Looking at Challenges and Mitigation in Symbolic Execution Based Fuzzing Through the Lens of Survey Papers

Valentin Huber

November 28, 2023

Abstract

Contents

1 Introduction

- "Today, testing is the primary way to check the correctness of software. Billions of dollars are spent on testing in the software industry, as testing usually accounts for about 50% of the cost of software development. It was recently estimated that software failures currently cost the US economy alone about $60 billion every year, and that improvements in software testing infrastructure might save one-third of this cost." [1]
- "The attack (WannaCry) startled the global economy by hitting its impact on around 230K–300K computers in about 150 countries, leading to an estimated substantial financial impact of US $4–$8 billion worldwide" [2]
- "Software testing is the most commonly used technique for validating the quality of software, but it is typically a mostly manual process that accounts for a large fraction of software development and maintenance." [3]
- Fuzzing is one of several software vulnerability techniques. [4]
- "Compared with other techniques, fuzzing requires few knowledge of targets and could be easily scaled up to large applications, and thus has become the most popular vulnerability discovery solution, especially in the industry." [5]
- "The term "fuzz" was originally coined by Miller et al. in 1990 to refer to a program that "generates a stream of random characters to be consumed by a target program" [6]" [7]
- "There are many different dynamic analyses that can be described as "fuzzing." A unifying feature of fuzzers is that they operate on, and produce, concrete inputs. Otherwise, fuzzers might be instantiated with many different design choices and many different parameter settings." [8]
- "Google could find 20K vulnerabilities in Chrome using fuzz testing" [2]
- Fuzzing is used by lots of big players: Google, Microsoft, DoD, Cisco, Adobe all employ fuzzing as part of their secure development practices, and many of those have contributed to or written their own open-source or commercial fuzzers [2].

2 Theoretical Principles

- Program under test (PUT)
- Two approaches: Random mutation as described in *An Empirical Study of the Reliability of UNIX Utilities* by Miller et al. [6] and pure symbolic execution, as introduced in [9].
- The latter is infeasible for large programs and for any program that interacts with the environment, the former in its purest form is not very effective.
  - "A key disadvantage of classical symbolic execution is that it cannot generate an input if the symbolic path constraint along a feasible execution path contains formulas that cannot be (efficiently) solved by a constraint solver (for example, nonlinear constraints)." [10]
  - "The blackbox and whitebox strategies achieved similar results in all categories. This shows that, when testing applications with highly-structured inputs in a limited amount of time (2 hours), whitebox fuzzing, with the power of symbolic execution, does not improve much over simple blackbox fuzzing. In fact, in the code generator, those grammar-less strategies do not improve coverage much above the initial set of seed inputs." [11]
- Generally:
  - "The process begins by choosing a corpus of "seed" inputs with which to test the target program. The fuzzer then repeatedly mutates these inputs and evaluates the program under test. If the result produces "interesting" behavior, the fuzzer keeps the mutated input for future use and records what was observed. Eventually the fuzzer stops, either due to reaching a particular goal (e.g., finding a certain sort of bug) or reaching a timeout." [8]
  - "Different fuzzers record different observations when running the program under test. In a "black box" fuzzer, a single observation is made: whether the program crashed. In "gray box" fuzzing, observations also consist of intermediate information about the execution, for example, the branches taken during execution as determined by pairs of basic block identifiers executed directly in sequence. "White box" fuzzers can make observations and modifications by exploiting the semantics of application source (or binary) code, possibly

involving sophisticated reasoning. Gathering additional observations adds overhead. Different fuzzers make different choices, hoping to trade higher overhead for better bug-finding effectiveness." [8]
  - "In any of these cases, the output from the fuzzer is some concrete input(s) and configurations that can be used from outside of the fuzzer to reproduce the observation. This allows software developers to confirm, reproduce, and debug issues." [8]
- "Many of these tools also automatically find well-defined bugs, such as assertion errors, divisions by zero, NULL pointer dereferences, etc." [12]

3 Related Works and Methods

- Large scientific body of work
- Review papers
  - Well cited
  - New
  - Specific topics to see if challenges and solutions differ
- Gathered by
  - Search engines
  - Cited in review papers
  - Lists in review papers (as in [2])
  - Cited in important primary papers

3.1 General Survey Papers

- Fuzzing: The State of the Art (Feb. 2012) [13]
- An orchestrated survey of methodologies for automated software test case generation (Aug. 2013) [14]
- A Survey of Symbolic Execution Techniques (May 2018) [15]
- A systematic review of fuzzing techniques (Jun. 2018) [16]
- Fuzzing: State of the Art (Sep. 2018) [17]
- Fuzzing: hack, art, and science (Jan. 2020) [18]
- A Systematic Review of Search Strategies in Dynamic Symbolic Execution (Oct. 2020) [19]
- A Survey of Hybrid Fuzzing based on Symbolic Execution (Jan. 2021) [20]
- Fuzzing: Challenges and Reflections (May 2021) [21]
- Exploratory Review of Hybrid Fuzzing for Automated Vulnerability Detection (Sep. 2021) [22]
- Fuzzing vulnerability discovery techniques: Survey, challenges and future directions (Sep. 2022) [23]

3.1.1 *All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)* (Jul. 2010) [12]

Using a simple intermediate language (SIMPIL), this paper discusses taint analysis and forward symbolic execution, including examples and analysis of the theoretical foundations of symbolic execution. While fuzzing is mentioned in multiple instances, it is not the main focus. However, it still lists many of the drawbacks and advantages fuzzers based on symbolic execution have, and the additional perspective was valuable in assembling this review.

