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

Valentin Huber

Institute of Applied Information Technology
Zürich University of Applied Sciences ZHAW
contact@valentinhuber.me

December 15, 2023

Abstract

Testing is a critical and substantial part of software development. While manual testing is not scalable, automated testing in the form of fuzzing has its limitations. To enhance fuzzing's effectiveness, one approach is to combine it with program analysis techniques such as symbolic execution. However, this also presents its own set of challenges. This work utilizes survey papers on fuzzing to explore this in both academia and industry extensively studied field. After summarizing the focus of each paper and reviewing their discussion of symbolic execution, this work filters, collects, aggregates, and extends their analysis of the inherent challenges that symbolic execution based fuzzing faces, along with the mitigation attempts introduced by a diverse range of systems. The challenges are then categorized, and primary works and their innovations are associated with the appropriate challenge, forming an extensive list of approaches and examples. A total of 78 fuzzing systems employing symbolic execution were identified from 17 survey papers. This paper concludes by discussing the limitations of using survey papers to digest vast research fields and suggesting potential further analyses based on the surveyed material.

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] Software testing often accounts for more than 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 emerged in both industry and academia, taking design inspiration from a variety of software engineering concepts. These are combined into programs that generate various concrete inputs, which are then repeatedly fed into a particular program under test (PUT), and 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 using fuzz testing to find 20,000 vulnerabilities in Chrome alone. [2]

Another approach to automated software testing is symbolic execution [9]. Test frameworks based on symbolic execution run programs not 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 rea-

son 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 also cannot handle non-trivial software that may be multi-threaded or interact with its environment.

Over the past decades, many fuzzers [10]–[87] have employed symbolic execution based techniques, with great success: SAGE [75] "reportedly found a third of the Windows 7 bugs between 2007-2009" [88].

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, Section 3 explains the reasoning behind using existing survey papers to base this work on.
- Third, an overview of existing survey papers investigating the state of fuzzing is given in Section 4, along with a brief summary of each paper.
- Fourth, 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.
- Fifth, possible additions to this work are discussed in Section 6.
- Finally, the contributions of this work are summarized in Section 7.

2 Theoretical Principles

To comprehend the limitations of symbolic execution and the innovations presented in papers, it is crucial to have a fundamental understanding of both general fuzzing procedures and symbolic execution.

2.1 Fuzzing

Miller *et al.* in 1990 both invented the term and laid the foundation for fuzzing. They observed that during a stormy night, lightning strikes would introduce interference into their dial-up communication channel to a UNIX system, altering the intended inputs and crashing the tools they were using. They then attempted to systematically reproduce this by repeatedly running tools with random inputs containing a combination of printable, non-printable, and `NULL` bytes. On various UNIX systems, they were able to crash between 25 and 30% of all utilities tested. [7]

2.1.1 Similarities Across Fuzzers

Since then, 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 fuzzer's 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. [90]

2.1.2 Architecture of a Fuzzer

Figure 1 shows the architecture of a typical fuzzer. Most fuzzers contain similar parts that are responsible for each step during the fuzzing process. These are explained below

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 type of PUT they support: Some require access to the source code to add instrumentation during compilation, or because they work on an intermediate representation (IR), such as LLVM bytecode.

Bug Detector The bug detector watches 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 implemented authentication.

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

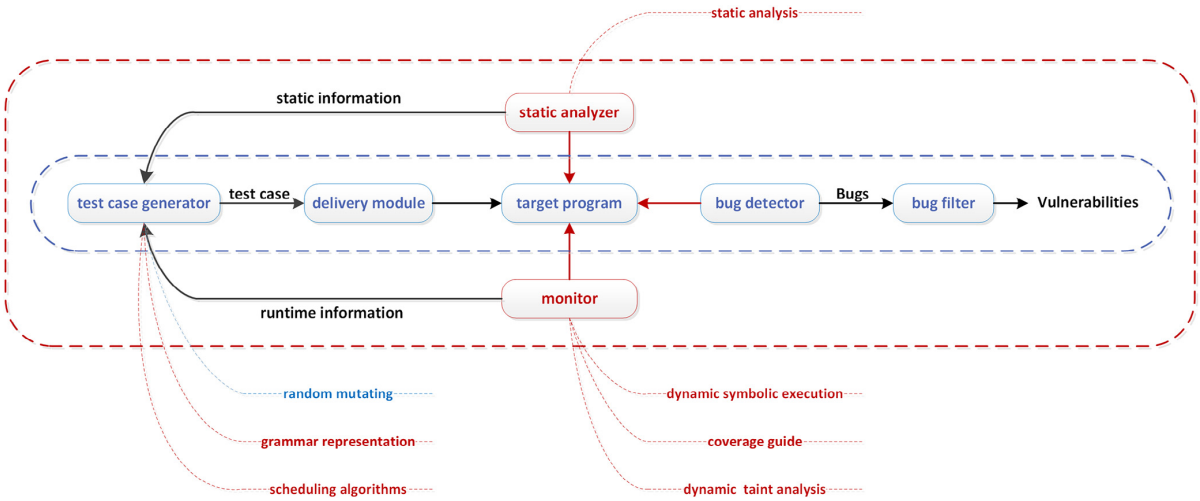

Figure 1: Architectural Diagram of a Fuzzing System [89]

Test Case Generator The task of the test case generator 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 inputs to the PUT as their input, and then repeatedly mutate them if the results produce "interesting" behavior [8].

The prioritization is either random or based on a heuristic that assigns a value to each possible next input, which is used to select the next input to pass to the delivery module. Heuristics often take into account information produced by the monitor, such as coverage or distance to target instructions. These approaches can be further combined, for example, by alternating random and heuristic-based input selection.

