Online Appendix to: A Survey of Symbolic Execution Techniques

ROBERTO BALDONI, Cyber Intelligence and Information Security Research Center, Sapienza
EMILIO COPPA, SEASON Lab, Sapienza University of Rome
DANIELE CONO D'ELIA, SEASON Lab, Sapienza University of Rome
CAMIL DEMETRESCU, SEASON Lab, Sapienza University of Rome
IRENE FINOCCHI, SEASON Lab, Sapienza University of Rome

A. ADDITIONAL TABLES

Tools. Table I lists a number of symbolic execution engines that have worked as incubators for several of the techniques surveyed in this article. The novel contributions introduced by tools that represented milestones in the area are described in the appropriate sections throughout the main article.

Path Selection Heuristics. Table II provides a categorization of the search heuristics that have been discussed in Section 2.3 of the main article. For each category, we list several works that have proposed interesting embodiments of the category.

B. SYMBOLIC EXECUTION OF BINARY CODE

The importance of performing symbolic analysis of program properties on binary code is on the rise for a number of reasons. Binary code analysis is attractive as it reasons on code that will actually execute: not requiring the source code significantly extends the applicability of such techniques (to, e.g., common off-the-shelf proprietary programs, firmwares for embedded systems, and malicious software), and it gives the ground truth important for security applications whereas source code analysis may yield misleading results due to compiler optimizations [Song et al. 2008]. Binary analysis is

Symbolic engine	**References**	**Project URL** (last retrieved: December 2017)
CUTE	[Sen et al. 2005]	–
DART	[Godefroid et al. 2005]	–
JCUTE	[Sen and Agha 2006]	https://github.com/osl/jcute
KLEE	[Cadar et al. 2006; Cadar et al. 2008]	https://klee.github.io/
SAGE	[Godefroid et al. 2008; Elkarablieh et al. 2009]	–
BITBLAZE	[Song et al. 2008]	http://bitblaze.cs.berkeley.edu/
CREST	[Burnim and Sen 2008]	https://github.com/jburnim/crest
PEX	[Tillmann and De Halleux 2008]	http://research.microsoft.com/en-us/projects/pex/
RUBYX	[Chaudhuri and Foster 2010]	–
JAVA PATHFINDER	[Pasareanu and Rungta 2010]	http://babelfish.arc.nasa.gov/trac/jpf
OTTER	[Reisner et al. 2010]	https://bitbucket.org/khooyp/otter/
BAP	[Brumley et al. 2011]	https://github.com/BinaryAnalysisPlatform/bap
CLOUD9	[Bucur et al. 2011]	http://cloud9.epfl.ch/
MAYHEM	[Cha et al. 2012]	–
SYMDROID	[Jeon et al. 2012]	–
S^2E	[Chipounov et al. 2012]	http://s2e.systems/
FUZZBALL	[Martignoni et al. 2012; Caselden et al. 2013]	http://bitblaze.cs.berkeley.edu/fuzzball.html
JALANGI	[Sen et al. 2013]	https://github.com/Samsung/jalangi2
PATHGRIND	[Sharma 2014]	https://github.com/codelion/pathgrind
KITE	[do Val 2014]	http://www.cs.ubc.ca/labs/isd/Projects/Kite
SYMJS	[Li et al. 2014]	–
CIVL	[Siegel et al. 2015]	http://vsl.cis.udel.edu/civl/
KEY	[Hentschel et al. 2014]	http://www.key-project.org/
ANGR	[Shoshitaishvili et al. 2015; Shoshitaishvili et al. 2016]	http://angr.io/
TRITON	[Saudel and Salwan 2015]	http://triton.quarkslab.com/
PYEXZ3	[Ball and Daniel 2015]	https://github.com/thomasjball/PyExZ3
JDART	[Luckow et al. 2016]	https://github.com/psycopaths/jdart
CATG	–	https://github.com/ksen007/janala2
PYSYMEMU	–	https://github.com/feliam/pysymemu/
MIASM	–	https://github.com/cea-sec/miasm

Table I: Selection of symbolic execution engines, along with their reference article(s) and software project web site (if any).

Heuristic	**Goal**
BFS	*Maximize coverage* [Chipounov et al. 2012; Tillmann and De Halleux 2008]
DFS	*Exhaust paths, minimize memory usage* [Cadar et al. 2006; Chipounov et al. 2012] [Tillmann and De Halleux 2008; Godefroid et al. 2005]
Random path selection	*Randomly pick a path with probability based on its length* [Cadar et al. 2008]
Code coverage search	*Prioritize paths that may explore unexplored code or that may soon reach a particular target program point* [Cadar et al. 2006; Cadar et al. 2008; Cha et al. 2012] [Chipounov et al. 2012; Groce and Visser 2002; Ma et al. 2011]
Buggy-path-first	*Prioritize bug-friendly path* [Avgerinos et al. 2011]
Loop exhaustion	*Fully explore specific loops* [Avgerinos et al. 2011]
Symbolic instruction pointers	*Prioritize paths with symbolic instruction pointers* [Cha et al. 2012]
Symbolic memory accesses	*Prioritize paths with symbolic memory accesses* [Cha et al. 2012]
Fitness function	*Prioritize paths based on a fitness function* [Xie et al. 2009; Cadar and Sen 2013; Xie et al. 2009]
Subpath-guided search	*Use frequency distributions of explored subpaths to prioritize less covered parts of a program* [Li et al. 2013]
Property-guided search	*Prioritize paths that are most likely to satisfy the target property* [Zhang et al. 2015]

Table II: Common path selection heuristics discussed in Section 2.3.

relevant also for programs written in dynamic languages and executed in runtimes that deeply transform and optimize the code through just-in-time compilation.

