Challenges and Opportunities with Concolic Testing

Raghudeep Kannavara[1], Christopher J Havlicek[1], Bo Chen[2], Mark R Tuttle[1], Kai Cong[1], Sandip Ray[1], Fei Xie[2]
[1]Intel Corporation, 2111 NE 25th Ave, Hillsboro, OR, USA 97124
[2]Portland State University, 1825 SW Broadway, Portland, OR, USA 97201

***Abstract*—Although concolic testing is increasingly being explored as a viable software verification technique, its adoption in mainstream software development and testing in the industry is not yet extensive. In this paper, we discuss challenges to widespread adoption of concolic testing in an industrial setting and highlight further opportunities where concolic testing can find renewed applicability.**

Keywords – concolic testing; security testing; dynamic analysis; pre-silicon validation; malware analysis; firmware testing;

I. INTRODUCTION

The growing complexity of today's software demands sophisticated software analysis tools and techniques to enable the development of robust, reliable and secure software. Moreover, increasing usage of third party libraries or plugins where source code is not readily available presents additional challenges to effective software testing [11]. The cost to fix bugs prior to releasing the software is often much lower than the cost to fix bugs post release, especially in the case of security bugs. A number of automated software testing tools and techniques are commonly used in the industry. While automation significantly reduces the overhead of manual testing, finding deeply embedded security defects is not always automatable. Furthermore, automated software testing is prone to false positives or false negatives. Hence, there is a burgeoning need to advance the state of the art in software testing. In this context, concolic testing is starting to emerge as a promising technique to enable better software verification. However, the adoption of concolic testing in mainstream software development and testing activities in the industry is not yet widespread due to multifold reasons, as explained in this paper. On the other hand, we also find that there are further opportunities where concolic testing can be successfully applicable.

II. PRELIMINARIES

The goal of software verification is to assure that software fully satisfies all the expected requirements. The fundamental approaches to software verification are static analysis and dynamic analysis.

A. Static Analysis

Static program analysis is the analysis of computer software that is performed without actually executing programs [3]. In most cases the analysis is performed on some version of the source code and in the other cases some form of the object code. The term is usually applied to the analysis performed by an automated tool, with human analysis being called program understanding, program comprehension or code review. On the contrary, automated static analysis tools do not support all programming languages, produce numerous false positives or false negatives and are only as good as the rules they use to scan with. Moreover, vulnerabilities introduced during runtime are not detected.

B. Dynamic Analysis

Dynamic program analysis is the analysis of computer software that is performed by executing programs on a real or virtual processor. For dynamic program analysis to be effective, the target program must be executed with sufficient test inputs to produce interesting behavior. Fuzz testing or fuzzing is a dynamic analysis technique, often automated or semi-automated, that involves providing invalid, unexpected, or random data to the inputs of a computer program. The program is then monitored for exceptions such as crashes, or failing built-in code assertions or for finding potential memory leaks. Fuzzing is commonly used to test for security problems in software or computer systems. Unfortunately, fuzzing offers fairly shallow coverage, because many of the inputs needed to reach new code paths are exceedingly unlikely to be generated purely by chance. Hence, the difficult corner cases that can be exploited to subvert system security may be left undiscovered.

III. CONCOLIC TESTING

Automated test case generation is essential to reduce the laborious task of generating test cases manually. A promising alternative to fuzzing is concolic testing. Concolic testing is a hybrid software verification technique that interleaves concrete execution, i.e., testing on particular inputs, with symbolic execution, a classical technique that treats program variables as symbolic variables [20]. While formal verification techniques can be helpful in proving the correctness of systems, the main focus of concolic testing techniques are rather to find defects in real-world software.

A high-level overview of a concolic testing scheme is as shown in figure 1. The software under test is executed concretely to obtain the execution trace consisting of the branches taken along with any other pertinent information. The execution trace is then fed to a symbolic execution engine (SEE) for symbolic execution. The SEE re-executes the concrete trace symbolically. It treats certain variables as having symbolic or unknown values, and it keeps track of the expressions involving these symbolic values that are built up as it executes the sequence of operations in the trace. Those branches in the trace with conditionals involving symbolic values are branches whose outcomes could be changed with an appropriate choice of concrete values for the symbolic values. A constraint solver can find these concrete values, and produce

a new test case exercising a new path through the program. These new test cases are then, in turn, concretely executed. The process repeats until all possible paths through the software have been explored or a user specified constraint is satisfied.