Delivery Module The delivery module is responsible for passing the generated test case to the PUT in the expected form and triggering the actual execution. This can be as simple as running a command line utility with specified command line arguments, but can also include creating files to be accessed by the PUT, or even emulating user interaction.

Monitor Fuzzing systems can be further classified based on how much access the monitor module has to the PUT. Blackbox fuzzers make a single observation: whether or not the PUT has 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 sophisticated reasoning about program structure. Systems that use taint analysis or symbolic execution are categorized as whitebox fuzzers because they deeply examine the program structure during their analysis. Different systems make different tradeoffs, accepting higher analysis costs in the hope of better bug-finding effectiveness. [8]

Static Analyzer The static analyzer, as its name suggests, attempts to extract information by statically examining the PUT's source code, IR or binary. Such information may include grammars accepted by the PUT or program paths that are considered 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 [91] or Triton [92]) allow to perform symbolic execution on source code, an IR or a binary of a PUT.

2.2.1 Performing Symbolic Execution

During the execution of a given program, a symbolic execution engine computes and updates the so-called symbolic state with each instruction. 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, i.e. the conditions on symbolic expressions to arrive at a certain point in the program, such as $\phi_1 \leq 2$ or $\phi_2 = \phi_3 \wedge \phi_4 = 3$.

During execution, the symbolic state changes according to the specific instruction being 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 extended with an appropriate condition. The exploration of a program can follow different heuristics, such as breath- or depth-first search. Finally, the path constraint collected along certain paths can be formulated as a query to a Satisfiability Modulo Theory (SMT) solver. This solver can then prove whether the path associated with the constraint is reachable with *any* input, and generate an input that directs the PUT execution in exactly the same way as the analyzed execution.

2.2.2 Categorizing Symbolic Execution Implementations

Symbolic execution implementations can be categorized in several ways: Where static dynamic execution exclusively and exhaustively executes 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 multiple paths to be computed in parallel, while offline symbolic execution examines one path at a time. Finally, one can distinguish between partial and full symbolic execution, where only a part or all of the variables and therefore the computation is done symbolically. [93]

3 Methods

Fuzzing is a extensively researched topic. For the search term "Fuzzing", Web of Science [94] finds 2,741, Scopus [95] finds 2,410, and Google Scholar [96] finds approximately 29,300 works. Even when filtered by the top places, summarizing the current state of symbolic execution in fuzzing from the ground up would be a task beyond the scope of this project.

However, since fuzzing is such a well-published topic, other researchers have taken on the task 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 taken by the author of this paper. Section 4 contains a list of the survey papers considered. To ensure accuracy and a consistent level of detail, the works discussed in these survey papers (primary works) are used to confirm or refute how the survey papers discuss the contributions of each primary work. However, the information in the review papers was generally trusted, and an in-depth analysis of each primary paper mentioned was beyond the scope of this work. Therefore, minor inaccuracies may have gone undetected.

The following rules were applied when selecting review papers for inclusion in this work:

- First, various search engines were used to generate a list of survey papers in the field of fuzzing. Since even the amount of survey papers is overwhelming (Google Scholar [96] returns more than 25,000 results for the search term "fuzzing survey"), works were only included if their title, abstract, or keywords contained the word section "fuzz" and if Google Scholar [96] reports that they were cited more than 5 times or if they were published in a high-impact venue.
- Then, further review papers discussed, listed, or cited in other review papers were added (such as works listed in [2]).
- Works published before January 1, 2010 (such as [97] or [98]) were not included to further limit the scope of this survey.
- Then, review papers focusing on a specific technique (such as machine learning [99], [100]) were excluded.
- Similarly, review papers focusing on hybrid fuzzing [101], [102] were discarded, since most hybrid fuzzers use symbolic execution only in a very limited fashion, which negates most of the obstacles faced by classical symbolic execution based fuzzers. However, to provide some introductory insight, a short survey paper on hybrid fuzzing [103] was selected to be included.
- In addition, works focusing on a specific use case for fuzzing, such as testing Internet of Things

or other embedded devices [104]–[106], network protocols [107], [108], smart contracts [109], or JavaScript engines [110], [111], were excluded.

- Finally, a selection of additional works were added to the list based on the author's intuition.

4 Related Work

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.

***All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)* (Jul. 2010) [90]** 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. Although fuzzing is mentioned in several instances, it is not the main focus. However, many of the drawbacks and advantages of fuzzers based on symbolic execution are still listed, and the additional perspective was valuable in compiling this review.

***Symbolic Execution for Software Testing in Practice: Preliminary Assessment* (May 2011) [4]** After a brief overview of issues faced by symbolic execution based fuzzers, this paper focuses on eight high impact fuzzing tools (JPF-SE and Symbolic (Java) PathFinder [112], [113], DART [30], CUTE [25] and jCUTE [114], CREST [24], SAGE [75], Pex [68], EXE [40], and KLEE [55]).

***Fuzzing: The State of the Art* (Feb. 2012) [88]** McNally *et al.* from the Australian Department of Defence provide a comprehensive look at fuzzing - *Fuzzing: The State of the Art* is the longest of the survey papers discussed. After discussing the taxonomy, concepts, types, and history of fuzzing, the authors discuss a list of 15 papers from the academic literature and ten commercial and open source frameworks. Among these academic papers, they present four papers that employ symbolic execution, namely KLEE [55], SAGE [75], GWF [115], and TaintScope [85].

***Symbolic Execution for Software Testing: Three Decades Later* (Feb. 2013) [116]** As the title suggests, this survey paper 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 them by adapting and extending the algorithms. Finally, the authors present five high-impact tools they have worked on: DART [30], CUTE [25], CREST [24], EXE [40], and KLEE [55].