Working on binary code is often a challenging task for many program analyses due to its complexity and lack of a high-level semantics. Modern architectures offer complex instruction sets: modeling each instruction can be difficult, especially in the presence of multiple side effects on processor flags to determine branch conditions. The second major challenge comes from the high-level semantics of the source code being lost in the lowering process (Figure 12), especially when debugging information is absent. Types are not explicitly encoded in binary code: even with register types, it is common to read values assuming a different type (e.g., 8-bit integer) from what was used to store them (e.g., 16-bit integer). Similar considerations can be made for array bounds as well. Also, control flow graph information is not explicitly available, as control flow is performed through jump instructions at both inter- and intra-procedural level. The function abstraction at the binary level does not exist as we intend it at source-code level: functions can be separated in non-contiguous pieces, and code may also call in the middle of a code block generated for a source-level function.

In the remainder of this section we provide an overview of how symbolic executors can address some of the most significant challenges in the analysis of binary code.

B.1. Lifting to an Intermediate Representation

Motivated by the complexity in modeling native instructions and by the variety of architectures on which applications can be deployed (e.g., x86, x86-64, ARM, MIPS), symbolic executors for binary code typically rely on a *lifter* that transforms native instructions into an *intermediate representation* (IR), also known as *bytecode*. Modern compilers such as LLVM typically generate IR by *lowering* the user-provided source code during the first step of compilation, optimize it, and eventually lower it to native

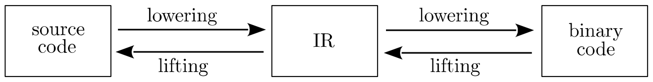

Fig. 12: Lowering and lifting processes in native vs. source code processing.

code for a specific platform. Source-code symbolic executors can resort to compiler-assisted lowering to reason on bytecode rather than source-language statements: for instance, KLEE [Cadar et al. 2008] reasons on the IR generated by the LLVM compiler for static languages such as C and C++. Figure 12 summarizes the relationships between source code, IR, and binary code.

Reasoning at the intermediate representation level allows for encoding program analyses in an architecture-agnostic fashion. Translated instructions will always expose all the side-effects of a native instruction, and support for additional platforms can be added over time. A number of symbolic executors use VEX, the IR of the Valgrind dynamic instrumentation framework [Nethercote and Seward 2007]. VEX is a RISC-like language designed for program analysis that offers a compact set of instructions to express programs in static single assignment form [Cytron et al. 1991]. Lifters are available for both 32-bit and 64-bit ARM, MIPS, PPC, and x86 binaries.

ANGR [Shoshitaishvili et al. 2016] performs analysis directly on VEX IR. Authors chose VEX over other IR formats as at that time it was the only choice that offered a publicly available implementation with support for many architectures. Also, they mention that writing a binary lifter can be a daunting task, and a well-documented and program analysis-oriented solution can be a bonus. BITBLAZE [Song et al. 2008] uses VEX too, although it translates it to a custom intermediate language. The reason for this is that VEX captures the side effects of some instructions only implicitly, such as the `EFLAGS` bits set by instructions of the x86 ISA: translating it to a custom language simplified the development of BITBLAZE's analysis framework.

The authors of S^{2}E [Chipounov et al. 2012] have implemented an x86-to-LLVM-IR lifter in order to use the KLEE [Cadar et al. 2008] symbolic execution engine for whole-system symbolic analysis of binary code in a virtualized environment. The translation is transparent to both the guest operating system and KLEE, thus enabling the analysis of binaries using the full power of KLEE. Another x86-to-LLVM-IR lifter that can be used to run KLEE on binary code is `mcsema`[1].

B.2. Reconstructing the Control Flow Graph

A control flow graph (CFG) can provide valuable information for a symbolic executor as it captures the set of potential control flow transfers for all feasible execution paths. A fundamental issue that arises when reconstructing CFGs for binaries is that the possible targets of an indirect jump may not be identified correctly. Direct jumps are straightforward to process: as they encode their targets explicitly in the code, successor basic blocks can be identified and visited until no new edge is found. The target of an indirect jump is determined instead at run time: it might be computed by carrying out a calculation (e.g., a jump table) or depend on the current calling context (e.g., a function pointer is passed as argument, or a virtual C++ method is invoked).

CFG recovery is typically an iterative refinement process based on a number of program analysis techniques. For instance, value-set analysis (VSA) [Duesterwald 2004] is a technique that can be used to identify a tight over-approximation of certain program state properties (e.g., the set of possible targets of an indirect jump or a memory write). In BITBLAZE [Song et al. 2008] an initial CFG is generated by inserting special successor nodes for unresolved indirect jump targets. This choice is conceptually

[1]https://github.com/trailofbits/mcsema.

similar to widening a fact to the bottom of a lattice in a data-flow analysis. When an analysis requires more precise information, VSA is then applied on demand.

