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

Valentin Huber

December 1, 2023

Abstract

Contents

1 Introduction

In 2022, the cost of poor software quality was estimated to be more than $2.4 trillion in the US alone. [1] Individual cyber attacks have had an estimated financial impact of up to $8 billion worldwide. [2] Testing software accounts for approximately 50% of development costs [3], but it is typically a mostly manual process [4]. Because manual testing requires many developer hours with in-depth knowledge of the system being tested, it does not scale well. Automated testing promises to be more cost effective in finding software defects and has therefore become the most popular vulnerability discovery solution, especially in the industry. [5]

One such automated vulnerability and bug testing technique is fuzzing. [6] In the seminal work by Miller *et al.* in 1990, the term "fuzz" is defined as a program that "generates a stream of random characters to be consumed by a target program" [7]. Since then, a rich ecosystem of fuzzing systems has developed in both industry and academia, taking design inspiration from a wide variety of software engineering concepts, and combining them into programs that generate various concrete inputs, which are then repeatedly fed into a particular program under test (PUT), and then check the program for illegal states or crashes. [8]

Fuzzing is widely used in industry, with major technology companies and government agencies such as Google, Microsoft, the US Department of Defense, Cisco and Adobe developing proprietary fuzzers and contributing to open source fuzzers. These are then used to great effect, with Google alone finding 20,000 vulnerabilities in Chrome alone using fuzz testing. [2]

Another approach to automated software testing is symbolic execution [9]. Test frameworks based on symbolic execution do not run programs with concrete inputs, but with variables representing all possible values. By tracking how these values are used, systems based on pure symbolic execution can then reason about and even prove certain hypotheses in a PUT. However, because these systems must essentially emulate the entire program, including all possible program states, pure symbolic execution only works on trivial programs, and breaks down on real-world programs because of their size. Naïve implementations further cannot handle non-trivial software that may be multi-threaded or interact with its environment.

Over the past decades, many fuzzers [3], [10]–[54] have employed symbolic execution based techniques, with great success: SAGE [46] "reportedly found a third of the Windows 7 bugs between 2007-2009" [55].

Since the academic research in this area is vast, existing reviews are used to filter the work on the topic at hand to the most important. The following contributions are made on this basis:

- First, the theoretical principles behind fuzzing and symbolic execution are explained in section 2.
- Second, the author explains their reasoning behind using existing survey papers to base this work on in section 3.
- Third, an overview of existing survey papers investigating the state of fuzzing is given in section 4, along with a short summary of each paper.
- Forth, the challenges fuzzing tools face in implementing symbolic execution techniques, and attempts to mitigate each of them, are listed along with examples of works that implement them in section 5.
- Finally, limitations and possible additions to this work are discussed in section 6.

2 Theoretical Principles

To understand the limitations of symbolic execution and innovations introduced in papers, a fundamental understanding of both general fuzzing procedures and symbolic execution is crucial.

2.1 Fuzzing

Miller *et al.* both invented the term and layed the foundation for fuzzing. They observed that, during a stormy night, lightning strikes would introduce interference in their dial-up based communication channel to a UNIX system, which changed the intended inputs and crashed the tools they were using. They then attempted to reproduce this systematically by repeatedly running tools with random inputs containing printable, non-printable and `NULL` bytes. On different UNIX systems, they were able to crash between 25 and 30% of all tested utility programs. [7]

Ever since then, these fuzzing systems have become more sophisticated and different techniques have been integrated. However, certain characteristics remain similar between all fuzzing systems. They output some concrete input(s) and configurations that can then be used to reproduce the observation, allowing confirmation, reproduction, and debugging of the discovered issues. [8] They automatically find well-defined bugs, such as assertion errors, divisions by zero, NULL pointer dereferences, etc. [57] Further, most systems contain similar parts responsible for the