***An orchestrated survey of methodologies for automated software test case generation* (Aug. 2013) [3]** Orchestrated surveys "consist of a collaborative work collecting self-standing sections, each focusing on a key surveyed topic" [3]. One of the topics discussed in this particular work is symbolic execution. It includes brief introductions to BBRBP [16], AGLT [11], Darwin [33], MATRIX RELOADED [58], and SRA [79] in its introduction. As mitigation strategies for path explosion, it discusses SMART [77], DDCSE [31], PFA [65], among others. For environment interaction, ASSIE [13] and Cinger [27] are presented. Finally, to attempt to solve constraints that are too complex for direct invocation of the SMT solver, MCSS [59], CORAL [21], and its extension introducing AVM [22] are cited.

***A systematic review of fuzzing techniques* (Jun. 2018) [89]** After discussing the structure of fuzzing systems and different targets observed in literature and industry, this paper focuses on innovations of primary works along the structure of fuzzers. The paper then discusses symbolic execution in the context of the sample generator and the monitor (see Section 2.1.2). The primary works discussed in this text include CUTE [25], KLEE [55], SAGE [75], TaintScope [85], Buzz-Fuzz [117], GWF [115], Dowser [37], BORG [17], Driller [38], and MoWF [62]. Additionally, the text contains a substantial section comparing symbolic execution based fuzzing systems and their contributions.

***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. Finally, they present TaintScope [85] and Driller [38] as examples of using symbolic execution for specifically for path exploration.

***Fuzzing: State of the Art* (Sep. 2018) [118]** The first mention of a symbolic execution based fuzzer in this paper is SAGE [75], which the authors use to summarize the limitations of whitebox fuzzing.

The following section discusses works that advance one step in the fuzzing workflow, including SYMFUZZ [80], TaintScope [85], and KATCH [54]. Then, and most clearly relevant to this work, *Fuzzing: State of the Art* contains a discussion of the limitations of taint analysis and dynamic symbolic execution. Works discussed in this section include SMART [77], HOTG [49], Cloud9 [28], APLS [12], SAGE [75], CGF [119], DeepFuzz [34], TCR [83], DiSE [35], and CRAXfuzz [23].

Evaluating Fuzz Testing **(Oct. 2018) [8]** While not a classic survey paper, *Evaluating Fuzz Testing* finds issues in how all 32 papers (not listed here for brevity) performed the evaluation of the system they introduced. It further proposes rules to follow to make an evaluation robust. Finally, it contains a list of what advances each paper examined claims to introduce.

Fuzzing: Hack, Art, and Science **(Jan. 2020) [120]** Godefroid gives an easy-to-understand introduction to fuzzing in this work. He distinguishes between blackbox, grammar-based, whitebox, graybox, and hybrid fuzzing, and provides code examples that show how some of these approaches work. In the section on whitebox fuzzing, SAGE [75] is discussed comparatively extensively, KLEE [55], S2E [74], and SPF [121] are mentioned.

A Survey of Hybrid Fuzzing Based on Symbolic Execution **(Jan. 2021) [103]** This paper focuses on hybrid fuzzing, as the title suggests - the combination of black/greybox and whitebox fuzzing. The discussed works include Hybrid Fuzz Testing [50], Driller [38], QSYM [69], SAVIOR [76], and Pangolin [67].

Fuzzing: Challenges and Reflections **(May 2021) [122]** In comparison to other works, the authors of this article take a less technical and more conceptual approach to surveying the state of the art in fuzzing. They identify areas for improvement, such as usability, residual risk estimation, and fuzzer evaluation techniques, and discuss current approaches and their limitations. To further legitimize their findings, they conducted a survey among security professionals from both academia and industry. The only symbolic execution based fuzzers mentioned are KLEE [55], SAGE [75], and Mayhem [61].

The Art, Science, and Engineering of Fuzzing: A Survey **(Nov. 2021) [123]** Beginning with the proposal of a taxonomy for fuzzing itself and the categorization of fuzzers, this paper proposes a general model of fuzzing, explaining the steps and approaches common to fuzzers. It further presents a genealogy, tracing the origins of important works back to the work of Miller *et al.* However, it "does not provide a comprehensive survey on DSE" [123], but only discusses whitebox fuzzing in a subsection and refers to other survey papers such as [3], [90] for a more complete overview.

Fuzzing: A Survey for Roadmap **(Sep. 2022) [124]** 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 includes a short section on symbolic execution in context input search space handling, but discusses very few papers directly, while often mentioning entire families of papers, only some of which rely on symbolic execution.

Fuzzing vulnerability discovery techniques: Survey, challenges and future directions **(Sep. 2022) [125]** After a short chapter on fuzzer classification, the main focus of this paper is on steps and issues along a typical fuzzer workflow, told through the papers that have made advances in each category. The paper concludes by highlighting current research challenges and suggesting possible approaches. Among the papers discussed are some that rely on symbolic execution: Angora [15], T-FUZZ [84], MoWF [62], and HFL [48].

A systematic review of fuzzing **(Oct. 2023) [126]** The authors of this paper guide the reader through advances in fuzzing along the works that introduced them. It includes a section about symbolic execution, which considers the following systems: Driller [38], QSYM [69], SAVIOR [76], DigFuzz [36], Pangolin [67], and QuickFuzz [70].

Demystify the Fuzzing Methods: A Comprehensive Survey **(Oct. 2023) [2]** This paper dedicates one chapter to explaining the fundamental logic of symbolic execution and then presents three implementations (Driller [38], CONFETTI [127], and FUZZOLIC [42]). It further examines advances in IoT firmware and kernel fuzzers, but does not explain where the advantages and disadvantages of using symbolic execution in these areas lie.