ANGR [Shoshitaishvili et al. 2016] implements two algorithms for CFG recovery. An iterative algorithm starts from the entry point of the program and interleaves a number of techniques to achieve speed and completeness, including VSA, inter-procedural backward program slicing, and symbolic execution of blocks. This algorithm is however rather slow and may miss code portions reachable only through unresolved jump targets. The authors thus devise a fast secondary algorithm that uses a number of heuristics to identify functions based on prologue signatures, and performs simple analyses (e.g., a lightweight alias analysis) to solve a number of indirect jumps. The algorithm is context-insensitive, so it can be used to quickly recover a CFG without a concern for understanding the reachability of functions from one another.

B.3. Code Obfuscation

In recent years, code obfuscation has received considerable attention as a cheap way to hinder the understanding of the inner workings of a proprietary program. Obfuscation is employed not only to thwart software piracy and improve software security, but also to avoid detection and resist analysis for malicious software [Udupa et al. 2005; Yadegari et al. 2015].

A significant motivation behind using symbolic/concolic execution in the analysis of malware is to deal with code obfuscations. However, current analysis techniques have trouble getting around some of those obfuscations, leading to imprecision and/or excessive resource usage [Yadegari and Debray 2015]. For instance, obfuscation tools can transform conditional branches into indirect jumps that symbolic analysis find difficult to analyze, while run-time code self-modification might conceal conditional jumps on symbolic values so that they are missed by the analysis.

A few works have described obfuscation techniques aiming at thwarting symbolic execution. [Sharif et al. 2008] uses one-way hash functions to devise a *conditional code obfuscation* scheme that makes it hard to identify the values of symbolic variables for which branch conditions are satisfied. They also present an encryption scheme for the code to execute based on a key derived from the value that satisfies a branch condition. [Wang et al. 2011] takes a step forward by proposing an obfuscation technique that is effective despite it uses linear operations only, for which symbolic execution usually works well. The obfuscation tool inserts a simple loop incorporating an unsolved mathematical conjecture that converges to a known value after a number of iterations, and the produced result is then combined with the original branch condition.

[Hai et al. 2016] presents BE-PUM, a tool to generate a precise CFG in the presence of obfuscation techniques that are common in the malware domain, including indirect jumps, structured exception handlers (SEHs), overlapping instructions, and self-modifying code. While engines such as BITBLAZE [Song et al. 2008] typically rely on disassemblers like IDA Pro[2], BE-PUM relies on concolic execution to deobfuscate code, using a binary emulator for the user process and stubs for API calls.

[Yadegari and Debray 2015] discusses the limitations of symbolic execution in the presence of three generic obfuscation techniques: (1) conditional-to-indirect jump transformation, also known as *symbolic jump problem* [Schwartz et al. 2010]; (2) conditional-to-conditional jump transformation, where the predicate is deeply changed; and (3) symbolic code, when code modification is carried out using an input-derived value. The authors show how resorting to bit-level taint analysis and architecture-aware constraint generation can allow symbolic execution to circumvent such obfuscations.

[2]https://www.hex-rays.com/products/ida/.

C. APPLICATIONS OF SYMBOLIC EXECUTION

[Cadar et al. 2011] observes how the recent explosion of research work in symbolic execution makes for an interesting story about the increasing impact of this program analysis since its introduction in the mid '70s. The availability of powerful off-the-shelf SMT solvers and hardware resources, along with advances in symbolic execution techniques to deal with the challenges identified in Section 1.2, facilitated the application of symbolic execution to increasing large problem instances from many domains.

In this section we do not aim at presenting a comprehensive overview of applications of symbolic execution. Our goal is instead to provide the reader with a selection of works appeared in the last few years that either incubated novel ideas that might be effective in other domains too (e.g., to deal with the path explosion problem), or significantly affected the state of the art of a specific field.

The works we are about to discuss are drawn from four domains: software testing, program understanding, bug exploitation, and authentication bypass. Other fields that have seen uses of symbolic execution, such as automatic filter generation (e.g., [Brumley et al. 2006; Costa et al. 2007]) and code analysis (e.g., [Hayden et al. 2012; Banescu et al. 2017]), are not covered here. Also, we do not address techniques tailored to programs with concurrent threads (e.g., [Bergan et al. 2014; Guo et al. 2015]) or floating-point arithmetic (e.g., [Ramachandran et al. 2015; Liew et al. 2017]).

C.1. Software Testing

Software testing strategies typically attempt to execute a program with the intent of finding bugs. As manual test input generation is an error-prone and usually non-exhaustive process, automated testing techniques have drawn a lot of attention over the years. Random testing techniques such as fuzzing are cheap in terms of run-time overhead, but fail to obtain a wide exploration of a program state space. Symbolic and concolic execution techniques on the other hand achieve a more exhaustive exploration, but they become expensive as the length of the execution grows: for this reason, they usually reveal shallow bugs only.

