x
3  sum = 0;
4  for (int i = 0; i < len; i++) {
5    // Begin region for static unrolling
6    if (x[i] < 0) sum += -1;
7    else if (x[i] > 0) sum += 1;
8    // End region for static unrolling
9  }
10 if (sum < 0) { // ... handles negative sum
11 } else if (sum > 0) { // ... handles positive sum
12 } else { // ... contains a bug
13 }
```

Listing 1: An example to loop through a symbolic array with three execution paths through the loop body

```
; one variable per array entry
(declare-fun x0 () (_ BitVec 32))
(declare-fun x1 () (_ BitVec 32))
; a variable to represent 'sum'
(declare-fun sum () (_ BitVec 32))
; one 'sum' variable per loop iteration
(declare-fun sum0 () (_ BitVec 32))
(declare-fun sum1 () (_ BitVec 32))
; unrolled lines 5, 6 in Listing 1
(assert
  (or (and (= x0 #x00000000) (= sum0 #x00000000))
    (or (and (bvsgt x0 #x00000000) (= sum0 #x00000001))
        (and (bvslt x0 #x00000000) (= sum0 #xffffffff)))))
; second iteration of unrolling lines 5, 6
(assert
  (or (and (= x1 #x00000000) (= sum1 #x00000000))
    (or (and (bvsgt x1 #x00000000) (= sum1 #x00000001))
        (and (bvslt x1 #x00000000) (= sum1 #xffffffff)))))
; merge function for 'sum' variable
(assert (= sum (bvadd sum0 sum1)))
; branch on line 9 of Listing 1
(assert (bvslt sum #x00000000))
```

Listing 2: SMT2 representation of multi-path execution in Listing 1 using $len = 2$

We present such predicates in SMT2 notation in Listing 2 assuming the array x to contain two symbolic integers named $x0$ and

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

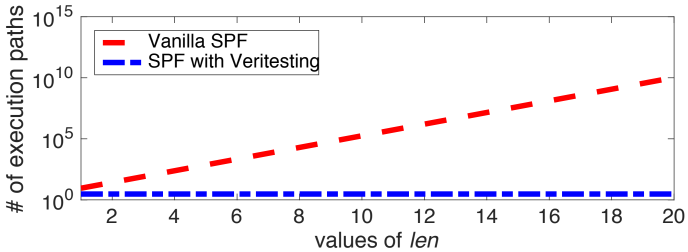
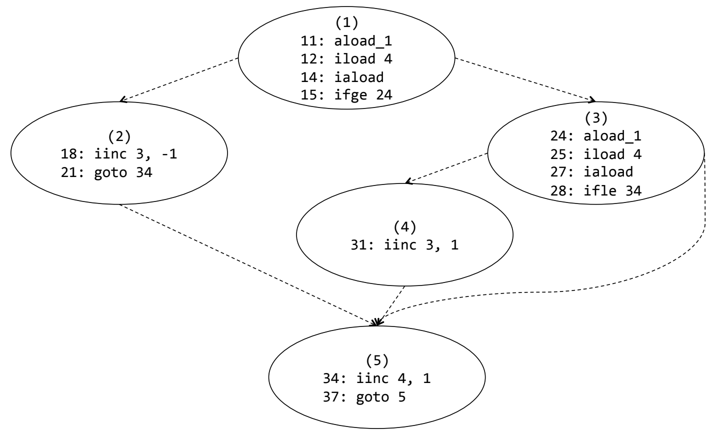


Figure 2: Control-flow graph with `else return;` added as line 7 of Listing 1



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 6, 7 of Listing 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. The exponential speed-up from our predicates, representing a multi-path region, allows us to find the bug using just three test cases. Veritesting provides such performance improvements by using static analysis to identify multi-path regions and create predicates to represent them.

2. CHALLENGES

While the performance improvement demonstrated on the code in Listing 1 is impressive, it is perhaps not representative of most Java code. Java conventions encourage the use of indirection when accessing class fields using non-static *get* and *set* methods, as well as liberal use of exceptions. Unlike C compilers, which assume a "closed world" and often inline simple functions into the body of calling methods to improve performance, the Java compiler must assume an "open world" in which a class may be used in a new context, so inlining of non-final methods is unsafe.