5 Challenges and Mitigation

This section focuses on attempts to mitigate issues inherent in symbolic execution. It categorizes the challenges and presents a (non-exhaustive) list of works that have introduced some improvement to address them. Many of the listed papers have also implemented more general efficiency improvements, such as SAGE's [75] (and several other papers inspired by it) generational search, which generates multiple new inputs from a single run of the symbolic execution engine by solving the constraint formula with the constraint from each branch flipped independently.

5.1 Environment Interactions

Within a symbolic execution environment, one can reason about program behavior, but this obviously breaks down when a program contains actions that interact with the real world and cannot be modeled. Examples of this would be unhandled instructions, system calls, interrupts, inter-process communication such as pipes or sockets, or interaction with external systems in general, since they may return unpredictable results because their logic is opaque to the symbolic execution engine. [2] They may also introduce additional symbolic variables in their return values or have other side effects.

Concolic Execution For this reason, almost all of the systems examined perform some form of concolic execution. This allows them to simply use the concrete values whenever they encounter instructions that cannot be executed symbolically, acting as an exit whenever no other approach solves a problem. By using concrete values, however, symbolic execution systems sacrifice completeness. But for any system that interacts with the environment in non-trivial ways, this is usually the only way to test it. For non-emulatable function calls, parameters that are not influenceable by inputs — that is, they are not symbolic — are simply used directly, while values for symbolic variables are chosen randomly from the set of solutions to the current constraints on them. [4] In addition, there are further advances that certain systems have made, which mean that they do not have to drop down to concrete values, but can evaluate more logic symbolically.

Emulating Function Calls One attempt to deal with system calls, common to many symbolic execution engines, is not to perform system calls or even call external libraries, but to present the symbolic execution engine with code that emulates their effects. This preserves the logical integrity of the resulting constraints, and is often more efficient than the original library, since it can rely on the operator's understanding of the function's effect, rather than on how that logic must be transformed to be executable by a machine. Examples of this technique would be EXE [40] or KLEE [55], and all other systems built on top of them (including PYGMALION [66], KATCH [54], and Cloud9 [28]).

The downside of this approach is obvious: these summaries must be created, maintained, and tested manually, while fuzzing systems generally strive to require as little operator interaction as possible. There are attempts to automatically infer input-output relational logic in the form of path constraints to be added for complex instructions based on their execution with many different values [13]. In addition, there are systems that analyze a PUT and prompt the operator to present models only for the parts of the program that actually introduce imprecision, such as Cinger [27].

Kernel Fuzzing HFL [48] is a kernel fuzzer that relies heavily 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" [48]. To solve these 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" [48].

5.2 Memory Modelling

As described in Section 2.2, symbolic execution engines keep a memory representation of the process in which the PUT is running in memory. However, modelling this memory presents some challenges.

Arithmetic Operations on a physical machine differ from pure mathematical operations on symbolic expressions. For example, integers can over- or underflow. Modeling floating-point arithmetic is even more difficult because the imprecision introduced by rounding must be accurately represented in the resulting constraints to ensure accuracy of the analysis. These problems are largely solved by symbolic execution engines by representing numbers not as mathematical numbers but as bitmaps, as they would be on the machine. The corresponding constraints must then be added at the bitmap level as well, introducing additional complexity. Certain engines further use

special constraint solvers that improve floating-point constraint handling (such as FloPSy [44]) and complex mathematical constraints (such as CORAL [21] and its extension [22]).

When an instruction performs a jump to a symbolic address, this presents a unique challenge to symbolic execution units, since this single operation increases the size of the program's state space. This is a major scaling problem, to the point of making computation infeasible on all but the most trivial programs. To address this challenge, one may be tempted to simply drop down to a concrete value, as described above, with the discussed drawback of lost completeness, but this is the simplest and most inaccurate way. To maintain more than this base level of accuracy, several strategies have been proposed.

One such strategy involves separating different operations on symbolic pointers. Although dereferencing a symbolic pointer can be challenging, comparing pointers may still be feasible. Early adopters of this strategy included CUTE [25] and CREST [24], which only consider (in)equality predicates with symbolic pointers. By performing static analysis and solving the constraints on the symbol in question, the amount of valid values it can take can be drastically reduced, thus reducing the number of states that need to be explored.

EXE [40] and KLEE [55] introduced the concept of regarding symbolic pointers as array accesses. An object accessed with a symbolic pointer is copied as often as necessary to model all possible results, including error states. Or, in other words, "a sound strategy is to consider it a load from any possible satisfying assignment for the expression" [90].

However, if multiple symbolic pointers access the same data type from different parts of the program, the situation becomes more complex. This is because different pointers might point to different or the same values, thus again creating a scaling problem by introducing many permutations. To address this issue, symbolic execution engines, such as DART [30], perform alias analysis at run-time. Alternatively, systems like Vin [18] rewrite all memory addresses as scalars based on their name, which is efficient but relies on a potentially unsound assumption of variable name to value equality. Finally, certain SMT solvers (like STP [128] or Z3 [129]) can handle alias analysis and can be used to delegate some of the work. This is done by tools such as EXE [40], KLEE [55], and SAGE [75].

5.3 Path Explosion

One of the primary issues that symbolic execution faces is what is known as path explosion. This refers to the fact that the number of program paths is usually exponential with the number of static branches in the code. Since naïve symbolic execution attempts to emulate all possible branches, this requires an unattainable amount of memory and compute resources for all but trivial programs. If the examination is based on a simple depth-first search, it gets stuck in non-terminating loops with symbolic conditions and is therefore rarely used. However, both EXE [40] and KLEE [55] can be configured to run in this mode.

Symbolic execution, when implemented in this way, can inherently help with path explosion by only examining possible branches. An example: When running EXE [40] on `tcpdump`, only 42% of the instructions contained symbolic operands, less than 20% of the symbolic branches have both sides feasible. [40]

One simple approach is to set a user-defined timeout for the symbolic execution engine, after which approaches not based on symbolic execution are used. This is called hybrid fuzzing [50]. It is further discussed in Section 5.7.