3.1.2 *Symbolic Execution for Software Testing in Practice: Preliminary Assessment* (May 2011) [3]

After giving a short overview of issues faced by symbolic execution based fuzzers, this paper focuses on eight high impact fuzzing tools (JPF-SE and Symbolic (Java) PathFinder [24], [25], DART [1], CUTE [26] and jCUTE [27], CREST [28], SAGE [29], Pex [30], EXE [31], and KLEE [32]).

3.1.3 *Symbolic Execution for Software Testing: Three Decades Later* (Feb. 2013) [10]

This survey paper, as the title suggests, focuses on symbolic execution. Starting with an explanation of classical symbolic execution, it then provides a list of issues that fuzzing tools based on symbolic execution face, along with attempts to mitigate those by adapting and extending the algorithms. Finally, the authors present five high-impact tools they worked on: DART [1], CUTE [26], CREST [28], EXE [31], and KLEE [32].

3.1.4 *Symbolic Execution for Software Testing: Three Decades Later* (Feb. 2013) [10]

While not a classic survey paper, *Evaluating Fuzz Testing* finds issues in how all 32 papers performed the evaluation of the system they introduced. It further proposes rules to follow to make an evaluation

robust. Last, it contains a list of what advances each paper examined claims to introduce.

3.1.5 *Fuzzing: a survey* (Jun. 2018) [5]

Li *et al.* focus on coverage-guided fuzzing, mentioning other approaches that can be mixed in and different applications it can be used for. They further categorize whitebox fuzzers in generation based fuzzers, mentioning SPIKE [33] (whose website and source seems to no longer be available), Sulley [34] (with BooFuzz [35] as a currently maintained fork), and Peach [36], and mutation based fuzzers with the only mention being Miller [37]. These are only mentioned, without any discussion. Last, they broadly discuss the challenges symbolic execution in fuzzing faces and, in an other section, present TaintScope [38] and Driller [39] as examples of using symbolic execution for specifically for path exploration.

3.1.6 *The Art, Science, and Engineering of Fuzzing: A Survey* (Nov. 2021) [7]

Starting with proposing a taxonomy for fuzzing itself and categorizing fuzzers, this paper proposes a general-purpose model of fuzzing, explaining the steps and approaches common fuzzers share. It further presents a genealogy, tracing the origins of important papers back to the work of Miller *et al.* However, it "does not provide a comprehensive survey on DSE" [7], but only discusses whitebox fuzzing in a subsection and refers to other survey papers such as [12], [14] for a more complete overview.

3.1.7 *Fuzzing: A Survey for Roadmap* (Sep. 2022) [40]

Similar to what is attempted in this paper, *Fuzzing: A Survey for Roadmap* lists issues along common steps in fuzzing along with attempted solutions, but without the focus on symbolic execution. It does contain a short section about symbolic execution in the context input search space handling, but only discusses very few papers directly while often mentioning entire families of papers, with only some relying on symbolic execution.

3.1.8 *A systematic review of fuzzing* (Oct. 2023) [41]

The authors of this paper guide the reader through advances in fuzzing along the works that introduced those. It includes a section about symbolic execution, which considers the following systems: Driller [39], QSYM [42], SAVIOR [43], DigFuzz [44], Pangolin [45], and QuickFuzz [46].

3.1.9 *Demystify the Fuzzing Methods: A Comprehensive Survey* (Oct. 2023) [2]

This paper dedicates one of its chapter to first explaining the fundamental logic of symbolic execution, and then presenting three implementations (Driller [39], CONFETTI [47], and FUZZOLIC [48]). It further investigates advances in IoT firmware and kernel fuzzers, but does not explain where up- and downsides of using symbolic execution in these domains lay.

3.2 Specific Survey Papers

- Network protocol fuzz testing for information systems and applications: a survey and taxonomy (Nov. 2016) [49]
- A Survey of Dynamic Analysis and Test Generation for JavaScript (Sep. 2017) [50]
- Fuzzing the Internet of Things: A Review on the Techniques and Challenges for Efficient Vulnerability Discovery in Embedded Systems (2021) [51]
- Firmware Fuzzing: The State of the Art (2021) [52]
- Research on Fuzzing Technology for JavaScript Engines (2021) [53]
- Ethereum Smart Contract Analysis Tools: A Systematic Review (2022) [54]
- Embedded fuzzing: a review of challenges, tools, and solutions (Sep. 2022) [55]
- Fuzzing of Embedded Systems: A Survey (Dec. 2022) [56]
- A Survey on the Development of Network Protocol Fuzzing Techniques (2023) [57]

4 Challenges and Mitigation

Heavily based on [3], [10], extended based on information from all other information considered as listed in Section 3.

4.1 Impossible Constraints

- "A key disadvantage of classical symbolic execution is that it cannot generate an input if the symbolic path constraint along a feasible execution path contains formulas that cannot be (efficiently) solved by a constraint solver (for example, nonlinear constraints)." [10]
- Two techniques that alleviate this problem:

  - Dynamic Symbolic Execution (DSE), or Concolic Testing (like DART [1] and its successor CUTE [26], and CREST [28]): Run concrete and symbolic execution at the same time, keep mapping between values, solve path constraint with one sub-constraint flipped to get input for an other path. "A key observation in DART is that imprecision in symbolic execution can be alleviated using concrete values and randomization" [3]
  - Execution-Generated Testing (EGT) [58] (like EXE [31] and KLEE [32]): Only execute symbolically if any operands are symbolic

- These handle imprecision in symbex (like interaction with outside code (that is not instrumented for symbex), constraint solving timeouts, unhandled instructions (floating point), or system calls (`read`, interrupts, etc.)) by just using concrete values.