[Majumdar and Sen 2007] proposes *hybrid concolic testing* for test input generation, which combines random search and concolic execution to achieve both deep program states and wide exploration. The two techniques are interleaved: in particular, when random testing saturates (i.e., it is unable to hit new code coverage points after a number of steps), concolic execution is used to mutate the current program state by performing a bounded depth-first search for an uncovered coverage point. For a fixed time budget, the technique outperforms both random and concolic testing in terms of branch coverage. The intuition behind this approach is that many programs show behaviors where a state can be easily reached through random testing, but then a precise sequence of events – identifiable by a symbolic engine – is required to hit a specific coverage point.

[Stephens et al. 2016] refines this idea by devising Driller, a vulnerability excavation tool based on ANGR [Shoshitaishvili et al. 2016] that interleaves fuzzing and concolic execution to discover memory corruption vulnerabilities. The authors remark that user inputs can be categorized as *general* input, which has a wide range of valid values, and *specific* input; a check for particular values of a specific input splits an application into *compartments*. Driller offloads the majority of unique path discovery to a fuzzy engine, and relies on concolic execution to move across compartments. During the fuzzy phase, Driller marks a number of inputs as interesting (for instance, when an input was the first to trigger some state transition) and once it gets stuck in the exploration, it passes the set of such paths to a concolic engine, which preconstraints the program states to ensure consistency with the results of the native execution. On the dataset used for the DARPA Cyber Grand Challenge qualifying event, Driller could identify crashing inputs in 77 applications, including both the 68 and 16 applications for which fuzzing and symbolic execution alone succeeded, respectively. For 6 applications, Driller was the only one to detect a vulnerability.

Maintenance of large and complex applications is a very hard task. Fixing bugs can sometimes introduce new and unexpected issues in the software, which in turn may require several hours or even weeks to be detected and properly addressed by the developers. [Qi et al. 2012] tackles the problem of identifying the root cause of failures during regression testing. Given a program P and a newer revision of the program P', if a testing input t generates a failure in P' but not in P, then symbolic execution is used to track the path constraints π and π' when executing P and P' on the failing input t, respectively. Using an SMT solver, a new input t' is generated by solving the formula $\pi \wedge \neg\pi'$. If t' exists (i.e., the formula is satisfiable), then P' has one or more *deviations* in the control flow graph with respect to P that can be the root cause of the failure. By carefully tracking branch conditions during symbolic execution, [Qi et al. 2012] are also able to pinpoint which branches are responsible for these deviations. If $\pi \wedge \neg\pi'$ is unsatisfiable, the symmetric formula $\neg\pi \wedge \pi'$ is evaluated and analogous actions are taken to detect possible branch conditions that may have led to the failure. If also $\neg\pi \wedge \pi'$ is unsatisfiable, the root cause of the problem cannot be determined.

Another interesting work that targets the problem of software regressions through the use of symbolic execution is [Böhme et al. 2013]. The work introduces an approach called *partition-based regression verification* that combines the advantages of both regression verification (RV) and regression testing (RT). Indeed, RV is a very powerful technique for identifying regressions but hardly scales to large programs due to the difficulty in proving behavioral equivalence between the original and the modified program. On the other hand, RT allows for checking a modified program for regressions by testing selected concrete sample inputs, making it more scalable but providing limited verification guarantees. The main intuition behind partition-based regression verification is the identification of *differential partitions*. Each differential partition can be seen as a subset of the input space for which the two program versions – given the same path constraints – either expose the same output (*equivalence-revealing partition*) or produce different results (*difference-revealing partition*). For each partition, a test case is generated and added to the regression test suite, which can later be used by a developer for classical RT. Since differential partitions are derived by exploiting symbolic execution, this approach suffers from the common limitations that come with this technique. However, if the exploration is interrupted (e.g., due to excessive time or memory usage), partition-based regression verification can still provide guarantees over the subset of input space that has been covered so far by the detected partitions.

Directed incremental symbolic execution (DiSE) is usually used for regression testing. As pointed out in the main article, its strength lies in applying static analyses in synergy with symbolic execution, directing the exploration to the sole code portions affected by changes. [Backes et al. 2013] uses DiSE to generate summaries of behaviors affected by differences, and proves behavioral equivalence of two program versions by comparing the affected behaviors only. Their approach is sound and complete for sequential programs under a given depth bound for the symbolic exploration.

Static data flow analysis tools can significantly help developers track malicious data leaks in software applications. Unfortunately, they often report several alleged bugs that only after a manual inspection can be regarded as false positives. To mitigate this issue, [Arzt et al. 2015] proposes TASMAN, a system that, after performing data-flow analysis to track information leaks, uses symbolic backward execution to test each reported bug. Starting from a leaking statement, TASMAN explores the code backwards, pruning any path that can be proved unfeasible. If all the paths starting at the leaking statement are discarded by TASMAN, the bug is deemed a false positive.

C.2. Program Understanding

While symbolic execution is largely employed in testing activities, over the few last years several works (e.g., [Geldenhuys et al. 2012; Filieri et al. 2013; Chen et al. 2016]) have shown how it can be valuable also for program understanding activities.