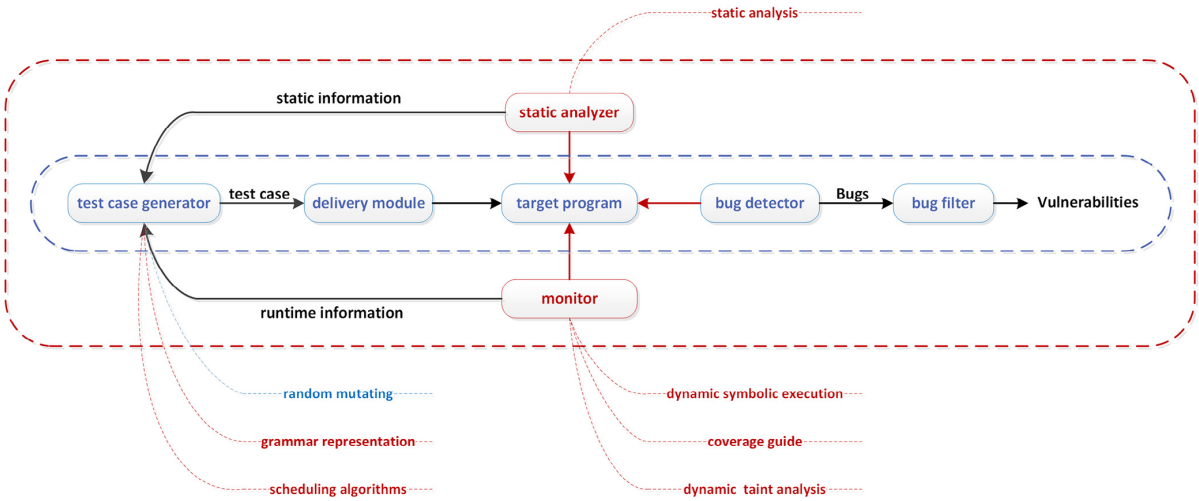

Figure 1: Architectural Diagram of a Fuzzing System [56]

steps during the execution. Figure 1 shows the architecture of a typical fuzzing system.

2.1.1 Target Program

The target program (also known as the program under test, or PUT) is the software to be tested. Different fuzzers have different requirements for the kind of PUT they support: Some require access to source code to add instrumentation during the compilation or because they work on a intermediate representation (IR), such as LLVM bytecode.

2.1.2 Bug Detector

The bug detector observes the PUT during execution for illegal states. The simplest implementations trigger on program crashes, more sophisticated bug detectors might check if certain protected parts of the PUT are accessed, for example to check the authentication implemented.

2.1.3 Bug Filter

The list of inputs that put a PUT into an illegal state may contain duplicates, in a sense that the exploit the same software defect. The bug filter attempts to deduplicate this list based on some heuristics like the order basic blocks were executed in or stack hashes.

2.1.4 Test Case Generator

The test case generator's job is to generate and select the next input to be tested. Based on the test case generator, fuzzing systems can be categorized as either mutation-based or generation-based. Mutation-based fuzzers take some input to the PUT as their input, and then repeatedly mutate those if the results produce "interesting" behavior [8].

The prioritization is either done randomly or based on some heuristic, which assign each possible next input a value used to select the next input to be passed to the delivery module. Heuristics often take into account information produced by the monitor like coverage or distance to target instructions. These approaches can further be combined, e.g. by alternating random and heuristic-based input selection.

2.1.5 Delivery Module

The delivery module is responsible to pass the generated test case to the PUT in the expected form and trigger the actual execution. This might be as simple as starting a command line utility with certain command line arguments, but might also include creating files that are accessed by the PUT or even emulating user interaction.

2.1.6 Monitor

Fuzzing systems can further be classified based on how much access the monitor module has to the PUT. Blackbox fuzzers make a single observation: whether or not the PUT crashed. Graybox fuzzers have limited access to the PUT during execution. Typical information extracted by graybox fuzzers includes which basic blocks were executed in what order, or generally code coverage based on instruction, basic block, or statement. Whitebox fuzzers have full access to the PUT and allow for sophisticated reasoning about the program structure. Systems that employ taint analysis or symbolic execution are categorized as whitebox

fuzzers, since they deeply examine the program structure during their analysis. Different systems make different tradeoffs, accepting higher analysis cost, in the hope of better bug-finding effectiveness. [8]

2.1.7 Static Analyizer

The static analyzer, as its name suggests, attempts to extract information by statically examining the PUT's source code, IR or binary. Such information might include grammars accepted by the PUT or program paths deemed high-risk.

2.2 Symbolic Execution

The concept of symbolic execution in the context of program testing was introduced in 1976 by King. [9] By not executing a PUT with concrete values (such as the number `23` or the string `"Hello World!"`) but instead modelling certain or all values with mathematical variables, it allows to explore all possible paths of a program. Different engines (such as angr [58] or Triton [59]) allow performing symbolic execution on source code, an IR, or a binary of a PUT.