IV. Challenges With Concolic Testing

The promise of concolic testing is not without challenges that have hindered the adoption of concolic testing by broader software testing community. In this section, we attempt to enumerate these concerns and explain the potential tradeoffs to address them.

A. Expensive constraint solving

A constraint solver is a program that computes solutions to logic formulas in a given logic and is an integral part of symbolic execution. The performance of the constraint solver used by a symbolic execution technique considerably affects the overall performance. Programs that generate very large symbolic representations that cannot be solved in practice cannot be tested effectively. To address this problem, Erete et al. propose to use domain and contextual information to optimize the performance of constraint solvers during symbolic execution [8]. While such approaches can improve the constraint solver performance for a specific use-case, they are not applicable across the board. Moreover, the expertise required to optimize constraint solvers is not generic and requires a much better understanding of the structure or properties of the software under test to inform and improve constraint solvers.

B. Issues with symbolic representations

Qu et al. highlight the limitations of concolic testing in handling float/double data types and pointers [1]. Most constraint solvers do not support float or double data types, and some solvers are unable to handle non-linear arithmetic constraints. When the constraint is beyond the ability of the solver, symbolic execution is unable to proceed farther along that path and switches to explore different paths. Moreover, while analyzing programs written in C/C++, making a pointer or a memory model as symbolic can lead to rapid state explosion. While hybridizing concolic testing can overcome this limitation by substituting symbolic inputs with concrete values to allow continued exploration of a stalled path, the technique devolves to a form of random testing and loses the advantage of exploring new behaviors [4].

C. Handling multithreaded programs

Real-world programs are often large, complex, concurrent or multithreaded and interrupt driven resulting in inherent non-determinism of such programs. This represents a key bottleneck in symbolic execution. In order to use concolic testing for multithreaded programs, Sen et al. determine the partial order relation or the exact race conditions between the various events in the execution path, for a given concrete execution at runtime [9]. Subsequently, they systematically re-