Previous approaches to veritesting exit static code regions when indirect calls to functions or non-local jumps are made. In this section, we explore how the structure of Java programs might reduce the performance of a naïve veritesting approach.

2.1 Exit Points

Integrating veritesting with SPF requires that we can represent a region of a Java program with multiple exit points as a disjunctive formula. Each exit point describes a possible distinct continuation of the current path after the static code region completes execution. Avgerinos et al. [Avgerinos et al. 2014] defined exit points as unresolved jumps, function boundaries, and system calls. These exit points are nodes in a control-flow graph which represent non-local control flow, and therefore, need to be explored using plain dynamic symbolic execution. In the context of Java bytecode, we find such non-local control flow in five ways listed as follows: (1) return statements, (2) exceptions, (3) virtual function invocations (*invokevirtual*, *invokeinterface*), (4) reflection, and (5) native calls.

The primary benefit of implementing veritesting comes from its conversion of branches into disjunctions in a multi-path region. But this benefit exists only when the number of different exit points from the disjunctive formula is less than the number of execution paths through the region in the first place. For example, all execution paths in the first three-way branch in Listing 1 joined together on line 7, causing the three-way branch to have a single exit point. Therefore, it is crucial for us to study the number of exit points for each of our statically-analyzed regions vs. the number of branches within the region.

These five types of exit points create the kind of non-local control flow which formed the frontier of the visible control-flow graph created as a result of *CFGReduce* step by Avgerinos et al. However, many of these exit points are used pervasively by Java developers. Running into exit points too often causes the performance gain from having fewer branches to be lost.

We investigated the occurrence of these exit points by creating a Soot-based [Vallée-Rai et al. 1999] static analysis of six large open-source projects written in Java. Software faults from these six projects are maintained in the Defects4J [Just et al. 2014] repository. We used Soot to create a control-flow graph for every method body in every class file in these six projects. For each control-flow graph, we used nodes corresponding to conditional jump bytecodes as a starting point of our analysis. We measured the number of instructions encountered when traversing down each side of the branch until we get to the immediate post-dominator [Aho et al. 2007] of our starting point. A node d strictly postdominates node n if every path from n to the exit node of the graph goes through d and d is not the same as n. d is an immediate post-dominator of n if it strictly postdominates n but does not postdominate any other strict postdominators of n. Starting from a branch, if there were no exit points encountered on any side of the branch until we got to its immediate post-dominator, we considered this region as a pure multi-path region and calculated its size in bytecode instructions. On observing 10 randomly chosen files from each project of the Defects4J repository, we did not find any instance of more than five nested branches. We allowed up to five nested branches and calculated the number of bytecode instructions from the earliest and the latest starting point in our control-flow graph traversal to an exit point.

As an example, Figure 2 shows the control-flow graph for the three-way branch in Listing 1. Nodes are numbered 1 through 5 with node (1) being the starting point and node (5) being the immediate post-dominator of node (1). Nodes contain bytecode instructions along with instruction offsets. Counting the number of instructions after the `ifge` instruction in node (1) through node (2) to node (5) gives us two instructions. The same count going through nodes (3) and (4) to (5) gives us 5 instructions, and through nodes (3) to (5) gives us 4 instructions. With this three-way branch having only one exit point, this region can be considered a pure multi-path region.