Otherwise, three main strategies are common: reducing the search space, using advanced data structures and other optimizations to delay path explosion, and guiding the search to limit the effects of path explosion.

5.3.1 Search Space Reduction

Search space reduction can be performed in several ways: First, "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." [72] This takes advantage of the fact that there are often multiple ways to get to the same program state by performing symbolic state analysis at runtime. Further optimizations based on partial order and symmetry reduction were introduced by Khurshid *et al.* [46]

Similarly, MoWF [62] uses knowledge gained by its built-in blackbox fuzzer to prune invalid inputs, thus preventing its symbolic execution engine from getting stuck in input checking and error handling code. CESE [20] uses context-free grammars to restrict its symbolic execution engine to interesting paths, as opposed to error handling during parsing. TCR [83] intelligently reduces existing test cases and prioritizes

the remaining ones according to heuristics to maximize exploration efficiency.

5.3.2 Using Advanced Data Structures

State Representation Since the symbolic state often has only minor differences between closely related paths, it can be represented by a data structure that allows copy-on-write. This means that only the changed parts actually need to be stored, which significantly reduces memory consumption. Many systems use these strategies: KLEE [55], Mayhem [61], S2E [74], BORG [17], and Cloud9 [28]. States can also be statically merged, as in KLEE-FP [56]. To further reduce memory pressure, state information can be transferred to disk, as in Mayhem [61], BORG [17], and SAGE [75]. The latter also introduced a compact representation for path constraints. [75]

Function and Data Structure Summaries To limit the size of paths to be exercised (and thus reduce the amount of symbolic expressions and the size of path constraints), functions can be analyzed and represented by a summary. These summaries can be generated either automatically, as in SMART [77], Higher Order Test Generation [49], or Demand-Driven Compositional Symbolic Execution [31], or manually, as described by Bjørner *et al.* [65]. The latter also introduced a system where common complex structures such as strings and regular expressions can be manually transformed into constraints. [65] The use of function summaries essentially reduces the problem of interprocedural paths to reasoning about intraprocedural paths. This has been demonstrated in LATEST [57].

5.3.3 Guiding the Execution

The approach to mitigating the path explosion problem that has received the most attention is execution guiding. It attempts to direct execution to parts of the PUT that are deemed interesting, in order to complete the analysis (or at least produce findings) as quickly as possible, and before the fuzzer gets stuck and fails to make significant progress.

Since it is unknowable where exactly findings will be produced, search heuristics of varying complexity are used to select the next input to evaluate. These can take any number of inputs from the program monitor (see Section 2.1.2) and trade off increased complexity for better precision.

Dynamic Analysis Most commonly, symbolic execution based fuzzers guide execution by using the input that increases the instruction, basic block, or statement) coverage of the code. Examples include EXE [40], SAGE [75], and CREST [24]. Complementary, Fitnex [43] is a state-dependent fitness function that measures the distance between already discovered feasible paths to a given test target (as defined by any other heuristic).

Other systems reward inputs that lead to longer runtimes, as in Automatic Generation of Load Tests [11], or those that produce very different outputs based on very similar inputs, as in Symbolic Robustness Analysis [79]. QuickFuzz [70] weighs the approximate cost of executing a given path against its demand, and selects the next input based on this ratio.

Static Analysis The second group of attempts revolves around static analysis, usually in the form of examining the control flow graph (CFG) of a PUT. CREST [24] "is an extensible platform for building and experimenting with heuristics for selecting paths to explore" [116] and supports analysis of CFGs. This analysis may mark parts of the PUT as interesting based on various criteria and direct the execution towards them. GRT [45] and VUzzer [86] are examples of systems that use general static analysis. The following are examples of more specific static analysis heuristics:

- Directed Incremental Symbolic Execution [35], Directed Test Suite Augmentation [32], MATRIX RELOADED [58], and KATCH [54] direct execution towards code that has changed in a patch, thus using fuzzing as regression testing.
- Chopper [26] also uses code differences between two versions of a PUT to steer execution, but reverses the incentive structure: It ignores (or lazily executes) certain functions deemed uninteresting in order to focus on certain parts of the PUT and prevent path explosion. While this could be done manually, Chopper [26] uses a heuristic to mark as uninteresting parts of the PUT that are far away from the code changed in the examined patch.
- Darwin [33] uses symbolic execution to generate inputs that are slightly different from crashing inputs, which can then be used to triage and debug a particular software bug.
- CRAXfuzz [23] contains heuristics to identify interesting function calls such as `malloc`.
- Dowser [37] looks for pointer dereferences in loops.

- SAVIOR [76] uses UndefinedBehaviorSanitizer [130] to find potential bugs.
- BORG [17] directs the analysis to buffer overreads.

Alternatively, the approach can be reversed, working backwards from interesting parts of the PUT to external control structures, as in DrillerGo [39].

Probabilistic Approaches Recent work has introduced approaches that use heuristics to assign a probability to each potential next input and select which to execute based on that value. DeepFuzz [34] does this with the goal of finding inputs that penetrate deeper into a PUT. MEUZZ [60] uses machine learning to examine potential next inputs and uses symbolic execution to validate the results. Finally, DigFuzz [36] uses Monte Carlo path optimization to quantify the difficulty of each path using graybox fuzzing, and then lets the whitebox fuzzer focus on the paths that are believed to be the most challenging for graybox fuzzing to make progress.