[Geldenhuys et al. 2012] introduces *probabilistic symbolic execution*, an approach that makes it possible to compute the probability of executing different code portions of a program. This is achieved by exploiting model counting techniques, such as the `LattE` [Loera et al. 2004] toolset, to determine the number of solutions for the different path constraints given by the alternative execution paths of a program.

The work by [Filieri et al. 2013] takes a step further by using probabilistic symbolic execution to perform software reliability analysis. Reliability is computed as the probability of executing paths that have been labeled as successful given a usage profile, which represents the input space of all the successfully accomplished external interactions (with the user and with external resources) of the program. Since in general the termination of symbolic execution cannot be guaranteed in presence of loops, the proposed technique resorts to bounded exploration. Nonetheless, the authors define a metric for evaluating the confidence of their reliability estimation, allowing a developer to increase the bounds in order to improve the confidence value.

Of a different flavor is the work by [Chen et al. 2016], which uses probabilistic symbolic execution to conduct performance analysis. Based on usage profiles and on path execution probabilities, paths are classified into two types: *low-probability* and *high-probability*. Initially, high-probability paths are explored in a way that maximizes path diversity, generating a first set of test inputs. In a second phase, low-probability paths are analyzed using symbolic execution, generating a second set of test inputs that should expose executions characterized by the best and by the worst execution times. Finally, the program is executed using the test inputs generated during the two phases, and its running time is measured to generate performance distributions.

Another interesting application of symbolic execution to program understanding is presented in [Pasareanu et al. 2016]. The technique exploits model counting and symbolic execution for computing quantitative bounds on the amount of information that can be leaked by a program through side-channel attacks.

C.3. Bug Exploitation

Bugs are a consequence of the nature of human factors in software development and are everywhere. Those that can be exploited by an attacker should normally be fixed first: systems for automatically and effectively identifying them are thus very valuable.

AEG [Avgerinos et al. 2011] employs preconditioned symbolic execution to analyze a potentially buggy program in source form and look for bugs amenable to stack smashing or return-into-libc exploits [Pincus and Baker 2004], which are popular control hijack attack techniques. The tool augments path constraints with exploitability constraints and queries a constraint solver, generating a concrete exploit when the constraints are satisfiable. The authors devise the *buggy-path-first* and *loop-exhaustion* strategies (Table II) to prioritize paths in the search. On a suite of 14 Linux applications, AEG discovered 16 vulnerabilities, 2 of which were previously unknown, and constructed control hijack exploits for them.

MAYHEM [Cha et al. 2012] takes another step forward by presenting the first system for binary programs that is able identify end-to-end exploitable bugs. It adopts a hybrid execution model based on checkpoints and two components: a concrete executor that injects taint-analysis instrumentation in the code and a symbolic executor that takes over when a tainted branch or jump instruction is met. Exploitability constraints for symbolic instruction pointers and format strings are generated, targeting a wide range of exploits, e.g., SEH-based and jump-to-register ones. Three path selection heuristics help prioritizing paths that are most likely to contain vulnerabilities (e.g., those containing symbolic memory accesses or instruction pointers). A virtualization layer intercepts and emulates all the system calls to the host OS, while preconditioned symbolic execution can be used to reduce the size of the search space. Also, restricting symbolic execution to tainted basic blocks only gives very good speedups in this setting, as in the reported experiments more than 95% of the processed instructions were not tainted. MAYHEM was able to find exploitable vulnerabilities in the 29 Linux and Windows applications considered in the evaluation, 2 of which were previously undocumented. Although the goal in MAYHEM is to reveal exploitable bugs, the generated simple exploits can be likely transformed in an automated fashion to work in the presence of classical OS defenses such as data execution prevention and address space layout randomization [Schwartz et al. 2011].

C.4. Authentication Bypass

Software backdoors are a method of bypassing authentication in an algorithm, a software product, or even in a full computer system. Although sometimes these software flaws are injected by external attackers using subtle tricks such as compiler tampering [Karger and Schell 1974], there are reported cases of backdoors that have been surreptitiously installed by the hardware and/or software manufacturers [Costin et al. 2014], or even by governments [Zitter 2013].

Different works (e.g., [Davidson et al. 2013; Zaddach et al. 2014; Shoshitaishvili et al. 2015]) have exploited symbolic execution for analyzing the behavior of binary firmwares. Indeed, an advantage of this technique is that it can be used even in environments, such as embedded systems, where the documentation and the source code that are publicly released by the manufacturer are typically very limited or none at all. For instance, [Shoshitaishvili et al. 2015] proposes Firmalice, a binary analysis framework based on ANGR [Shoshitaishvili et al. 2016] that can be effectively used for identifying authentication bypass flaws inside firmwares running on devices such as routers and printers. Given a user-provided description of a privileged operation in the device, Firmalice identifies a set of program points that, if executed, forces the privileged operation to be performed. The program slice that involves the privileged program points is then symbolically analyzed using ANGR. If any such point can be reached by the engine, a set of concrete inputs is generated using an SMT solver. These values can be then used to effectively bypass authentication inside the device. On three commercially available devices, Firmalice could detect vulnerabilities in two of them, and determine that a backdoor in the third firmware is not remotely exploitable.