2.2.1 Performing Symbolic Execution

During execution of a certain program, with each instruction, a symbolic execution engine computes and updates the so-called symbolic state. It contains:

- the inputs marked as symbolic as symbols α_i,
- the symbolic expression store σ, which in turn contains
- symbolic expressions ϕ_j, which are either a reference to a symbol α_i, or an arithmetic combination of symbolic expressions, such as $\phi_j = \phi_k - \phi_l$, and finally
- the path constraint π, which is the conjunction of all branch constraints, which are the conditions on symbolic expressions to end up at a certain point in the program, such as $\phi_1 \leq 2$ and $\phi_2 = \phi_3$.

During execution, the symbolic state is changed according to the specific instruction currently executed. If it performs some form of manipulation on existing data, this is represented by adding new symbolic expressions to the symbolic store. If a branch is (not) taken based on a check on variables or registers containing symbolic expressions, the path constraint is amended with an appropriate condition. The exploration of a program can follow different heuristics, such as breath- or depth-first search.

2.2.2 Categoryzing Symbolic Execution Implementations

Symbolic execution implementations can be categorized in several ways: Where static dynamic execution exclusively and exhaustively performs the process described above, dynamic (or concolic, a portmanteau of concrete and symbolic) symbolic execution executes a program symbolically and with concrete values in parallel.

Online symbolic execution allows calculating multiple paths in parallel, while offline symbolic execution examines one path after the other. Finally, one can distinguish between partial and full symbolic execution, where only a part, or all the variable, and therefore calculation is done symbolically. [60]

3 Methods

Fuzzing is an extensively researched topic. For the search term "Fuzzing", Web of Science [61] finds 2,741, Scopus [62] 2,410, and Google Scholar [63] approximately 29,300 works. Even when filtered by top venues, to summarize the current state of symbolic execution in fuzzing would be a task beyond what is feasible in this project.

However, since fuzzing is such a well-published topic, other researchers have taken on the work of summarizing the state of the art, each group with a slightly different focus. These survey papers can therefore be used to approximate a complete picture of the current state of the art. This is the approach chosen by the author of this paper. Section 4 contains a list of survey papers taken into consideration. To ensure accuracy and a consistent level of detail, works discussed in these review papers (primary works) are used to corroborate or refute how review papers discuss the contributions of each primary work.

To select review papers to consider, the following rules were applied:

- First, different search engines were used to create a list of survey papers in the field of fuzzing.
- Then, further review papers discussed or listed in other review papers were added (such as works listed in [2]).
- Furthermore, review papers cited in review papers or primary works were added to list.
- Then, a selection of additional works were added to the list based on the author's intuition.
- Review papers that had a specific focus (such as machine learning in fuzzing [64], [65]) were disregarded.

- Finally, works that focused on a specific use case for fuzzing, such as testing internet of things devices, network protocols, or JavaScript engines were disregarded. A list of these can be found in Appendix 1.

4 Survey Papers

This section summarizes contributions of existing survey papers selected as described in Section 3 and lists relevant primary works discussed in each. Primary works mentioned without discussion are omitted.

4.1 *All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)* (Jul. 2010) [57]

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.

4.2 *Symbolic Execution for Software Testing in Practice: Preliminary Assessment* (May 2011) [4]

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 [66], [67], DART [3], CUTE [13] and jCUTE [68], CREST [12], SAGE [46], Pex [42], EXE [24], and KLEE [35]).

4.3 *Fuzzing: The State of the Art* (Feb. 2012) [55]

McNally *et al.* from Australia's Department of Defence provide an extensive look at fuzzing — *Fuzzing: The State of the Art* is the longest of the discussed survey papers. After discussing the taxonomy, concepts, types, and history of fuzzing, they discuss a list of 15 works from scientific literature and ten commercial and open-source frameworks. In those scientific works, they present four papers that employ symbolic execution, namely KLEE [35], SAGE [46], GWF [69], and TaintScope [53].

4.4 *Symbolic Execution for Software Testing: Three Decades Later* (Feb. 2013) [70]

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 [3], CUTE [13], CREST [12], EXE [24], and KLEE [35].

4.5 *Evaluating Fuzz Testing* (2018) [8]

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.

4.6 *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 broadly discuss the challenges symbolic execution in fuzzing faces. Last, they present TaintScope [53] and Driller [22] as examples of using symbolic execution for specifically for path exploration.

4.7 *The Art, Science, and Engineering of Fuzzing: A Survey* (Nov. 2021) [71]

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" [71], but only discusses whitebox fuzzing in a subsection and refers to other survey papers such as [57], [72] for a more complete overview.