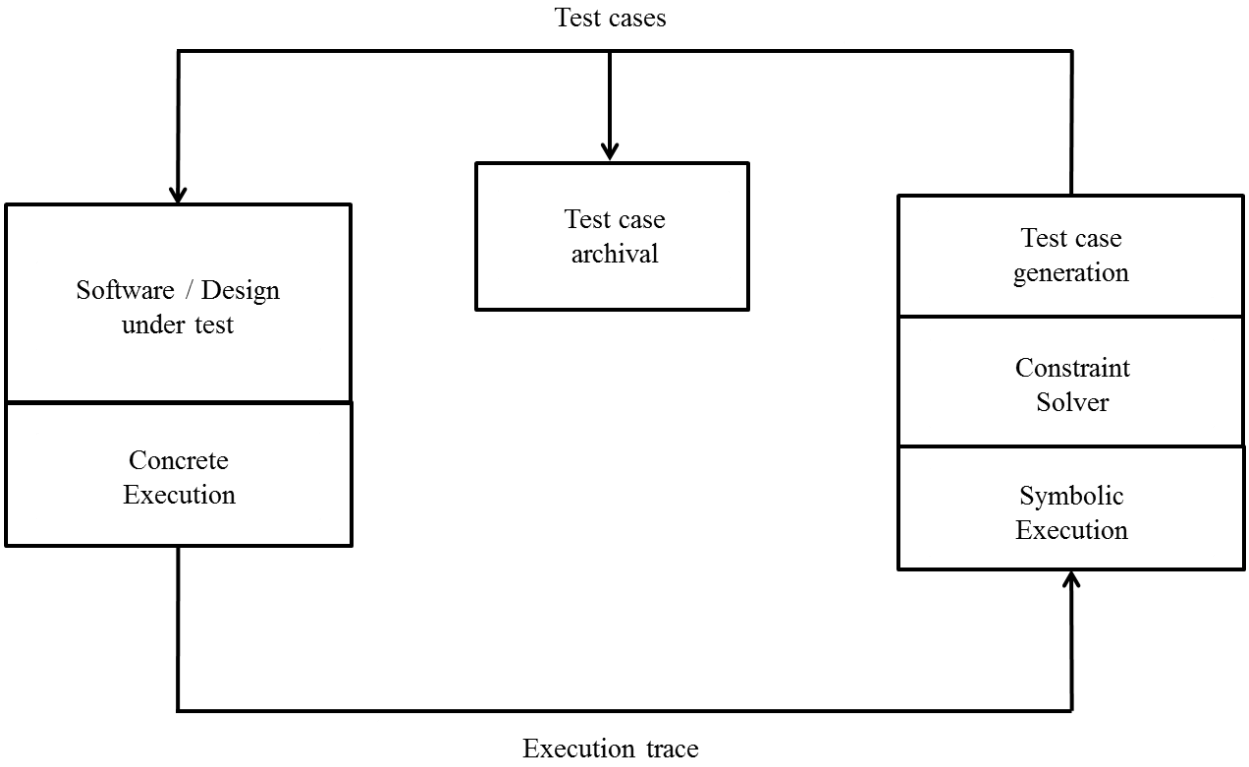

Figure 1. Concolic Testing Overview

order or permute the events involved in these races by generating new thread schedules as well as generate new test inputs thus exploring one representative from each partial order. While this is a feasible testing algorithm for concurrent programs, some potential bugs can be missed.

D. Scalability issues

Concolic testing suffers from scalability concerns. Industrial or enterprise software is usually complex, composed of multiple components with independent functionalities and specific dependencies. Testing such complex software as a single unit is not feasible. To address such limitations, Kim et al. develop Scalable COncolic testing for REliable software (SCORE) framework [2]. The SCORE framework employs a distributed concolic algorithm that can utilize a large number of computing nodes in a scalable manner so as to achieve a linear increase in the speed of test case generation as the number of distributed nodes increases while having a low communication overhead among distributed nodes. But such distributed systems unusually introduce an overhead in terms of additional time, cost, hardware and labor resources required to use and maintain them.

E. Path explosion

Another key challenge of symbolic execution is the huge number of programs paths resulting in a huge search space. This causes failure to search the most fruitful portion of a large or exponential path tree leading to poor coverage. Hence, given a fixed time budget, it is critical to explore the most relevant paths first. Numerous solutions have been explored to constrain path explosion. Jaffar et al. propose a method based on interpolation to mitigate path explosion [5]. Whenever an unsatisfiable path condition is fed to the solver, an interpolant is generated at each program point along the path. The interpolant at a given program point can be seen as a formula that captures the reason of infeasibility of paths at the program point. As a result, if the program point is encountered again through a different path such that the interpolant is implied, the new path can be subsumed. Other approaches include pruning redundant paths by tracking the memory locations read and written by the checked code, in order to determine when the remainder of a particular execution is capable of exploring new behaviors [10]. On the flip side, such optimizations could reduce path coverage.

F. Handling native calls or system calls

Calls to standard libraries, third party libraries or system calls that are not really in scope for automated testing represent a unique challenge. Limiting the scope of concolic testing to precisely the software under test is often times not possible due to interrupt handling and runtime dependencies that can lead program execution and trace generation beyond the software under test where the constraint solver has limited visibility. Dinges et al. mitigate this risk by developing a Concolic Walk technique which splits the path condition into linear and non-linear constraints, finds a point in the polytope induced by the linear constraints with an off-the-shelf solver, and then, starting from this point, uses adaptive search within the polytope, guided by the constraint fitness functions, to find a solution to the whole path condition [7]. Although this approach constrains the software under test within the polytope to limit testing, it can generalize poorly and may not scale easily.

G. Integration

Even if every one of the technical challenges just described is solved, concolic testing will remain little-used by the industry until the problem of the test harness is addressed. In simple applications of concolic testing, the test harness is a simple piece of code that identifies the variables to be treated symbolically, and invokes the software under test with these symbolic values. This is particularly easy when the software under test is a purely functional piece of code whose behavior depends only on its inputs. But this never happens in practice, including firmware, where the behavior of code depends on values buried in enormous data structures built up over time or by elaborate system initialization procedures. The test harness must essentially become a model of the environment of the software under test. Building a test harness can be hard work, and is overhead that may not be acceptable to development and validation teams. There are exceptions, of course, such as companies like Microsoft where concolic testing has been extraordinarily successful. For those exceptions, however, the model of the environment is simple enough (e.g., a set of files) to generate automatically. In general, the lack of automation for concolic testing stands in stark contrast to static analysis tools like Klocwork [21] that have been deeply integrated into overnight builds to produce effective bug reports the next morning with little effort on the part of developers. Without a comparable level of integration into production development and testing environments, concolic testing runs the risk of being left behind, even with its ability to find high-quality bugs that may be undiscovered by other tools.