REFERENCES

Steven Arzt, Siegfried Rasthofer, Robert Hahn, and Eric Bodden. 2015. Using Targeted Symbolic Execution for Reducing False-positives in Dataflow Analysis. In *Proc. of the 4th ACM SIGPLAN Int. Workshop on State Of the Art in Program Analysis (SOAP'15)*. ACM, 1–6.

Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. 2011. AEG: Automatic Exploit Generation. In *Proc. Network and Distributed System Security Symp. (NDSS'11)*.

John D. Backes, Suzette Person, Neha Rungta, and Oksana Tkachuk. 2013. Regression Verification Using Impact Summaries. In *Proc. 20th Int. SPIN Symposium on Model Checking of Software (SPIN'13)*. 99–116.

Thomas Ball and Jakub Daniel. 2015. Deconstructing Dynamic Symbolic Execution. In *Proc. 2014 Marktoberdorf Summer School on Dependable Software Systems Engineering*. IOS Press.

Sebastian Banescu, Christian Collberg, and Alexander Pretschner. 2017. Predicting the Resilience of Obfuscated Code Against Symbolic Execution Attacks via Machine Learning. In *26th USENIX Security Symp., USENIX Security 17*. USENIX Association, Vancouver, BC, 661–678.

Tom Bergan, Dan Grossman, and Luis Ceze. 2014. Symbolic Execution of Multithreaded Programs from Arbitrary Program Contexts. In *Proc. 2014 ACM Int. Conf. on Object Oriented Programming Systems Languages & Applications (OOPSLA'14)*. ACM, 491–506.

Marcel Böhme, Bruno C. d. S. Oliveira, and Abhik Roychoudhury. 2013. Partition-based Regression Verification. In *Proc. 2013 International Conference on Software Engineering (ICSE'13)*. IEEE Press, 302–311.

David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A Binary Analysis Platform. In *Proc. 23rd Int. Conf. on Computer Aided Verification (CAV'11)*. 463–469.

David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. 2006. Towards Automatic Generation of Vulnerability-Based Signatures. In *Proc. 2006 IEEE Symposium on Security and Privacy (SP'06)*. IEEE Computer Society, Washington, DC, USA, 2–16.

Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. 2011. Parallel Symbolic Execution for Automated Real-world Software Testing. In *Proc. 6th Conf. on Comp. Systems (EuroSys'11)*. 183–198.

Jacob Burnim and Koushik Sen. 2008. Heuristics for Scalable Dynamic Test Generation. In *Proc. 23rd IEEE/ACM Int. Conf. on Automated Software Engineering (ASE'08)*. IEEE Computer Society, 443–446.

Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proc. 8th USENIX Conf. on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, 209–224.

Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2006. EXE: Automatically Generating Inputs of Death. In *Proc. 13th ACM Conf. on Computer and Communications Security (CCS'06)*. ACM, 322–335.

Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Pasareanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. 2011. Symbolic Execution for Software Testing in Practice: Preliminary Assessment. In *Proc. 33rd Inter. Conf. on Software Engineering*. ACM, 1066–1071.

Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (2013), 82–90.

Dan Caselden, Alex Bazhanyuk, Mathias Payer, Stephen McCamant, and Dawn Song. 2013. HI-CFG: Construction by Binary Analysis and Application to Attack Polymorphism. In *18th European Symp. on Research in Computer Security*. 164–181.

Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *Proc. 2012 IEEE Symp. on Sec. and Privacy (SP'12)*. IEEE Comp. Society, 380–394.

Avik Chaudhuri and Jeffrey S. Foster. 2010. Symbolic Security Analysis of Ruby-on-rails Web Applications. In *Proc. 17th ACM Conf. on Computer and Communications Security (CCS 2010)*. ACM, 585–594.

Bihuan Chen, Yang Liu, and Wei Le. 2016. Generating Performance Distributions via Probabilistic Symbolic Execution. In *Proc. 38th Int. Conf. on Software Engineering (ICSE'16)*. ACM, 49–60.

Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2012. The S2E Platform: Design, Implementation, and Applications. *ACM Transactions on Computer Systems (TOCS)* 30, 1 (2012), 2:1–2:49.

Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. 2007. Bouncer: Securing Software by Blocking Bad Input. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07)*. ACM, New York, NY, USA, 117–130.

Andrei Costin, Jonas Zaddach, Aurelien Francillon, and Davide Balzarotti. 2014. A Large-Scale Analysis of the Security of Embedded Firmwares. In *Proc. 23rd USENIX Security Symp.* 95–110.

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 4 (1991), 451–490.

Drew Davidson, Benjamin Moench, Somesh Jha, and Thomas Ristenpart. 2013. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *Proc. 22nd USENIX Conf. on Security (SEC 2013)*. USENIX Association, 463–478.

Celina Gomes do Val. 2014. *Conflict-Driven Symbolic Execution: How to Learn to Get Better*. MSc Thesis. University of British Columbia.

Evelyn Duesterwald (Ed.). 2004. *Analyzing Memory Accesses in x86 Executables*. Springer.