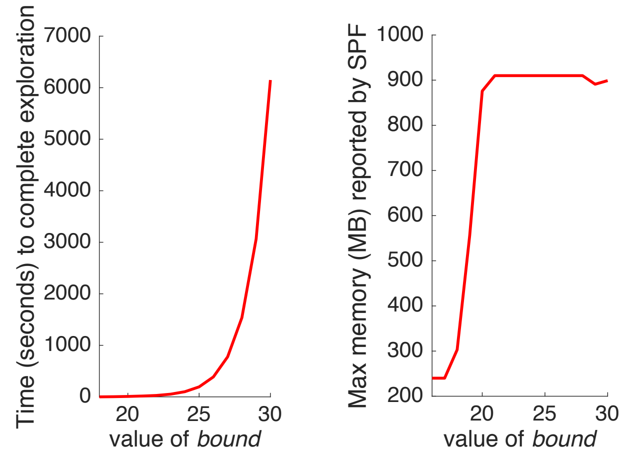
We performed this analysis on the six projects in the Defects4J repository. The *if-return*, *if-invokevirtual*, *if-throw* columns in Table 1 report the average number of instructions observed be-

	#classes	if-return	if-invokevirtual	if-throw	region size
chart	679	8.44	27.47	4.33	13.59
closure	1339	7.35	22.1	9.5	11.66
lang	170	6.70	11.64	7.09	9.60
math	1104	18.27	56.61	9.56	27.06
mockito	382	6.02	12.51	8.05	13.57
time	209	7.79	13.10	7.08	8.10

Table 1: Soot-based analysis for number of bytecode instructions between starting and exit points

	if-return	if-invokevirtual	if-throw	region count
chart	1712	7760	521	6627
closure	3853	7466	138	9258
lang	3602	1589	539	2065
math	2219	5582	662	15375
mockito	372	572	15	574
time	1202	984	204	1421

Table 2: Number of occurences in Soot-based static analysis



(a) Time required for covering paths in Listing 3

(b) Maximum memory usage of SPF for covering paths in Listing 3

Figure 3: Time and memory usage of Listing 3 when increasing *bound*

tween any *if* bytecode instruction and a *return*, *invokevirtual* or *invokeinterface*, or *throw* bytecode instruction. These same columns in Table 2 report the total number of times we observed an occurence of one of *return*, *invokevirtual*, *invokeinterface*, *throw* opcode-containing bytecode instructions before reaching the immediate post-dominator of the starting *if* node on any side of the branch. The *region size* and *region count* columns in Tables 1 and 2 report the average size (in terms of Java bytecode instructions) and total number of occurences of multi-path regions with a single exit point. These represent pure multi-path regions. While we discover thousands of regions which have a single exit point, these regions are small. They also show that early *return* instructions are another often used construct in Java. We believe that creating multi-path regions for these no-exit-point cases alone would provide a significant performance boost to SPF. Table 2 shows *invokevirtual* or *invokeinterface* instructions are encountered far more often than *return* or *throw* instructions. This can be intuitively explained by the extensive use of runtime polymorphism and exception handling by Java developers. Instead of using *invokevirtual* and *invokeinterface* instructions as exit points, continuing our predicate construction for multi-path regions beyond them would almost double the number of multi-path regions.

While dynamic dispatch and return statements are challenges for veritesting, there exist other challenges which need to be addressed to push the veritesting frontier further. Heap allocations, which occur within a veritesting region, can be treated as exit points but including them in our predicate construction requires careful consideration of the objects being allocated and correct interaction with the heap maintained by the symbolic execution engine. Method calls and field accesses on symbolic objects also require special consideration because the symbolic execution engine will also have to construct the symbolic object to perform veritesting. Solving such challenges will allow us to have veritesting regions with fewer exit points, thereby amplifying the benefits we get from veritesting.

2.2 Shared Expressions

Veritesting causes regions of code to be executed using static symbolic execution. Symbolic formulas representing the static symbolic execution are then gathered at the exit points of the region and added to the path expression and symbolic store of dynamic symbolic execution. This causes large disjunctive formulas to be substituted and reused multiple times, necessitating the use of techniques like hash consing [Goto 1974], or its variants such as maximally-shared graphs [Babic 2008].To evaluate reuse of structurally equivalent expressions in SPF, consider the code shown in Listing 3. The number of loop iterations is controlled by a user-supplied value for *bound*.

```
1 // x is symbolic, bound is concrete
2 public void testSharing(int x, int bound) {
3   for(int i=0; i < bound; i++) x = x + x;
4   if ( x < -50 || x > 50) ...;
5   else ...
6 }
```

Listing 3: An example with an increasing formula size with every loop iteration

The function *testSharing* adds the value of x to itself in every loop iteration on line 3 of Listing 3. On line 4, the code branches on the value of the value of x. We symbolically executed the *testSharing* method with x set to be symbolic and *bound* set to be a concrete value. We set the minimum and maximum symbolic integer values to be -100 and 100 respectively. We increased the value of *bound* from 1 to 30 and recorded the time taken for complete path coverage. Figure 3 shows the trend seen in running time and memory usage for increasing values of *bound*. Figure 3a shows that the running time remains small until the value for *bound* is 18, and then starts to rise exponentially. Figure 3b shows that at a value of 17 for *bound*, the memory usage starts to rise from 240 MB and has reached 910 MB when *bound* equals 21. We also observed that the number of expression objects undergoes a linear increase with the value of *bound*. These three observations lead us to the hypothesis that while SPF does share expression objects internally, the traversal of such expressions breaks the sharing and causes an increase in time and memory.