  - If none of the operands are symbolically, just use them
  - If any are, use the concrete values (direct in concolic, solution from path constraint in EGT)

- Downside: missing some feasible paths, and therefore sacrificing completeness.
- Further Ideas: Special Constraint Solvers that improve floating point based constraint handling (like FloPSy [59]) and complex mathematical constraints (like CORAL [60])

4.2 System Calls

- "Additionally, symbolic execution creates conflicts while handling system calls, since it does not support modeling all possible system calls and inter-process communication, such as pipes or sockets. Likewise, the non-deterministic behavior of system calls complicates the generation of inputs that consistently trigger specific paths." [2]
- HFL [61] is a kernel fuzzer that heavily relies on symbolic execution. It lists three main issues the authors had to overcome: "(1) indirect control transfers determined by system call arguments (2) controlling and matching internal system state via system calls, and (3) nested argument type inference for invoking system calls" [61]. To solve those issues, HFL "(1) converts implicit control transfers to explicit transfers, (2) infers system call sequence to build a consistent system state, and (3) identifies nested arguments types of system calls" [61].

4.3 Environment Interaction

- "[. . . KLEE's [32]] ability to handle interactions with the outside environment — e.g., with data read from the file system or over the network — by providing models designed to explore all possible legal interactions with the outside world." [3]

4.4 Path Explosion

Path Explosion: program path count usually exponential in the number of static branches in the code.

- Simple depth-first search, however this naïve approach gets stuck in non-terminating loops with symbolic conditions and is therefore rarely used. Both EXE [31] and KLEE [32] can however be configured to run in this mode.
- Symbex inherently helps by only looking at possible branches. Example: EXE [31] on `tcpdump`: only 42% of instructions contained symbolic operands, less than 20% of of symbolic branches have both sides feasible [31]
- Prioritization of which path to explore next using heuristics (like statement or branch coverage (and using static analysis to guide), favouring statements that were run the fewest number of times, or random). Examples: EXE [31], SAGE [29], CREST [28]

  - Interleave random and symbolic execution. Examples: Hybrid Concolic Testing [39], [62], [63]
  - Guide towards changes in a patch: Directed Incremental Symbolic Execution [64] and Directed Test Suite Augmentation [65]
  - Parallel state-space search algorithms like generational search (which generates multiple new inputs from a single symbolic execution): SAGE [29], Eclipser [66]
  - "CREST [28] is an extensible platform for building and experimenting with heuristics for selecting which paths to explore" [10]
  - "we propose a novel approach called Fitnex, a search strategy that uses state-dependent fitness values (computed

through a fitness function) to guide path exploration. The fitness function measures how close an already discovered feasible path is to a particular test target (e.g., covering a not-yet-covered branch)" [67]
  - Let users select uninteresting parts of the code and avoid it (as in [68])
  - Weigh the approximate cost of executing a certain path against its demand, as done in QuickFuzz [46]
  - Probabilistic approach: Use Monte Carlo path optimization to quantify the difficulty of each path using grey-box fuzzing to then let the white-box fuzzer focus on the paths that are believed to be most challenging for grey-box fuzzing to make progress. [44]
  - Guide execution towards code parts deemed to be interesting based on static analysis, such as pointer dereferences in loops as implemented in Dowser [69], potential bugs according to UndefinedBehaviorSanitizer [70] in SAVIOR [43] or more general prior static or dynamic program analysis such as in GRT [71] or VUzzer [72] to guide the symbolic execution engine.

- Pruning redundant paths ("if a program path reaches the same program point with the same symbolic constraints as a previously explored path, then this path will continue to execute exactly the same from that point on and thus can be discarded." [73]) Eliminating redundant paths by analyzing the values read and written by the program.
- Sharing among states/copy on write: KLEE [32]
- Caching function summaries for later use by higher-level functions. Example: SMART [74]
- Lazy test generation (as in LATEST [75])
- Static path merging (as in KLEE-FP [76])
- partial order and symmetry reductions (as in GSE [77])
- Compact representation of path constraints (as in SAGE [29])

4.5 Constraint Solving

Dominates runtime

- Irrelevant constraint elimination: Generally, we go from a solvable constraint set (namely the current execution with the solution being the current concrete values) to one where only one constraint changes (the one we flipped). Typically, major major parts of the constraint set are not influenced by the change and can be excluded from what is passed to the solver. We can then just use the values from the previous iteration. This is implemented, among others, in [29].
- Identify independent sub-queries and solve them independently, as is done in EXE [31] and KLEE [32].
- Optimizing SMT queries before passing them to the solver. The optimization itself, however, can already be too complex to compute to employ this strategy effectively.
- Mocking and stubbing: Moles [78]
- Incremental solving: Reuse the results of previous similar queries, because subsets of the constraints are still solved by the same results and supersets often do not invalidate existing solutions. (CUTE [26] and counterexample caching scheme in KLEE [32])
- Cache prior SMT query results and reuse them for future queries. Pangolin [45] uses polyhedral path abstraction to replace query parts for more efficient models based on prior results.
- Improved SMT solvers (like Z3 [79] used in e.g. SAGE [29], STP [80], or cvc5 [81] built during the development of EXE [31])
- Intriguer [82] uses taint analysis to discover instructions accessing a wide range of input bytes, and then performs symbolic execution for those instructions deemed important and only invoke the underlying SMT solver for complicated queries.
- Approximate SMT solvers, such as FUZZY-SAT implemented in FUZZOLOGIC [48], or optimistic SMT solvers, as in QSYM [42], generate interesting inputs based on a SMT query without relying on classic (and expensive) SMT solvers.
- Similarly, Eclipser [66] uses instrumentation on the PUT to generate partial path conditions, which can then be solved without invoking SMT solvers to generate further inputs.
- Do not use intermediate representation (IR) to execute symbolically, but integrate the symbolic emulation with the native execution through dynamic binary translation, which prevents additional instructions (since often multiple RISC instructions are necessary to replace one CISC instruction), and allows finer-grained control

over the constraint, thus making it smaller. Example: QSYM [42]

- SMT formulas can be transformed into programs, which in turn can then be solved using a coverage-guided fuzzer to generate solutions to the initial formula. JFI [83] uses this technique to find solutions to floating-point constraints.