5.4 Constraint Solving

For most fuzzing systems, constraint solving dominates the runtime. Therefore, it is an area that has received a lot of attention, and several optimizations have been proposed to make query evaluation faster and, in some cases, even possible. They can be categorized as follows

5.4.1 Query Optimization

Query Splitting While solving the entire query may not be feasible, solving parts of it often is. This approach can be further divided into two approaches. The first attempts to identify independent sub-queries and solve them independently (implemented in EXE [40] and KLEE [55], among others). This has the obvious limitation that even the smallest independent but internally interdependent parts of the query may still be too large to solve. The second approach (implemented in Mixed Concrete-Symbolic Solving [59]) sacrifices some of its claim to completeness by solving only parts of the query and using the solution to solve the rest. This approach can produce false negatives, where there is a solution but not with the results selected by the solver based on the first query.

Query Reduction Symbolic execution often generates inefficient queries for the logic that a particular function or program emulates. By using heuristics such as pattern matching, multiple constraints can be reduced to a single constraint. However, while this approach shrinks the query passed to the SMT solver, it introduces a new scaling problem itself - the optimization passes for complex queries may consume an infeasible amount of time or resources, even to the point where solving the query without optimization is more efficient. Therefore, when introducing such heuristics, systems must carefully balance optimization and SMT solver execution time.

Optimization can be done either at symbolic execution runtime or afterwards on the complete query. Examples of the former are "loop-guard patterns matching rules to identify a constraint that defines the number of iterations of input-dependent loops during dynamic symbolic execution, then set new constraints representing the pre- and post-loop conditions to summarize sets of executions of that loop" [89], as implemented in SAGE [75], BORG [17], or APLS [12].

Alternatively, the use of function summaries (as discussed in Section 5.3.2) can also help to reduce the complexity of SMT queries, since the generated logical relationship between input and output is not bound to the code and therefore to the available instructions. In addition, certain functions can be assigned only partial function summaries if they are called with only certain values, similar to mocks and stubs in unit testing. Moles [63] provides a framework for this.

Finally, in certain contexts it may be advantageous not to use Intermediate Representation (IR) for symbolic execution, but to integrate symbolic emulation with native execution through dynamic binary translation, which avoids additional instructions (since it often takes several RISC instructions in IR to replace one CISC instruction) and allows finer-grained control over the constraint, thus making it smaller. This is done in QSYM [69].

5.4.2 Using Information from Previous Queries

Exploiting Query Similarities Since fuzzing systems based on concolic execution generate the next input based on the path that the PUT executed under the previous input by inverting one of the constraints, there is a strong temporal relationship between successive constraint sets and thus SMT queries. A solution to the first query is obviously known in the form of the input that originally generated the query. To solve the second, very similar query, fuzzing systems can thus use parts of the first solution, since if the part of the query they solved did not change, they might also solve the second query.

This is a common approach, with subtle differ-

ences in implementation: SAGE [75] excludes parts of the query solved by previous results from the query passed to the SMT solver. CUTE [25] relies on a similar mechanism and adds an optimization that supersets of queries often do not invalidate existing solutions. An example of this is when additional checks are performed in the PUT on a particular variable that was previously tested to be a valid input to the function in question further down the stack. Finally, KLEE [55] introduced a more versatile counterexample caching scheme for its SMT queries.

Advanced Query Relationship Analysis More recently, systems have introduced complex data structures to store their SMT queries, which then allow solvers to efficiently combine information gathered from previous invocations by exploiting the mathematical similarities between queries. Pangolin [67], for example, uses polyhedral path abstraction to replace query parts for more efficient models based on previous results.

5.4.3 Improved SMT Solvers

While some of the improvements discussed above can be implemented in the fuzzers, a whole class of optimizations must be implemented in the SMT solver. For this reason, improved SMT solvers have been developed alongside many of the more influential symbolic execution based fuzzers. Since they are not part of the fuzzing systems themselves, they have not been analyzed as part of this work. Nevertheless, the following SMT solvers are worth mentioning:

Z3 [129] is one of the most powerful SMT solvers available today. It is developed at Microsoft and is available as open source under a MIT license and therefore widely used, e.g. in SAGE [75] or angr [91] (which is used by Driller [38]). Other SMT solvers used in symbolic execution based fuzzing are STP [128], Yices [131], or cvc5 [132] (which was built during the development of EXE [40]).

5.4.4 Alternatives to SMT Solvers

While the approaches listed above may improve the runtime of SMT solvers, they do not solve the fundamental problem of the exponential complexity they face. A set of fuzzers therefore implement systems that allow them to reason about the relationship between input and program execution path without relying on SMT solvers. It can be argued that these systems should no longer be classified as using symbolic execution, since their solutions are often only approximate, but since their fundamental approach is similar, and they are often listed next to symbolic execution based fuzzers in review papers, they are still discussed here.

Fuzzers that analyse the dependency between input data and the state space, and approximate solutions to path constraints based on this to avoid expensive calls to an SMT solver, are fairly common [87]:

- Intriguer [52] uses taint analysis to detect instructions that access a wide range of input bytes, then performs symbolic execution for those instructions deemed important, invoking the underlying SMT solver only for complicated queries.
- Angora [15] solves path constraints using a combination of context-sensitive branch coverage, scalable byte-level taint tracking, gradient descent search, input length exploration, and type and shape inference.
- Eclipser [41] uses instrumentation on the PUT to generate partial path conditions, which can then be solved without invoking SMT solvers to generate further input.
- SMT formulas can be transformed into programs, which can then be solved using a coverage-guided fuzzer to generate solutions to the initial formula. JFS [53] uses this technique to find solutions to floating-point constraints.
- REDQUEEN [71] exploits the fact that much of the input data ends up in the state space and uses simple transformations on the input data rather than relying on taint analysis or symbolic execution to bypass checksums.
- Other proposals include approximate SMT solvers, such as FUZZY-SAT implemented in FUZZOLOGIC [42], or optimistic SMT solvers, such as in QSYM [69].