2.3 Complex Expressions

During exploration, SPF creates conjunctions of expressions and adds them to its *PathCondition* to determine satisfiability of paths. These expressions are allowed to have a *Comparator* (a comparison operator such as !=) as the top-level operator; however, comparison and Boolean operators are not allowed in sub-expressions. Thus, the current set of SPF expressions is insufficiently expressive to represent the disjunctive formulas required for multi-path regions, and we are currently refactoring the SPF expression hierarchy to allow for richer expression types.

3. IMPLEMENTING VERITESTING FOR JAVA

In order to implement veritesting for SPF, we are leveraging existing tools and framework features within SPF. The trickiest implementation aspects involve determining the bounds of static code regions over Java bytecodes and the mechanism for switching from "standard" symbolic execution using the SPF *SymbolicListener* class to one that can evaluate these multi-path code regions. We briefly describe our prototype implementations for these aspects.

3.1 Soot-based analysis for veritesting

Veritesting requires static construction of predicates of a multi-path region which represent changes to the path expression of the dynamic symbolic executor. It also requires construction of a control-flow graph of method bodies from Java bytecode and finding exit points of the region, which in turn requires creation of a control-flow graph of the region. Implementing veritesting is made simpler by using a static single assignment (SSA) [Bilardi and Pingali 1999] representation of the multi-path region. Using an SSA form allows us to use the ϕ-expressions created by the SSA form and translate them into points at the end of the veritesting region where updates to system state along different paths in the region can be merged. Soot [Vallée-Rai et al. 1999] is a static analysis framework for Java programs that has both these features, with ExceptionalUnitGraph [Sable Research Group 2017b] and the Shimple IR [Sable Research Group 2017a]. For simple regions with only one exit point, like the one presented in Listing 1, we were able to use Soot to automate static construction of the predicate representing an update to the expression. For doing this, we used nodes with more than one successor as the starting point, found the immediate post-dominator of the starting point, and traversed the control-flow graph on all sides of such branches. During such a traversal, we constructed predicates representing the multi-path region, similar to the ones presented in Listing 2. As explained in Section 2.1, including virtual function invocations in the construction of our predicates amplifies the benefits of veritesting even further. We plan to automate this inclusion in the future using Soot. Providing SPF with updates to be made to its symbolic store also requires Soot to maintain stack location information for variables. We plan to automate SPF's symbolic store updates using Soot in the future.

3.2 Multi-path Symbolic Execution with SPF

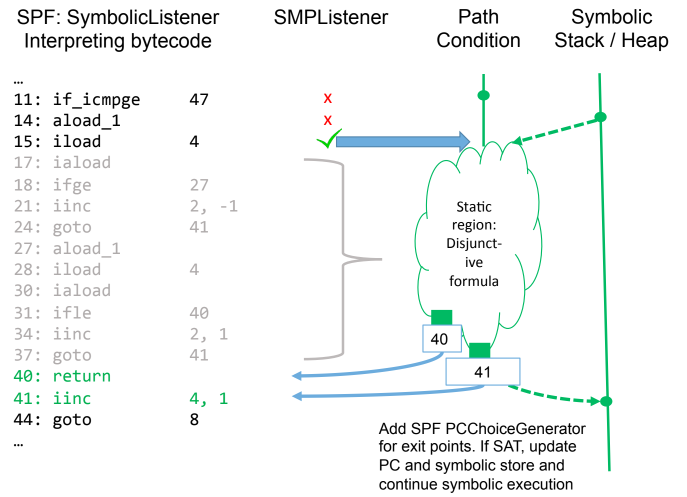
Integration of veritesting requires changing Symbolic PathFinder so that it can use a Soot-based analysis. We present the sequence of actions SPF must take to implement veritesting in Figure 4.