4.6 Memory Modelling

Things like modelling `int`s as mathematical integers being imprecise since it ignores over-/underflows, and pointers being hard to deal with.

- Issue: Dereferencing symbolic pointer, as in pointer which can be influenced from the input.
  - "A sound strategy is to consider it a load from any possible satisfying assignment for the expression." [12]
  - "Symbolic memory addresses can lead to aliasing issues even along a single execution path. A potential address alias occurs when two memory operations refer to the same address." [12]
  - (Potentially) unsound assumptions: optionally rewrite all memory addresses as scalars based on name, like Vine [84]
  - Pass the dealiazing step to the SMT solver like CVC Lite [85] or STP [80].
  - Perform alias analysis. However, like in DART [1], "part of the allure of forward symbolic execution is that it can be done at run-time" [12].
  - EXE [31] and KLEE [32] "perform a mix of alias analyses and letting the SMT solver worry about aliasing" [12]
  - Other systems like DART [1] and CUTE [26] cannot handle non-linear constraints and therefore cannot deal with symbolic references.
- "On the one end of the spectrum is a system like DART [1] that only reasons about concrete pointers, or systems like CUTE [26] and CREST [28] that support only equality and inequality constraints for pointers, which can be efficiently solved. [26] At the other end are systems like EXE [31], and more recently KLEE [32] and SAGE [29] that model pointers using the theory of arrays with selections and updates implemented by solvers like STP or Z3." [10]

4.7 Handling Concurrency

- Testing usually difficult because of the inherent non-determinism.
- "Concolic testing was successfully combined with a variant of partial order reduction to test concurrent programs effectively. [27], [86]–[88]" [10]
- "Generalized Symbolic Execution [77] performs symbolic execution by leveraging an off-the-shelf model checker, whose built-in capabilities allow handling multi-threading (and other forms of non-determinism)" [3]
- "This method requires that a sequential version of the program be provided, to serve as the specification for the parallel one. The key idea is to use model checking, together with symbolic execution, to establish the equivalence of the two programs." [89] (complex parallel numerical computations)

4.8 Recursive Data Structures

- "GSE handles input recursive data structures by using lazy initialization. GSE starts execution of the method on inputs with uninitialized fields and non-deterministically initializes fields when they are first ac- cessed during the method's symbolic execution." [3]
- "Pex [30] supports the generation of test inputs of primitive types as well as (recursive) complex data types, for which Pex automatically computes a factory method which creates an instance of a complex data type by invoking a constructor and a sequence of methods, whose parameters are also determined by Pex." [3]

4.9 Symbolic Jump Addresses

Symbolic target addresses of jump instructions are an obvious issue for symbolic execution based systems. Standard ways of handling these include:

- Concolic execution: Perform and trace the execution of a program under test, let it jump to the concrete address observed during this run, and finally perform symbolic execution on the trace. This leaves some potentially possible program states unexplored. Examples include CUTE [26].
- Pass the reasoning issue to the SMT solver. This however makes the SMT queries more complicated and since constraint solving is already

an issue in many cases (see Section 4.5), this may not solve the issue after all.

- Use static analysis to locate possible jump targets.

4.10 System Calls and OS Interactions

System calls (such as calls to `read` or interrupts) pose an obvious obstacle to pure symbolic execution, since they may introduce new symbolic variables or, more importantly, have side effects. This can be mitigated by manually creating summaries of these side effects (as done in EXE [31] and KLEE [32]), or, again, employing concolic execution with all the upsides and drawbacks discussed before.

4.11 Selective Symbolic Execution

Not exhaustive.

- Tools like Driller [39] perform classical fuzzing (in the case of Driller using AFL [90]) until they are *stuck*, meaning they are unable to produce inputs that discover additional paths. Driller then, and only then, invokes its concolic execution engine (Driller uses angr [91]) to trace the program under investigation executed with one of the previously generated inputs. Finally, it solves the resulting path constraints with one condition flipped to produce an input that will reach new parts of the software. Because Driller does not use symbolic execution for its primary discovery tool, it does not suffer from issues such as path explosion, because it only ever executes one path at a time using symbolic execution.

- By removing code blocks that are deemed irrelevant, T-FUZZ [92] prevents its mutation-based fuzzer (which does not use symbolic execution) from getting stuck. It then employs symbolic execution to validate the bugs found.

- IFL [93] generates quality input to a smart contract based on a symbolic execution engine and then uses them to train a neural network. This can then be used to fuzz other smart contracts since they often implement similar functionality.

- TaintScope [38] uses taint analysis to bypass checksum checks and then symbolic execution to fix checksum fields in malformed test cases.

5 Conclusions

5.1 Future Work

- Look at author overlap between the survey papers and influential primary papers to guide which review papers seem important (if you independently collect primary papers), if survey paper authors overemphasize their own contributions and maybe even miss other important developments
- History and composition of systems: Which influenced which, which builds on top of/extends which, etc.

