JRex: Higher-Order Static Regions for Efficient Symbolic Execution of Java

ABSTRACT

Scaling symbolic execution to industrial-sized programs is an important open research problem. Veritesting introduced bounded static regions in symbolic execution to improve scalability by combining the advantages of static symbolic execution with those of dynamic symbolic execution. Bounded static regions reduce the number of paths to explore in symbolic execution by 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 many more non-local control jumps caused by runtime polymorphism, exceptions, native calls, and dynamic class loading. If these languages features are not accounted for, the static code regions described by veritesting are often small and may not lead to substantial reduction in paths. In addition, the use of disjunctive regions forces previously concrete assignments to become symbolic leading to additional solver calls.

In this paper, we describe JRex, which adds static regions to Java Symbolic Pathfinder. Unlike previous approaches, it adds support for *higher order* Veritesting regions into which we can instantiate static regions for staticly- and dynamically-dispatched function calls. Although our tool is still an unoptimized prototype, we show that it dramatically outperforms Java Symbolic Pathfinder on a number of benchmark examples, and that support for higher-order regions substantially improves the performance of Veritesting for Java programs.

Keywords

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

1. INTRODUCTION

Symbolic execution is a popular analysis technique that performs non-standard execution of a program by treating each path as a predicate. 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], checking correctness of transport protocols [Sun et al. 2017]. 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 [Kuznetsov et al. 2012] 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 [Avgerinos et al. 2014] 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 [Avgerinos et al. 2014], 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.

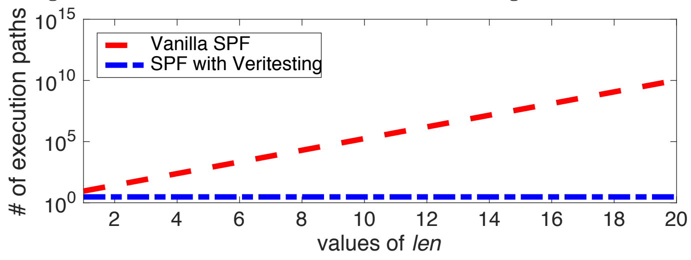
```
1  // x = ArrayList of symbolic integers with
2  // concrete length
3  for (int i = 0; i < x.size(); i++) {
4    // Begin region for static unrolling
5    if (x.get(i) < 0) sum += -1;
6    else if (x.get(i) > 0) sum += 1;
7    // End region for static unrolling
8  }
9  if (sum < 0) System.out.println("neg");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

Listing 1: 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. 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 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 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.

Unfortunately, as originally proposed, Veritesting would be un-

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



able 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 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.

```
; 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 #x10000000) (= sum1 #x10000000))
    (or (and (bvsgt x1 #x10000000) (= sum1 #x10000001))
        (and (bvslt x1 #x10000000) (= 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 $len = 2$

2. PRELIMINARIES

2.1 CFG

A control flow graph (CFG) in computer science is a representation, using graph notation, of all paths that might be traversed through a program during its execution. In a control flow graph each node in the graph represents a basic block, i.e. a straight-line piece of code without any jumps or jump targets; jump targets start a block, and jumps end a block. Directed edges are used to represent jumps in the control flow. There are, in most presentations, two specially designated blocks: the entry block, through which control enters into the flow graph, and the exit block, through which all control flow leaves.

2.2 SPF

Symbolic PathFinder (SPF) [Păsăreanu et al. 2013] combines symbolic execution with model checking and constraint solving for test case generation. In this tool, programs are executed on symbolic inputs representing multiple concrete inputs. Values of variables are represented as numeric constraints, generated from analysis of the code structure. These constraints are then solved to generate test inputs guaranteed to reach that part of code. Essentially SPF performs symbolic execution for Java programs at the bytecode level. Symbolic PathFinder uses the analysis engine of the Ames JPF model checking tool (i.e. jpf-core) [Visser et al. 2003].

2.3 Veritesting

Veritesting[Avgerinos et al. 2014] is a technique that reduces the number of paths that need to be explored by avoiding unnecessary forking. Veritesting identifies regions of if-statements that can be statically explored and captured in a logical formula (usually as disjunction of formulas representing different branches in if-statement). We will call this formula VeriFormula which captures possible different execution paths. The result of this process is a VeriFormula, which can be submitted to a SMT solver. If the formula is satisfiable, then there is some path through the code region that reaches the exit point. In this case, dynamic symbolic execution is resumed after updating the PC with the VeriFormula and also updating symbolic values of variable if needed.

3. ARCHITECTURE

3.1 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].

3.2 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. To support that use used Green[Visser et al. 2012] AST.

4. EXPERIMENTS

5. DISCUSSION

6. 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.

7. CONCLUSION

8. REFERENCES

[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.

[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.

[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`

[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.

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

[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.

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