Bassem Elkarablieh, Patrice Godefroid, and Michael Y. Levin. 2009. Precise Pointer Reasoning for Dynamic Test Generation. In *Proc. 18th Int. Symp. on Software Testing and Analysis (ISSTA'09)*. ACM, 129–140.

Antonio Filieri, Corina S. Pasareanu, and Willem Visser. 2013. Reliability Analysis in Symbolic Pathfinder. In *Proc. 2013 Int. Conf. on Software Engineering (ICSE'13)*. IEEE Press, Piscataway, NJ, USA, 622–631.

Jaco Geldenhuys, Matthew B. Dwyer, and Willem Visser. 2012. Probabilistic Symbolic Execution. In *Proc. 2012 Int. Symp. on Software Testing and Analysis (ISSTA'12)*. ACM, 166–176.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'05)*. 213–223.

Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated Whitebox Fuzz Testing. In *Proc. Network and Distributed System Security Symp. (NDSS'08)*.

Alex Groce and Willem Visser. 2002. Model Checking Java Programs Using Structural Heuristics. In *Proc. 2002 ACM SIGSOFT Int. Symp. on Software Testing and Analysis (ISSTA'02)*. ACM, 12–21.

Shengjian Guo, Markus Kusano, Chao Wang, Zijiang Yang, and Aarti Gupta. 2015. Assertion Guided Symbolic Execution of Multithreaded Programs. In *Proc. 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'15)*. ACM, 854–865.

Nguyen Minh Hai, Mizuhito Ogawa, and Quan Thanh Tho. 2016. Obfuscation Code Localization Based on CFG Generation of Malware. In *Proc. 8th Int. Symp on Foundations and Practice of Security (FPS'15)*, Joaquin Garcia-Alfaro, Evangelos Kranakis, and Guillaume Bonfante (Eds.). Springer Int. Publishing, 229–247.

Christopher M. Hayden, Stephen Magill, Michael Hicks, Nate Foster, and Jeffrey S. Foster. 2012. Specifying and Verifying the Correctness of Dynamic Software Updates. In *Proc. 4th Int. Conf. on Verified Software: Theories, Tools, Experiments (VSTTE'12)*. Springer-Verlag, Berlin, Heidelberg, 278–293.

Martin Hentschel, Richard Bubel, and Reiner Hähnle. 2014. Symbolic Execution Debugger (SED). In *Proc. 5th Int. Conf. on Runtime Verification (RV'14)*. 255–262.

Jinseong Jeon, Kristopher K. Micinski, and Jeffrey S. Foster. 2012. *SymDroid: Symbolic Execution for Dalvik Bytecode*. Technical Report CS-TR-5022. Depart. of Computer Science, Univ. of Maryland, College Park.

Paul A. Karger and Roger R. Schell. 1974. *Multics security evaluation: Vulnerability analysis*. Technical Report. HQ Electronic Systems Division: Hanscom AFB, MA.

Guodong Li, Esben Andreasen, and Indradeep Ghosh. 2014. SymJS: Automatic Symbolic Testing of JavaScript Web Applications. In *Proc. 22nd ACM SIGSOFT Int. Symp. on Foundations of Software Engineering (FSE'14)*. ACM, 449–459.

You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. 2013. Steering Symbolic Execution to Less Traveled Paths. In *Proc. ACM SIGPLAN Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA'13)*. 19–32.

Daniel Liew, Daniel Schemmel, Cristian Cadar, Alastair F. Donaldson, Rafael Zähl, and Klaus Wehrle. 2017. Floating-point Symbolic Execution: A Case Study in N-version Programming. In *Proc. 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17)*. IEEE Press, 601–612.

Jess A. De Loera, Raymond Hemmecke, Jeremiah Tauzer, and Ruriko Yoshida. 2004. Effective lattice point counting in rational convex polytopes. *Journal of Symbolic Computation* 38, 4 (2004), 1273 – 1302. Symbolic Computation in Algebra and Geometry.

Kasper Luckow, Marko Dimjašević, Dimitra Giannakopoulou, Falk Howar, Malte Isberner, Temesghen Kahsai, Zvonimir Rakamarić, and Vishwanath Raman. 2016. JDart: A Dynamic Symbolic Analysis Framework. In *Proc. 22nd Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'16)*. 442–459.

Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. 2011. Directed Symbolic Execution. In *Proc. 18th Int. Conf. on Static Analysis (SAS'11)*. 95–111.

Rupak Majumdar and Koushik Sen. 2007. Hybrid Concolic Testing. In *Proc. 29th Int. Conf. on Software Engineering (ICSE'07)*. IEEE Computer Society, 416–426.

Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. 2012. Path-exploration Lifting: Hi-fi Tests for Lo-fi Emulators. In *Proc. Seventeenth Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. ACM, 337–348.

Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proc. 28th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'07)*. ACM, 89–100.

C. S. Pasareanu, Q. S. Phan, and P. Malacaria. 2016. Multi-run Side-Channel Analysis Using Symbolic Execution and Max-SMT. In *2016 IEEE 29th Computer Security Foundations Symp. (CSF'16)*. 387–400.