5.1.1 Bibliometry

References

[1] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05, Chicago, IL, USA: Association for Computing Machinery, 2005, pp. 213–223, ISBN: 1595930566. DOI: 10.1145/1065010.1065036. [Online]. Available: https://doi.org/10.1145/1065010.1065036.

[2] S. Mallissery and Y.-S. Wu, "Demystify the fuzzing methods: A comprehensive survey," *ACM Comput. Surv.*, vol. 56, no. 3, Oct. 2023, ISSN: 0360-0300. DOI: 10.1145/3623375. [Online]. Available: https://doi.org/10.1145/3623375.

[3] C. Cadar, P. Godefroid, S. Khurshid, *et al.*, "Symbolic execution for software testing in practice: Preliminary assessment," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11, Waikiki, Honolulu, HI, USA: Association for Computing Machinery, May 2011, pp. 1066–1071, ISBN: 9781450304450. DOI: 10.1145/1985793.1985995. [Online]. Available: https://doi.org/10.1145/1985793.1985995.

[4] B. Liu, L. Shi, Z. Cai, and M. Li, "Software vulnerability discovery techniques: A survey," in *2012 Fourth International Conference on Multimedia Information Networking and Security*, 2012, pp. 152–156. DOI: 10.1109/MINES.2012.202.

[5] J. Li, B. Zhao, and C. Zhang, "Fuzzing: A survey," *Cybersecurity*, vol. 1, no. 1, p. 6, Jun. 2018, ISSN: 2523-3246. DOI: 10.1186/s42400-018-0002-y. [Online]. Available: https://doi.org/10.1186/s42400-018-0002-y.

[6] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990, ISSN: 0001-0782. DOI: 10.1145/96267.96279. [Online]. Available: https://doi.org/10.1145/96267.96279.

[7] V. J. Manès, H. Han, C. Han, *et al.*, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, Nov. 2021. DOI: 10.1109/TSE.2019.2946563.

[8] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18, Toronto, Canada: Association for Computing Machinery, 2018, pp. 2123–2138, ISBN: 9781450356930. DOI: 10.1145/3243734.3243804. [Online]. Available: https://doi.org/10.1145/3243734.3243804.

[9] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976, ISSN: 0001-0782. DOI: 10.1145/360248.360252. [Online]. Available: https://doi.org/10.1145/360248.360252.

[10] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, Feb. 2013, ISSN: 0001-0782. DOI: 10.1145/2408776.2408795. [Online]. Available: https://doi.org/10.1145/2408776.2408795.

[11] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08, Tucson, AZ, USA: Association for Computing Machinery, 2008, pp. 206–215, ISBN: 9781595938602. DOI: 10.1145/1375581.1375607. [Online]. Available: https://doi.org/10.1145/1375581.1375607.

[12] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *2010 IEEE Symposium on Security and Privacy*, Jul. 2010, pp. 317–331. DOI: 10.1109/SP.2010.26.

[13] R. McNally, K. K.-H. Yiu, D. A. Grove, and D. Gerhardy, "Fuzzing: The state of the art," *DSTO Defence Science and Technology Organisation*, Feb. 2012.

[14] S. Anand, E. K. Burke, T. Y. Chen, *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, Aug. 2013, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2013.02.061. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121213000563.

[15] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, May 2018, ISSN: 0360-0300. DOI: 10.1145/3182657. [Online]. Available: https://doi.org/10.1145/3182657.

[16] C. Chen, B. Cui, J. Ma, R. Wu, J. Guo, and W. Liu, "A systematic review of fuzzing techniques," *Computers & Security*, vol. 75, pp. 118–137, Jun. 2018, ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2018.02.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404818300658.

[17] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1199–1218, Sep. 2018. DOI: 10.1109/TR.2018.2834476.

[18] P. Godefroid, "Fuzzing: Hack, art, and science," *Commun. ACM*, vol. 63, no. 2, pp. 70–76, Jan. 2020, ISSN: 0001-0782. DOI: 10.1145/3363824. [Online]. Available: https://doi.org/10.1145/3363824.

[19] A. Sabbaghi and M. R. Keyvanpour, "A systematic review of search strategies in dynamic symbolic execution," *Computer Standards & Interfaces*, vol. 72, p. 103 444, Oct. 2020, ISSN: 0920-5489. DOI: https://doi.org/10.1016/j.csi.2020.103444. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0920548919300066.

[20] T. Zhang, Y. Jiang, R. Guo, X. Zheng, and H. Lu, "A survey of hybrid fuzzing based on symbolic execution," in *Proceedings of the 2020 International Conference on Cyberspace Innovation of Advanced Technologies*, ser. CIAT 2020, Guangzhou, China: Association for Computing Machinery, Jan. 2021, pp. 192–196, ISBN: 9781450387828. DOI: 10.1145/3444370.3444570. [Online]. Available: https://doi.org/10.1145/3444370.3444570.

[21] M. Boehme, C. Cadar, and A. ROYCHOUDHURY, "Fuzzing: Challenges and reflections," *IEEE Software*, vol. 38, no. 3, pp. 79–86, May 2021. DOI: 10.1109/MS.2020.3016773.

[22] F. Rustamov, J. Kim, J. Yu, and J. Yun, "Exploratory review of hybrid fuzzing for automated vulnerability detection," *IEEE Access*, vol. 9, pp. 131 166–131 190, Sep. 2021. DOI: 10.1109/ACCESS.2021.3114202.

[23] C. Beaman, M. Redbourne, J. D. Mummery, and S. Hakak, "Fuzzing vulnerability discovery techniques: Survey, challenges and future directions," *Computers & Security*, vol. 120, p. 102 813, Sep. 2022, ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2022.102813. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404822002073.