V. Further Opportunities With Concolic Testing

Concolic testing has extensively been explored to enable software verification. Several optimization techniques have been proposed to address the concerns associated with concolic testing to make it more practicable. There are numerous opensource projects such as KLEE [12], CREST [13] and proprietary solutions such as Microsoft SAGE [14] that are specifically designed to harness the power of concolic testing to aid software testing. But beyond verifying software, there are further opportunities where concolic testing can find renewed applicability while not being constrained by its existing limitations. In this section, we highlight some of those prominent opportunities.

A. Testing embedded software

Firmware is low-level software which can directly access hardware and is often shipped with the hardware platform. Firmware is continuing to increase in scale and importance in contemporary electronic devices. Moreover, firmware based security attacks are becoming prevalent and are difficult to detect or to fix in deployed products. Thus firmware validation is a critical part of system validation. Ahn et al. present a firmware validation approach based on automatically generating a test-set for the firmware with the goal of complete path coverage while considering its interactions with hardware and other firmware threads [15]. They use a service-function based Transaction Level Model (TLM) to harness concolic testing to generate tests that are directly used for the firmware

transaction and account for the multi-threaded interactions. Kim et al. follow a similar approach to firmware validation and highlight the importance of modifying the firmware build process to instrument the firmware and modelling the specialized environment required for firmware execution in order to harness concolic testing for firmware validation [16].

B. *Pre-silicon validation*

Pre-silicon validation refers to validation activities performed on a simulation or emulation model that transpires prior to fabricating actual silicon. Pre-silicon validation verifies the correctness and sufficiency of the design. Pre-silicon validation activities are critical to ensure product quality. Detecting and fixing issues early in the hardware development life cycle enables the manufacturer to stay competitive while delivering reliable and high quality products, especially in a high volume manufacturing industry like the semiconductor industry [6]. Fei et al. present a concolic testing approach to generate pre-silicon and post-silicon tests with virtual prototypes [18]. They identify device states under test from concrete executions of a virtual prototype based on the concept of device transaction, symbolically execute the virtual prototype from these device states to generate tests, and issue the generated tests concretely to the silicon device. With this approach, they observed significant coverage improvement with generated test cases and detected inconsistencies between virtual prototypes and silicon devices, exposing several virtual prototype or silicon device defects. Such approaches help validation engineers to easily and more precisely understand a silicon device using its virtual prototype, identify defects in the silicon device and detect bugs in the virtual prototype.

C. *Analyzing malware*

Existing signature based malware detection techniques are defenseless against mutants or polymorphic variations. Besides, modern malware is equipped with intelligent detour techniques against Sandbox detection mechanisms to nullify the defense system. To counter advancements in malware, automated analysis tools are required to analyze programmatic behavior of massive amounts of malicious codes. In order to enable better malware analysis, Joo et al. propose the use of concolic testing technique based event generator to trigger malicious behavioral routines in suspicious programs, not executed during normal conditions [17]. On the other hand, while concolic testing has been proven powerful in security analysis, it also provides a sharp scalpel for attacks like software cracking and piracy [19]. To counter this, control flow obfuscation techniques that aim to confuse the automated analyzers by obfuscating the programs' control flow structures are used to defend against software cracking and piracy.

VI. CONCLUSIONS

The ever increasing software complexity warrants continuous advancements in software testing methodologies. Concolic testing is progressively moving beyond academic research and gaining ground in the software industry as a viable software testing methodology. While no single method is a silver bullet to find all types of software defects, a combination of tools and techniques is the most practicable approach to software testing. Concolic testing is no more different in this case. Software test engineers use an array of tools and methodologies at their disposal to enable finding and fixing software defects early in the software development lifecycle. The choice of tools and techniques are driven by resource constraints, time to market and engineer expertise. In general, reducing the barriers to entry to concolic testing is one direction to improve software quality.