4.8 *Fuzzing: A Survey for Roadmap* (Sep. 2022) [73]

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.

4.9 *Fuzzing vulnerability discovery techniques: Survey, challenges and future directions* (Sep. 2022) [74]

After a short chapter on fuzzer classification, the main focus of this paper are steps and issues along a typical fuzzer workflow, told through the papers that made advances in each category. Finally, it presents current challenges in research and how they could be approached. The discussed papers include some that rely on symbolic execution: Angora [75], T-FUZZ [52], MoWF [38], and HFL [30].

4.10 *A systematic review of fuzzing* (Oct. 2023) [76]

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 [22], QSYM [43], SAVIOR [47], DigFuzz [20], Pangolin [41], and QuickFuzz [44].

4.11 *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 [22], CONFETTI [77], and FUZZOLIC [78]). 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.

4.12 TODO

- An orchestrated survey of methodologies for automated software test case generation (Aug. 2013) [72]
- A Survey of Symbolic Execution Techniques (May 2018) [79]
- A systematic review of fuzzing techniques (Jun. 2018) [56]
- Fuzzing: State of the Art (Sep. 2018) [80]
- Fuzzing: hack, art, and science (Jan. 2020) [81]
- A Systematic Review of Search Strategies in Dynamic Symbolic Execution (Oct. 2020) [82]
- A Survey of Hybrid Fuzzing based on Symbolic Execution (Jan. 2021) [83]
- Fuzzing: Challenges and Reflections (May 2021) [84]
- Exploratory Review of Hybrid Fuzzing for Automated Vulnerability Detection (Sep. 2021) [85]

5 Challenges and Mitigation

Heavily based on [4], [70], extended based on information from all other information considered as listed in Section 4.

This section focuses on attempts to mitigate inherent issues with symbolic execution. Many of the listed papers implemented further more general efficiency improvements, like SAGE's [46] (and multiple other papers inspired by it) generational search, which generates multiple new inputs from just one run of the symbolic execution engine by solving the constraint formula with the constraint from each branch flipped independently.

5.1 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)." [70]
- Two techniques that alleviate this problem:
  - Dynamic Symbolic Execution (DSE), or Concolic Testing (like DART [3] and its successor CUTE [13], and CREST [12]): Run concrete and symbolic execution at the same time, keep mapping between values, solve path constraint with one subconstraint 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" [4]

  - Execution-Generated Testing (EGT) [23] (like EXE [24] and KLEE [35]): 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 [26]) and complex mathematical constraints (like CORAL [11])

5.2 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 [30] 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" [30]. 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" [30].

5.3 Environment Interaction

- "[. . . KLEE's [35]] 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." [4]

5.4 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 [24] and KLEE [35] can however be configured to run in this mode.
- Symbex inherently helps by only looking at possible branches. Example: EXE [24] on `tcpdump`: only 42% of instructions contained symbolic operands, less than 20% of of symbolic branches have both sides feasible [24]
- 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 [24], SAGE [46], CREST [12]
  - "CREST [12] is an extensible platform for building and experimenting with heuristics for selecting which paths to explore" [70]
  - Interleave random and symbolic execution. Examples: Hybrid Concolic Testing [17], [22], [29]
  - Guide towards changes in a patch: Directed Incremental Symbolic Execution [19], Directed Test Suite Augmentation [18] and KATCH [34], which is based on KLEE [35].
  - Chopped symbolic execution ignores (resp. only lazily executes) certain functions deemed uninteresting to focus on certain parts of the PUT and prevent path explosion. This can either be done manually or automatically based on some heuristics (like code unassociated with changes in a patch in Chopper [16]).
  - Prioritize inputs that increase path coverage the most, as implemented in many fuzzers, including SAGE [46]
  - "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)" [25]

  - Weigh the approximate cost of executing a certain path against its demand, as done in QuickFuzz [44]
  - 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. [20]
  - Guide execution towards code parts deemed to be interesting based on static analysis, such as pointer dereferences in loops as implemented in Dowser [21], potential bugs according to UndefinedBehaviorSanitizer [86] in SAVIOR [47] or more general prior static or dynamic program analysis such as in GRT [27] or VUzzer [54] 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." [45]) Eliminating redundant paths by analyzing the values read and written by the program.
- Similarly, MoWF [38] uses knowledge gained by its built-in blackbox fuzzer to prune invalid inputs and thus prevents its symbolic execution engine to get stuck in input checking and error handling code.
- Sharing among states/copy on write: KLEE [35]
- Caching function summaries for later use by higher-level functions. Example: SMART [48]
- Lazy test generation (as in LATEST [37])
- Static path merging (as in KLEE-FP [36])
- partial order and symmetry reductions (as in GSE [28])
- Compact representation of path constraints (as in SAGE [46])