[24] S. Anand, C. S. Păsăreanu, and W. Visser, "Jpf–se: A symbolic execution extension to java pathfinder," in *Tools and Algorithms for the Construction and Analysis of Systems*, O. Grumberg and M. Huth, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 134–138, ISBN: 978-3-540-71209-1.

[25] K. Havelund and T. Pressburger, "Model checking java programs using java pathfinder," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 366–381, Mar. 2000, ISSN: 1433-2779. DOI: 10.1007/s100090050043. [Online]. Available: https://doi.org/10.1007/s100090050043.

[26] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 263–272, Sep. 2005, ISSN: 0163-5948. DOI: 10.1145/1095430.1081750. [Online]. Available: https://doi.org/10.1145/1095430.1081750.

[27] K. Sen and G. Agha, "Cute and jcute: Concolic unit testing and explicit path model-checking tools," in *Proceedings of the 18th International Conference on Computer Aided Verification*, ser. CAV'06, Seattle, WA: Springer-Verlag, 2006, pp. 419–423, ISBN: 354037406X. DOI: 10.1007/11817963_38. [Online]. Available: https://doi.org/10.1007/11817963_38.

[28] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008, pp. 443–446. DOI: 10.1109/ASE.2008.69.

[29] P. Godefroid, M. Y. Levin, and D. Molnar, "Automated whitebox fuzz testing," Nov. 2008. [Online]. Available: https://www.microsoft.com/en-us/research/publication/automated-whitebox-fuzz-testing/.

[30] N. Tillmann and J. de Halleux, "Pex–white box test generation for .net," in *Tests and Proofs*, B. Beckert and R. Hähnle, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 134–153, ISBN: 978-3-540-79124-9.

[31] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: Automatically generating inputs of death," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 2, Dec. 2008, ISSN: 1094-9224. DOI: 10.1145/1455518.1455522. [Online]. Available: https://doi.org/10.1145/1455518.1455522.

[32] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08, San Diego, California: USENIX Association, 2008, pp. 209–224.

[33] I. Inc. "Spike." (2014), [Online]. Available: http://www.immunitysec.com/resources-freesoftware.shtml (visited on 11/28/2023).

[34] R. Sears. "Sulley." (2019), [Online]. Available: https://github.com/OpenRCE/sulley (visited on 11/28/2023).

[35] J. Pereyda and M. Lindner. "Boofuzz." (2023), [Online]. Available: https://github.com/jtpereyda/boofuzz (visited on 11/28/2023).

[36] M. Eddington. "Peach." (2013), [Online]. Available: https://peachtech.gitlab.io/peach-fuzzer-community/ (visited on 11/28/2023).

[37] A. Takanen, J. D. Demott, C. Miller, and A. Kettunen, *Fuzzing for software security testing and quality assurance*. Artech House, 2018.

[38] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *2010 IEEE Symposium on Security and Privacy*, 2010, pp. 497–512. DOI: 10.1109/SP.2010.37.

[39] N. Stephens, J. Grosen, C. Salls, *et al.*, "Driller: Augmenting fuzzing through selective symbolic execution," in *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS)*, 2016.

[40] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: A survey for roadmap," *ACM Comput. Surv.*, vol. 54, no. 11s, Sep. 2022, ISSN: 0360-0300. DOI: 10.1145/3512345. [Online]. Available: https://doi.org/10.1145/3512345.

[41] X. Zhao, H. Qu, J. Xu, X. Li, W. Lv, and G.-G. Wang, "A systematic review of fuzzing," *Soft Computing*, pp. 1–30, Oct. 2023.

[42] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM : A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD: USENIX Association, Aug. 2018, pp. 745–761, ISBN: 978-1-939133-04-5. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/yun.

[43] Y. Chen, P. Li, J. Xu, *et al.*, "Savior: Towards bug-driven hybrid testing," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1580–1596. DOI: 10.1109/SP40000.2020.00002.

[44] L. Zhao, P. Cao, Y. Duan, H. Yin, and J. Xuan, "Probabilistic path prioritization for hybrid fuzzing," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 3, pp. 1955–1973, 2022. DOI: 10.1109/TDSC.2020.3042259.

[45] H. Huang, P. Yao, R. Wu, Q. Shi, and C. Zhang, "Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1613–1627. DOI: 10.1109/SP40000.2020.00063.

[46] G. Grieco, M. Ceresa, and P. Buiras, "Quickfuzz: An automatic random fuzzer for common file formats," *SIGPLAN Not.*, vol. 51, no. 12, pp. 13–20, Sep. 2016, ISSN: 0362-1340. DOI: 10.1145/3241625.2976017. [Online]. Available: https://doi.org/10.1145/3241625.2976017.

[47] J. Kukucka, L. Pina, P. Ammann, and J. Bell, "Confetti: Amplifying concolic guidance for fuzzers," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22, Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 438–450, ISBN: 9781450392211. DOI: 10.1145/3510003.3510628. [Online]. Available: https://doi.org/10.1145/3510003.3510628.

[48] L. Borzacchiello, E. Coppa, and C. Demetrescu, "Fuzzolic: Mixing fuzzing and concolic execution," *Computers & Security*, vol. 108, p. 102 368, 2021, ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2021.102368. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404821001929.

[49] T. L. Munea, H. Lim, and T. Shon, "Network protocol fuzz testing for information systems and applications: A survey and taxonomy," *Multimedia Tools and Applications*, vol. 75, no. 22, pp. 14 745–14 757, Nov. 2016, ISSN: 1573-7721. DOI: 10.1007/s11042-015-2763-6. [Online]. Available: https://doi.org/10.1007/s11042-015-2763-6.