5.5 Handling Concurrency

Testing real-life concurrent programs in general is difficult because of their inherent non-determinism. This is also a challenge for symbolic execution based fuzzers. However, some progress has been made in this area: Generalized Symbolic Execution [46] uses model checkers that are able to handle multi-threading (and other forms of non-determinism). In general, "concolic testing was successfully combined with a variant of partial order reduction to test concurrent programs effectively. [64], [73], [81], [114]" [116]

Another development in the area of testing concurrent programs with symbolic execution was introduced in SPIN [78]: It allows checking whether a concurrent program is equivalent to a sequential, less efficient but less error-prone implementation of the same logic. It is used to check complex parallel numerical computations.

5.6 Recursive Data Structures

Similar to concurrency, recursive data structures are difficult to handle in any test framework. In the area of symbolic execution fuzzers, GSE [46] uses lazy initialisation on recursive data structures. Pex [68] supports inputs of primitive types as well as complex (i.e. recursive) data types. To handle the latter, Pex analyses the PUT and generates factory methods for each. These invoke a constructor and a sequence of methods whose parameters are also determined by Pex's analysis.

5.7 Hybrid Fuzzing

Hybrid Fuzzing describes systems that combine multiple approaches to improve the effectiveness of fuzzing. Many of these systems include components that make use of symbolic execution. However, because they use symbolic execution selectively, many of the challenges described above are avoided. As described in Section 3, hybrid fuzzing systems have only been tangentially analysed for this thesis. However, to give the reader an introduction, this section discusses a non-exhaustive list of hybrid fuzzing systems.

Alternating Fuzzing Types Tools such as Driller [38] perform black- or greybox fuzzing (in the case of Driller using AFL [133]) until they are *stuck*, i.e. 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 that was executed with one of the previously generated inputs, and to generate a new input based on the collected path constraint that reaches a section of the PUT that Driller was previously unable to reach. Because Driller does not use symbolic execution as 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.

Other systems that interleave black-/greybox and symbolic execution based fuzzing are Hybrid Concolic Testing [47] and Cyberdyne [29]. DrillerGo [39] uses a similar approach, but specifically targets "unsafe system calls or suspicious locations or functions in the call stack of a reported vulnerability that we wish to reproduce" [39]. Other targeted hybrid fuzzing systems include 1dVul [10] or BugMiner [19].

PUT Manipulation By removing code blocks that are deemed irrelevant, T-FUZZ [84] prevents its mutation-based fuzzer (which does not use symbolic execution) from getting stuck. It then employs symbolic execution to validate the bugs found. TaintScope [85] uses taint analysis to bypass checksum checks, and then uses symbolic execution to fix checksum fields in malformed test cases. Similarly, Stinger [82] analyses a PUT and then uses symbolic execution only on those parts of the PUT that contribute to path constraints. In this way, the overhead introduced by a symbolic execution engine can be limited to parts of the program.

High Quality Test Input Generation IFL [51] generates high quality inputs 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 as they often implement similar functionality.

PYGMALION [66] uses symbolic execution to create a grammar from a program, generates valid inputs from this, and finally uses these in fuzzers (AFL [133] and KLEE [55]) to measure the code coverage achieved. AUTOGRAM [14] achieves a similar goal of generating a context-free grammar that a PUT will accept by running it with different inputs based on taint analysis.

Other Uses of Symbolic Execution SYMFUZZ [80] uses symbolic execution to determine the optimal mutation ratio of a given program-seed pair. When a software vendor's program crashes on a customer's device, BBRBP [16] generates new input that contains no, or at least less, private information. To achieve this, the instructions executed when a program is fed the original crashing input are recorded and then executed using symbolic execution. Finally, the generated path constraint is solved by an SMT solver, which produces a new, unrelated input.

6 Future Work

This work marks only the beginning of what information can be extracted from the combination of review papers and primary works discussed in this work or even in this field in general. What follows is a list of ideas for further analyses.

6.1 Additional Review Papers to Consider

Section 3 mentions different criteria, according to which survey papers were excluded from this work. However, they might still contain important information missed by excluding them. Specifically, certain works appeared to early [134]–[136], or focus on symbolic execution based software testing, but do not primarily concern fuzzing [98], [137]–[140].

6.2 Bibliometry

Survey papers, taken together, might be a good way of measuring the importance of primary papers. By carefully selecting survey papers, looking at their bibliography and counting how often each primary paper appears, one could get a measure of importance for each. To ensure fairness, the scores would have to be weighted according to how many review papers were only written after their publication date.

Compared to examining the bibliographies of primary papers, works that introduced a new technique that was copied or adapted by many subsequent papers would likely be weighted less heavily. This would bias the results towards papers that were highly influential because of their results and ongoing development, not just because of the techniques they introduced.

A time-based analysis might even distinguish between works that are still relevant today, as opposed to those that were superseded soon after publication by other works that implemented the same approach more successfully.

Finally, analysing the context in which a particular citation appears could further improve the accuracy of a bibliometric score of the importance of a primary work. If it is only mentioned, it could be considered less important than if it is discussed in detail. Instead of complex linguistic analysis, counting the number of times a source is cited or examining the section in which it appears could serve as an imperfect but easier to implement proxy.

6.3 Author Analysis

While reading the survey papers discussed in Section 4, one thing that became apparent was that some review papers were (co-)written by authors of important primary works. One example is Cristian Cadar, who is the author of EXE [40], KLEE [55], KLEE-FP [56] RWset [72], KATCH [54], EGT [141], Chopped [26], and JFS [53] and co-authored the review papers [4], [116], [122].

This is to be expected, as these authors already know the material very well. But it may also mean that their own work is over-represented in the review papers they have helped to compile. Investigating this relationship may provide an insight into how easy or difficult it is to gain recognition as an author new to the field. It may also serve as a measure of how likely it is that innovative work by new authors is overlooked by reviewers.

