Veritesting in Symbolic Execution of Java

ABSTRACT

Path explosion problem is still the main obsticale against scaling up symbolic execution to industrial sized projects. One interesting resolution to the problem is *Veritesting*, which represents regions of code as disjunctive formals over paths. Unlike the C compiler that inlines functions in programs, Integrating veritesting with Java bytecode presents unique challenges: notably, incorporating non-local control jumps caused by runtime polymorphism, exceptions, native calls, and dynamic 3 class loading. In this paper we present our robust implementation of Java based veritesting tool that supports dynamic dispatch.

Keywords

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

1. INTRODUCTION

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.

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

[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

[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

[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

[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