[50] E. Andreasen, L. Gong, A. Møller, *et al.*, "A survey of dynamic analysis and test generation for javascript," *ACM Comput. Surv.*, vol. 50, no. 5, Sep. 2017, ISSN: 0360-0300. DOI: 10.1145/3106739. [Online]. Available: https://doi.org/10.1145/3106739.

[51] M. Eceiza, J. L. Flores, and M. Iturbe, "Fuzzing the internet of things: A review on the techniques and challenges for efficient vulnerability discovery in embedded systems," *IEEE Internet of Things Journal*, vol. 8, no. 13, pp. 10 390–10 411, 2021. DOI: 10.1109/JIOT.2021.3056179.

[52] C. Zhang, Y. Wang, and L. Wang, "Firmware fuzzing: The state of the art," in *Proceedings of the 12th Asia-Pacific Symposium on Internetware*, ser. Internetware '20, Singapore, Singapore: Association for Computing Machinery, 2021, pp. 110–115, ISBN: 9781450388191. DOI: 10.1145/3457913.3457934. [Online]. Available: https://doi.org/10.1145/3457913.3457934.

[53] Y. Tian, X. Qin, and S. Gan, "Research on fuzzing technology for javascript engines," in *Proceedings of the 5th International Conference on Computer Science and Application Engineering*, ser. CSAE '21, Sanya, China: Association for Computing Machinery, 2021, ISBN: 9781450389853. DOI: 10.1145/3487075.3487107. [Online]. Available: https://doi.org/10.1145/3487075.3487107.

[54] S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur, and H.-N. Lee, "Ethereum smart contract analysis tools: A systematic review," *IEEE Access*, vol. 10, pp. 57 037–57 062, 2022. DOI: 10.1109/ACCESS.2022.3169902.

[55] M. Eisele, M. Maugeri, R. Shriwas, C. Huth, and G. Bella, "Embedded fuzzing: A review of challenges, tools, and solutions," *Cybersecurity*, vol. 5, no. 1, p. 18, Sep. 2022, ISSN: 2523-3246. DOI: 10.1186/s42400-022-00123-y. [Online]. Available: https://doi.org/10.1186/s42400-022-00123-y.

[56] J. Yun, F. Rustamov, J. Kim, and Y. Shin, "Fuzzing of embedded systems: A survey," *ACM Comput. Surv.*, vol. 55, no. 7, Dec. 2022, ISSN: 0360-0300. DOI: 10.1145/3538644. [Online]. Available: https://doi.org/10.1145/3538644.

[57] Z. Zhang, H. Zhang, J. Zhao, and Y. Yin, "A survey on the development of network protocol fuzzing techniques," *Electronics*, vol. 12, no. 13, 2023, ISSN: 2079-9292. DOI: 10.3390/electronics12132904. [Online]. Available: https://www.mdpi.com/2079-9292/12/13/2904.

[58] C. Cadar and D. Engler, "Execution generated test cases: How to make systems code crash itself," in *Model Checking Software*, P. Godefroid, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 2–23, ISBN: 978-3-540-31899-6.

[59] K. Lakhotia, N. Tillmann, M. Harman, and J. de Halleux, "Flopsy - search-based floating point constraint solving for symbolic execution," in *Testing Software and Systems*, A. Petrenko, A. Simão, and J. C. Maldonado, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 142–157, ISBN: 978-3-642-16573-3.

[60] M. Souza, M. Borges, M. d'Amorim, and C. S. Păsăreanu, "Coral: Solving complex constraints for symbolic pathfinder," in *NASA Formal Methods*, M. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 359–374, ISBN: 978-3-642-20398-5.

[61] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "Hfl: Hybrid fuzzing on the linux kernel," in *Network and Distributed System Security Symposium*, 2020.

[62] R. Majumdar and K. Sen, "Hybrid concolic testing," in *29th International Conference on Software Engineering (ICSE'07)*, 2007, pp. 416–426. DOI: 10.1109/ICSE.2007.41.

[63] P. Goodman and A. Dinaburg, "The past, present, and future of cyberdyne," *IEEE Security & Privacy*, vol. 16, no. 2, pp. 61–69, 2018. DOI: 10.1109/MSP.2018.1870859.

[64] G. Yang, S. Person, N. Rungta, and S. Khurshid, "Directed incremental symbolic execution," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 1, Oct. 2014, ISSN: 1049-331X. DOI: 10.1145/2629536. [Online]. Available: https://doi.org/10.1145/2629536.

[65] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen, "Directed test suite augmentation: Techniques and tradeoffs," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '10, Santa Fe, New Mexico, USA: Association for Computing Machinery, 2010, pp. 257–266, ISBN: 9781605587912. DOI: 10.1145/1882291.1882330. [Online]. Available: https://doi.org/10.1145/1882291.1882330.

[66] J. Choi, J. Jang, C. Han, and S. K. Cha, "Grey-box concolic testing on binary code," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 736–747. DOI: 10.1109/ICSE.2019.00082.

[67] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Fitness-guided path exploration in dynamic symbolic execution," in *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, 2009, pp. 359–368. DOI: 10.1109/DSN.2009.5270315.

[68] D. Trabish, A. Mattavelli, N. Rinetzky, and C. Cadar, "Chopped symbolic execution," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18, Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 350–360, ISBN: 9781450356381. DOI: 10.1145/3180155.3180251. [Online]. Available: https://doi.org/10.1145/3180155.3180251.