```
for (int i = 0; i < len; i++) {
  if (x[i] < 0) sum += -1;
  else if (x[i] > 0) sum += 1;
  else return;
}
```

Listing 4: Listing 1 modified to have a return statement in the three-way branch

The bytecode used in Figure 4 was obtained by compiling source code shown in Listing 4. This integration assumes our prior Soot-based analyis provides SPF with a table that maps instruction offsets (representing the start of a veritesting region) to a set containing (1) the multi-path region predicate to be added to the path expression as a conjunction, (2) symbolic store updates, (3) exit points, (4) the expression to branch on to one of the exit points. Using SPF's listener mechanism, we add a listener (named

Figure 4: Veritesting with Symbolic PathFinder



SMPListener in Figure 4) which listens for instructions that are starting points for a Symbolic Multi-Path (SMP) region. On finding such an instruction, *SMPListener* (1) updates the path expression, which may involve using the symbolic stack and/or heap, (2) updates the symbolic stack and heap, (3) creates a branch (using SPF's *PCChoiceGenerator*) to jump to one of the exit points (which are instructions at offsets 40 and 41 in Figure 4). SPF will then continue plain symbolic execution. Thus, veritesting causes SPF to explore fewer branches. For example, SPF only explores a two-way branch in Figure 4.

4. RELATED WORK

The original idea for veritesting was presented by Avgerinos et al. [Avgerinos et al. 2014]. They implemented veritesting on top of MAYHEM [Cha et al. 2012], 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` [Shoshitaishvili et al. 2016]. 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 [Visser et al. 2012] 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. [Sen et al. 2015]. 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 [Bodden et al. 2011] 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.

5. CONCLUSION

Veritesting provides a way to address the scalability challenges faced by dynamic symbolic execution. Nevertheless, integrating veritesting with symbolic execution at the Java bytecode level presents unique challenges. Our Soot-based analysis of six large open-source Java projects confirms that, while a number of opportunities to apply veritesting arise organically in Java bytecode, the benefit of veritesting can be amplified by adding "just in time" support of virtual function invocations and techniques for reducing the number of exit points introduced by exceptions and early return statements. In addition, careful integration with Symbolic PathFinder is necessary to achieve good performance.

6. ACKNOWLEDGEMENTS

We wish to thank the Google Summer of Code program for supporting this research. This material is based on work supported by the National Science Foundation under Grant Number 1563920.

7. REFERENCES

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

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

[Babic 2008] Domagoj Babic. 2008. *Exploiting structure for scalable software verification*. Ph.D. Dissertation. PhD thesis, University of British Columbia, Vancouver, Canada.

[Bilardi and Pingali 1999] Gianfranco Bilardi and Keshav Pingali. 1999. *The static single assignment form and its computation*. Technical Report.

[Bodden et al. 2011] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 241–250.

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

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

[Goto 1974] Eiichi Goto. 1974. *Monocopy and associative algorithms in an extended lisp*. Technical Report. TR 74-03, University of Tokyo.

[Just et al. 2014] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are Mutants a Valid Substitute for Real Faults in Software Testing?. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 654–665. `https://doi.org/10.1145/2635868.2635929`

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

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

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

[Sable Research Group 2017a] McGill University Sable Research Group. 2017a. A Brief Overview Of Shimple. `https://github.com/Sable/soot/wiki/A-brief-overview-of-Shimple`. (2017).

[Sable Research Group 2017b] McGill University Sable Research Group. 2017b. Exceptional Unit Graph (Soot API). `https://www.sable.mcgill.ca/soot/doc/soot/toolkits/graph/ExceptionalUnitGraph.html`. (2017).

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

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

[Sharma et al. 2017] Vaibhav Sharma, Kesha Hietala, and Stephen McCamant. 2017. Finding Semantically-Equivalent Binary Code By Synthesizing Adaptors. *ArXiv e-prints* (July 2017). arXiv:cs.SE/1707.01536

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

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

[Vallée-Rai et al. 1999] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '99)*. IBM Press, 13–24.

[Visser et al. 2012] Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. 2012. Green: Reducing, Reusing and Recycling Constraints in Program Analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. ACM, New York, NY, USA, Article 58, 11 pages.