Corina S. Pasareanu and Neha Rungta. 2010. Symbolic PathFinder: Symbolic Execution of Java Bytecode. In *Proc. IEEE/ACM Int. Conf. on Automated Software Engineering (ASE'10)*. ACM, 179–180.

Jonathan Pincus and Brandon Baker. 2004. Beyond stack smashing: recent advances in exploiting buffer overruns. *IEEE Security Privacy* 2, 4 (2004), 20–27.

Dawei Qi, Abhik Roychoudhury, Zhenkai Liang, and Kapil Vaswani. 2012. DARWIN: An Approach to Debugging Evolving Programs. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 21, 3, Article 19 (2012).

Jaideep Ramachandran, Corina S. Pasareanu, and Thomas Wahl. 2015. Symbolic Execution for Checking the Accuracy of Floating-Point Programs. *ACM SIGSOFT Software Engineering Notes* 40, 1 (2015), 1–5.

Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S. Foster, and Adam Porter. 2010. Using Symbolic Evaluation to Understand Behavior in Configurable Software Systems. In *Proc. 32nd ACM/IEEE Int. Conf. on Software Engineering (ICSE'10)*. ACM, 445–454.

Florent Saudel and Jonathan Salwan. 2015. Triton: A Dynamic Symbolic Execution Framework. In *Symp. sur la Sécurité des Technologies de l'Information et des Communications (SSTIC'15)*. 31–54.

Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proc. 2010 IEEE Symp. on Security and Privacy (SP'10)*. IEEE Computer Society, 317–331.

Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2011. Q: Exploit Hardening Made Easy. In *Proc. 20th USENIX Conf. on Security (SEC'11)*. USENIX Association, 25–25.

Koushik Sen and Gul Agha. 2006. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-checking Tools. In *Proc. 18th Int. Conf. on Computer Aided Verification (CAV'06)*. 419–423.

Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: A Selective Record-replay and Dynamic Analysis Framework for JavaScript. In *Proc. 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, 488–498.

Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proc. 10th European Software Engineering Conf. Held Jointly with 13th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering (ESEC/FSE'13)*. ACM, 263–272.

Monirul I. Sharif, Andrea Lanzi, Jonathon T. Giffin, and Wenke Lee. 2008. Impeding Malware Analysis Using Conditional Code Obfuscation. In *Proc. Network and Distributed System Security Symp. (NDSS'08)*.

Asankhaya Sharma. 2014. Exploiting Undefined Behaviors for Efficient Symbolic Execution. In *Companion Proc. 36th Int. Conf. on Software Engineering (ICSE Companion 2014)*. ACM, 727–729.

Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *22nd Annual Network and Distributed System Security Symp. (NDSS'15)*.

Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symp. on Security and Privacy (SP'16)*. 138–157.

Stephen F. Siegel, Manchun Zheng, Ziqing Luo, Timothy K. Zirkel, Andre V. Marianiello, John G. Edenhofner, Matthew B. Dwyer, and Michael S. Rogers. 2015. CIVL: The Concurrency Intermediate Verification Language. In *Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis (SC'15)*. ACM, Article 61.

Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proc. 4th Int. Conf. on Information Systems Security ((ICISS'08)*. 1–25.

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 *23nd Annual Network and Distr. System Sec. Symp. (NDSS'16)*.

Nikolai Tillmann and Jonathan De Halleux. 2008. Pex: White Box Test Generation for .NET. In *Proc. 2nd Int. Conf. on Tests and Proofs (TAP'08)*. 134–153.

Sharath K. Udupa, Saumya K. Debray, and Matias Madou. 2005. Deobfuscation: Reverse Engineering Obfuscated Code. In *Proc. 12th Working Conf. on Reverse Engineering (WCRE'05)*. IEEE Computer Society, 45–54.

Zhi Wang, Jiang Ming, Chunfu Jia, and Debin Gao. 2011. Linear Obfuscation to Combat Symbolic Execution. In *Proc. 16th European Symp. on Research in Computer Security (ESORICS'11)*, Vijay Atluri and Claudia Diaz (Eds.). Springer Berlin Heidelberg, 210–226.

Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. 2009. Fitness-guided path exploration in dynamic symbolic execution. In *Proc. 2009 IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN'09)*. 359–368.

Babak Yadegari and Saumya Debray. 2015. Symbolic Execution of Obfuscated Code. In *Proc. 22nd ACM SIGSAC Conf. on Computer and Communications Security (CCS'15)*. ACM, 732–744.

Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. 2015. A Generic Approach to Automatic Deobfuscation of Executable Code. In *Proc. 2015 IEEE Symp. on Security and Privacy (SP'15)*. IEEE Computer Society, 674–691.

Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. 2014. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *21st Annual Network and Distributed System Security Symp. (NDSS'14)*.

Yufeng Zhang, Zhenbang Chen, Ji Wang, Wei Dong, and Zhiming Liu. 2015. Regular Property Guided Dynamic Symbolic Execution. In *Proc. 37th Int. Conf. on Software Engineering (ICSE'15)*. 643–653.

Kim Zitter. 2013. How a Crypto Backdoor Pitted the Tech World Against the NSA. (2013). https://www.wired.com/2013/09/nsa-backdoor/all/.