5.5 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 [46].
- Identify independent sub-queries and solve them independently, as is done in EXE [24] and KLEE [35].
- 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 [39]
- 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 [13] and counterexample caching scheme in KLEE [35])
- Cache prior SMT query results and reuse them for future queries. Pangolin [41] uses polyhedral path abstraction to replace query parts for more efficient models based on prior results.
- Improved SMT solvers (like Z3 [87] used in e.g. SAGE [46], STP [50], or cvc5 [14] built during the development of EXE [24])
- Intriguer [32] 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.
- 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 [33] uses this technique to find solutions to floating-point constraints.
- 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 [43]
- While technically not using symbolic execution, analyzing the dependency between input data and state space and approximating solutions to path constraints based on those to prevent expensive calls to an SMT solver is fairly common [88]:

- Eclipser [89] uses instrumentation on the PUT to generate partial path conditions, which can then be solved without invoking SMT solvers to generate further inputs.
- Angora [75] uses solves path constraints by using a combination of context-sensitive branch coverage, scalable byte-level taint tracking, gradient descent searching, input length exploration, and type and shape inference.
- REDQUEEN [90] exploits the fact that much of the input data ends up in the state-space and uses simple transformations on the input data as opposed to relying on taint analysis or symbolic execution to bypass checksums.
- Other proposals include approximate SMT solvers, such as FUZZY-SAT implemented in FUZZOLOGIC [78], or optimistic SMT solvers, as in QSYM [43].

5.6 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." [57]
  - "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." [57]
  - (Potentially) unsound assumptions: optionally rewrite all memory addresses as scalars based on name, like Vine [10]
  - Pass the dealiazing step to the SMT solver like CVC Lite [15] or STP [50].
  - Perform alias analysis. However, like in DART [3], "part of the allure of forward symbolic execution is that it can be done at run-time" [57].
  - EXE [24] and KLEE [35] "perform a mix of alias analyses and letting the SMT solver worry about aliasing" [57]
  - Other systems like DART [3] and CUTE [13] cannot handle non-linear constraints and therefore cannot deal with symbolic references.
- "On the one end of the spectrum is a system like DART [3] that only reasons about concrete pointers, or systems like CUTE [13] and CREST [12] that support only equality and inequality constraints for pointers, which can be efficiently solved. [13] At the other end are systems like EXE [24], and more recently KLEE [35] and SAGE [46] that model pointers using the theory of arrays with selections and updates implemented by solvers like STP or Z3." [70]

5.7 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. [51], [68], [91], [92]" [70]
- "Generalized Symbolic Execution [28] 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)" [4]
- "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." [49] (complex parallel numerical computations)

5.8 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." [4]
- "Pex [42] 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." [4]

5.9 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 [13].
- 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 5.5), this may not solve the issue after all.
- Use static analysis to locate possible jump targets.

5.10 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 [24] and KLEE [35]), or, again, employing concolic execution with all the upsides and drawbacks discussed before.

5.11 Selective Symbolic Execution

Not exhaustive.

- Tools like Driller [22] perform classical fuzzing (in the case of Driller using AFL [93]) 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 [58]) 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 [52] 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 [31] 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 [53] uses taint analysis to bypass checksum checks and then symbolic execution to fix checksum fields in malformed test cases.

5.12 Using Symbolic Execution in Other Contexts

- PYGMALION [40] uses symbolic execution to generate a grammar from a program, generates valid inputs from that, and finally uses those in fuzzers (AFL [93] and KLEE [35]) to measure the achieved code coverage. AUTOGRAM [94] accomplishes a similar goal of producing a context-free grammar a PUT accepts by executing it with different inputs based on taint analysis.

6 Discussion and Future Work

6.1 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.

6.1.1 Bibliometry

Bibliography

References

[1] H. Krasner, "The cost of poor software quality in the us: A 2022 report," Consortium for Information & Software Quality (CISQ), Tech. Rep., Dec. 2022.

[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] 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.

[4] 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.

[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. 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.

[7] 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.