REFERENCES

[1] Xiao Qu, Brian Robinson, "A Case Study of Concolic Testing Tools and their Limitations," International Symposium on Empirical Software Engineering and Measurement, Banff, Canada, September 22-23, 2011

[2] M.Kim, Y.Kim and G.Rothermel, "A Scalable Distributed Concolic Testing Approach: An Empirical Evaluation," IEEE International Conference on Software Testing, Verification and Validation (ICST), Montreal, Canada, April 17-21, 2012

[3] Raghudeep Kannavara, "Securing Opensource Code via Static Analysis," Fifth International Conference on Software Testing, Verification & Validation, Montreal, Canada, April 17-21, 2012

[4] Rupak Majumdar, Koushik Sen, "Hybrid Concolic Testing," 29th International Conference on Software Engineering, Minneapolis, USA, 20-26 May 2007

[5] Joxan Jaffar, Vijayaraghavan Murali, Jorge A. Navas, "Boosting concolic testing via interpolation," ESEC/FSE 2013, Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, Pages 48-58

[6] Raghudeep Kannavara, "Towards a Unified Framework for Pre-Silicon Validation," Fourth International Conference on Information, Intelligence, Systems and Applications, Piraeus-Athens, Greece, July 10–12, 2013

[7] Peter Dinges, Gul Agha, "Solving complex path conditions through heuristic search on induced polytopes," 22nd ACM SIGSOFT Symposium on Foundations of Software Engineering. Hong Kong, November 16-21 2014

[8] Ikpeme Erete, Alessandro Orso, "Optimizing Constraint Solving to Better Support Symbolic Execution," ICSTW '11 Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, Pages 310-315

[9] Koushik Sen, Gul A Agha, "Concolic Testing of Multithreaded Programs and Its Application to Testing Security Protocols," Technical Report, UIUCDCS-R-2006-2676, 2006-01

[10] Peter Boonstoppel, Cristian Cadar, Dawson Engler, "RWset: Attacking Path Explosion in Constraint-Based Test Generation," Tools and Algorithms for the Construction and Analysis of Systems Lecture Notes in Computer Science Volume 4963, 2008, pp 351-366

[11] Raghudeep Kannavara, "Assessing the Threat Landscape for Software Libraries," 25th IEEE International Symposium on Software Reliability Engineering, Naples, Italy, November 3-6, 2014

[12] Cristian Cadar, Daniel Dunbar, Dawson Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," USENIX Symposium on Operating Systems Design and Implementation, San Diego, CA, USA, December 8-10, 2008

[13] Jacob Burnim, Koushik Sen, "Heuristics for Scalable Dynamic Test Generation," 23rd IEEE/ACM International Conference on Automated Software Engineering, Pages 443-446

[14] Patrice Godefroid, Michael Y. Levin, David Molnar, "SAGE: Whitebox Fuzzing for Security Testing," Magazine Queue - Networks, Volume 10 Issue 1, Pages 20, January 2012

[15] Sunha Ahn, Sharad Malik, "Automated Firmware Testing using Firmware-Hardware Interaction Patterns," 2014 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), New Delhi, India, 12-17 Oct. 2014

[16] Moonzoo Kim, Yunho Kim, Yoonkyu Jang, "Industrial Application of Concolic Testing on Embedded Software: Case Studies," IEEE International Conference on Software Testing, Verification and Validation (ICST), Montreal, Canada, April 17-21, 2012

[17] Jung-Uk Joo, Incheol Shin, Minsoo Kim, "Efficient Methods to Trigger Adversarial Behaviors from Malware during Virtual Execution in

SandBox," International Journal of Security and Its Applications, Vol.9, No.1 (2015), pp.369-376

[18] Kai Cong, Fei Xie, Li Lei, "Automatic Concolic Test Generation with Virtual Prototypes for Post-silicon Validation" International Conference on Computer-Aided Design, Austin, USA, November 2-6, 2015

[19] Haoyu Ma, Xinjie Ma, Weijie Liu, Zhipeng Huang, Debin GAO, Chunfu Jia, "Control Flow Obfuscation using Neural Network to Fight Concolic Testing," 10th International ICST Conference on Security and Privacy in Communication Networks (SecureComm 2014), Beijing, China, September 24–26, 2014

[20] Koushik Sen, Darko Marinov, Gul Agha, "CUTE: A Concolic Unit Testing Engine for C," 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, Lisbon, Portugal, September 05-09, 2005

[21] Rogue Wave Software, "The Business Case for Earlier Software Defect Detection and Compliance," QuinStreet, Inc. 2014