[69] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," in *Proceedings of the 22nd USENIX Conference on Security*, ser. SEC'13, Washington, D.C.: USENIX Association, 2013, pp. 49–64, ISBN: 9781931971034.

[70] "Undefinedbehaviorsanitizer." (2023), [Online]. Available: `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html` (visited on 11/25/2023).

[71] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner, and R. Ramler, "Grt: Program-analysis-guided random testing (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 212–223. DOI: 10.1109/ASE.2015.49.

[72] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing.," in *NDSS*, vol. 17, 2017, pp. 1–14.

[73] P. Boonstoppel, C. Cadar, and D. Engler, "Rwset: Attacking path explosion in constraint-based test generation," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08, Budapest, Hungary: Springer-Verlag, 2008, pp. 351–366, ISBN: 3540787992.

[74] P. Godefroid, "Compositional dynamic test generation," in *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '07, Nice, France: Association for Computing Machinery, 2007, pp. 47–54, ISBN: 1595935754. DOI: 10.1145/1190216.1190226. [Online]. Available: `https://doi.org/10.1145/1190216.1190226`.

[75] R. Majumdar and K. Sen, "Latest: Lazy dynamic test input generation," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2007-36, Mar. 2007. [Online]. Available: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-36.html`.

[76] P. Collingbourne, C. Cadar, and P. H. Kelly, "Symbolic crosschecking of floating-point and simd code," in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys '11, Salzburg, Austria: Association for Computing Machinery, 2011, pp. 315–328, ISBN: 9781450306348. DOI: 10.1145/1966445.1966475. [Online]. Available: `https://doi.org/10.1145/1966445.1966475`.

[77] S. Khurshid, C. S. PĂsĂreanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *Tools and Algorithms for the Construction and Analysis of Systems*, H. Garavel and J. Hatcliff, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 553–568, ISBN: 978-3-540-36577-8.

[78] J. de Halleux and N. Tillmann, "Moles: Tool-assisted environment isolation with closures," in *Objects, Models, Components, Patterns*, J. Vitek, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 253–270, ISBN: 978-3-642-13953-6.

[79] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340, ISBN: 978-3-540-78800-3.

[80] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *Computer Aided Verification*, W. Damm and H. Hermanns, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 519–531, ISBN: 978-3-540-73368-3.

[81] H. Barbosa, C. W. Barrett, M. Brain, *et al.*, "Cvc5: A versatile and industrial-strength SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, D. Fisman and G. Rosu, Eds., ser. Lecture Notes in Computer Science, vol. 13243, Springer, 2022, pp. 415–442. DOI: 10.1007/978-3-030-99524-9_24. [Online]. Available: `https://doi.org/10.1007/978-3-030-99524-9%5C_24`.

[82] M. Cho, S. Kim, and T. Kwon, "Intriguer: Field-level constraint solving for hybrid fuzzing," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19, London, United Kingdom: Association for Computing Machinery, 2019, pp. 515–530, ISBN: 9781450367479. DOI: 10.1145/3319535.3354249. [Online]. Available: `https://doi.org/10.1145/3319535.3354249`.

[83] D. Liew, C. Cadar, A. F. Donaldson, and J. R. Stinnett, "Just fuzz it: Solving floating-point constraints using coverage-guided fuzzing," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019, Tallinn, Estonia: Association for Computing Machinery, 2019, pp. 521–532, ISBN: 9781450355728. DOI: 10.1145/3338906.3338921. [Online]. Available: `https://doi.org/10.1145/3338906.3338921`.

[84] D. Song, D. Brumley, H. Yin, *et al.*, "BitBlaze: A new approach to computer security via binary analysis," in *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper.*, Hyderabad, India, Dec. 2008.

[85] C. Barrett and S. Berezin, "Cvc lite: A new implementation of the cooperating validity checker," in *Computer Aided Verification*, R. Alur and D. A. Peled, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 515–518, ISBN: 978-3-540-27813-9.

[86] K. Sen, "Scalable automated methods for dynamic program analysis," AAI3242987, Ph.D. dissertation, USA, 2006, ISBN: 9780542990465.

[87] K. Sen and G. Agha, "Automated systematic testing of open distributed programs," in *Fundamental Approaches to Software Engineering*, L. Baresi and R. Heckel, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 339–356, ISBN: 978-3-540-33094-3.

[88] K. Sen and G. Agha, "A race-detection and flipping algorithm for automated testing of multithreaded programs," in *Proceedings of the 2nd International Haifa Verification Conference on Hardware and Software, Verification and Testing*, ser. HVC'06, Haifa, Israel: Springer-Verlag, 2006, pp. 166–182, ISBN: 9783540708889.

[89] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke, "Combining symbolic execution with model checking to verify parallel numerical programs," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, no. 2, May 2008,

ISSN: 1049-331X. DOI: 10.1145/1348250.1348256. [Online]. Available: https://doi.org/10.1145/1348250.1348256.

[90] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++ : Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/woot20/presentation/fioraldi.

[91] F. Wang and Y. Shoshitaishvili, "Angr - the next generation of binary analysis," in *2017 IEEE Cybersecurity Development (SecDev)*, 2017, pp. 8–9. DOI: 10.1109/SecDev.2017.14.

[92] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: Fuzzing by program transformation," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 697–710. DOI: 10.1109/SP.2018.00056.

[93] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19, London, United Kingdom: Association for Computing Machinery, 2019, pp. 531–548, ISBN: 9781450367479. DOI: 10.1145/3319535.3363230. [Online]. Available: https://doi.org/10.1145/3319535.3363230.