[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] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] P. D. Marinescu and C. Cadar, "Katch: High-coverage testing of software patches," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013, Saint Petersburg, Russia: Association for Computing Machinery, 2013, pp. 235–245, ISBN: 9781450322379. DOI: 10.1145/2491411.2491438. [Online]. Available: https://doi.org/10.1145/2491411.2491438.

[35] 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.

[36] 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.

[37] 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.

[38] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Model-based whitebox fuzzing for program binaries," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '16, Singapore, Singapore: Association for Computing Machinery, 2016, pp. 543–553, ISBN: 9781450338455. DOI: 10.1145/2970276.2970316. [Online]. Available: https://doi.org/10.1145/2970276.2970316.

[39] 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.

[40] R. Gopinath, B. Mathis, M. Höschele, A. Kampmann, and A. Zeller, "Sample-free learning of input grammars for comprehensive software fuzzing," *arXiv preprint arXiv:1810.08289*, 2018.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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/.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

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

[52] 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.

[53] 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.

[54] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing.," in *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS)*, vol. 17, 2017, pp. 1–14.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] F. Saudel and J. Salwan, "Triton: A dynamic symbolic execution framework," in *Symposium sur la sécurité des technologies de l'information et des communications*, ser. SSTIC, Rennes, France, Jun. 2015, pp. 31–54.

[60] S. Flum and V. Huber, "Ghidrion: A ghidra plugin to support symbolic execution," Bachelor's Thesis, Zürich University of Applied Science — Institute of Applied Information Technology, Jun. 2023.

[61] "Web of science." (2023), [Online]. Available: https://www.webofscience.com (visited on 12/01/2023).

[62] "Scopus." (2023), [Online]. Available: https://www.scopus.com (visited on 12/01/2023).

[63] "Google scholar." (2023), [Online]. Available: https://scholar.google.com (visited on 12/01/2023).

[64] G. J. Saavedra, K. N. Rodhouse, D. M. Dunlavy, and P. W. Kegelmeyer, "A review of machine learning applications in fuzzing," *arXiv preprint arXiv:1906.11133*, 2019.

[65] Y. Wang, P. Jia, L. Liu, C. Huang, and Z. Liu, "A systematic review of fuzzing based on machine learning techniques," *PloS one*, vol. 15, no. 8, e0237749, 2020.

[66] 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.

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] 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.

[72] 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.

[73] 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.

[74] 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.

[75] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 711–725. DOI: 10.1109/SP.2018.00046.

[76] 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.

[77] 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.

[78] 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.

[79] 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.

[80] 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.

[81] 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.

[82] 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.

[83] 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.

[84] 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.

[85] 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.

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

[87] 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.

[88] A. Fioraldi, D. C. D'Elia, and E. Coppa, "Weizz: Automatic grey-box fuzzing for structured binary formats," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 1–13, ISBN: 9781450380089. DOI: 10.1145/3395363.3397372. [Online]. Available: https://doi.org/10.1145/3395363.3397372.

[89] 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.

[90] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence.," in *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS)*, vol. 19, 2019, pp. 1–15.

[91] 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.

[92] K. Sen and G. Agha, "A race-detection and flipping algorithm for automated testing of multi-threaded 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.

[93] 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.

[94] M. Hoschele and A. Zeller, "Mining input grammars with autogram," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017, pp. 31–34. DOI: 10.1109/ICSE-C.2017.14.

[95] 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.

[96] 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.

[97] 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.

[98] 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.

[99] 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.

[100] 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.

[101] 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.

[102] 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.

[103] 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.

Appendix

1 List of Survey Papers Focused on a Specific Use Case

- Network protocol fuzz testing for information systems and applications: a survey and taxonomy (Nov. 2016) [95]
- A Survey of Dynamic Analysis and Test Generation for JavaScript (Sep. 2017) [96]
- Fuzzing the Internet of Things: A Review on the Techniques and Challenges for Efficient Vulnerability Discovery in Embedded Systems (2021) [97]
- Firmware Fuzzing: The State of the Art (2021) [98]
- Research on Fuzzing Technology for JavaScript Engines (2021) [99]
- Ethereum Smart Contract Analysis Tools: A Systematic Review (2022) [100]

- Embedded fuzzing: a review of challenges, tools, and solutions (Sep. 2022) [101]
- Fuzzing of Embedded Systems: A Survey (Dec. 2022) [102]
- A Survey on the Development of Network Protocol Fuzzing Techniques (2023) [103]