6.4 Genealogy

Since most works discussed in this paper introduce improvements in a specific part of a fuzzing system, they often build on an existing system to provide the foundation. For example, PYGMALION [66], KATCH [54], and Cloud9 [28] build on KLEE [55], APLS [12] extends SAGE [75], and CRAXfuzz [23] extends S2E [74], which in turn extends KLEE [55].

Building a genealogy tree from the primary works discussed might provide a list of highly adaptable and stable projects, which in turn might be more suitable for deployment in commercial applications. *The Art, Science, and Engineering of Fuzzing: A Survey* [123] contains a diagram of such a genealogy. However, as this work does not focus on symbolic execution based fuzzing, it does not distinguish nuances within this subfield and only includes a limited number of projects, while *Exploratory Review of Hybrid Fuzzing for Automated Vulnerability Detection* [101] focuses on hybrid fuzzing systems only.

In addition, if not extending the code itself, ideas or concepts introduced by one paper may be copied or adapted by subsequent work. Building a genealogical dataset across these may reveal hidden relationships between projects and show the importance of ideas implemented in fuzzers across the board.

6.5 Meta Survey

Finally, one could compare review papers based on different categories: primary works discussed (see above), categorisations of fuzzers, or more specifically, categorisations of problems that symbolic execution faces in fuzzing and the solutions to them that different works propose.

7 Conclusion

This survey provides an introduction to fuzzing, in particular the use of symbolic execution in fuzzing systems, its challenges and innovations. It presents a discussion of the theoretical underpinnings of both

fuzzing in general and symbolic execution in particular, and explains how these two approaches to software testing have been combined to create state-of-the-art systems.

As this area has received considerable attention from both academia and industry, fuzzing survey papers are used to collect significant works, including summaries and discussions of them. First, the focus of each of the 17 selected survey papers is summarised, along with a list of 78 discussed symbolic execution based fuzzers. The latter are then combined, extended and categorised. The resulting list shows that a diverse set of attempts has been made to mitigate each of the inherent challenges of symbolic execution, thus making it applicable to real-world programs.

Finally, this work reflects on its own approach of using existing survey papers to gather information about a particular area. Both the limitations of this process and possible additions to this work are discussed. By using this approach, an otherwise unfeasibly large area of research has been explored, providing a comprehensive overview of the state of the art in symbolic execution based fuzzing.

Bibliography

[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] 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.

[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, Oct. 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] J. Peng, F. Li, B. Liu, *et al.*, "1dvul: Discovering 1-day vulnerabilities through binary patches," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2019, pp. 605–616. DOI: 10.1109/DSN.2019.00066.

[11] P. Zhang, S. Elbaum, and M. B. Dwyer, "Automatic generation of load tests," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, 2011, pp. 43–52. DOI: 10.1109/ASE.2011.6100093.

[12] P. Godefroid and D. Luchaup, "Automatic partial loop summarization in dynamic test generation," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11, Toronto, Ontario, Canada: Association for Computing Machinery, 2011, pp. 23–33, ISBN: 9781450305624. DOI: 10.1145/2001420.2001424. [Online]. Available: https://doi.org/10.1145/2001420.2001424.

[13] P. Godefroid and A. Taly, "Automated synthesis of symbolic instruction encodings from i/o samples," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12, Beijing, China: Association for Computing Machinery, 2012, pp. 441–452, ISBN: 9781450312059. DOI: 10.1145/2254064.2254116. [Online]. Available: https://doi.org/10.1145/2254064.2254116.

[14] 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.

[15] 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.

[16] M. Castro, M. Costa, and J.-P. Martin, "Better bug reporting with better privacy," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIII, Seattle, WA, USA: Association for Computing Machinery, 2008, pp. 319–328, ISBN: 9781595939586. DOI: 10.1145/1346281.1346322. [Online]. Available: https://doi.org/10.1145/1346281.1346322.

[17] M. Neugschwandtner, P. Milani Comparetti, I. Haller, and H. Bos, "The borg: Nanoprobing binaries for buffer overreads," in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '15, San Antonio, Texas, USA: Association for Computing Machinery, 2015, pp. 87–97, ISBN: 9781450331913. DOI: 10.1145/2699026.2699098. [Online]. Available: https://doi.org/10.1145/2699026.2699098.

[18] 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.

[19] F. Rustamov, J. Kim, J. Yu, H. Kim, and J. Yun, "Bugminer: Mining the hard-to-reach software vulnerabilities through the target-oriented hybrid fuzzer," *Electronics*, vol. 10, no. 1, 2021, ISSN: 2079-9292. DOI: 10.3390/electronics10010062. [Online]. Available: https://www.mdpi.com/2079-9292/10/1/62.

[20] R. Majumdar and R.-G. Xu, "Directed test generation using symbolic grammars," in *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07, Atlanta, Georgia, USA: Association for Computing Machinery, 2007, pp. 134–143, ISBN: 9781595938824. DOI: 10.1145/1321631.1321653. [Online]. Available: https://doi.org/10.1145/1321631.1321653.

[21] 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.

[22] M. Borges, M. d'Amorim, S. Anand, D. Bushnell, and C. S. Pasareanu, "Symbolic execution with interval solving and meta-heuristic search," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, 2012, pp. 111–120. DOI: 10.1109/ICST.2012.91.

[23] C.-C. Yeh, H. Chung, and S.-K. Huang, "Craxfuzz: Target-aware symbolic fuzz testing," in *2015 IEEE 39th Annual Computer Software and Applications Conference*, vol. 2, 2015, pp. 460–471. DOI: 10.1109/COMPSAC.2015.99.

[24] 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.

[25] 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.

[26] 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.

[27] S. Anand and M. J. Harrold, "Heap cloning: Enabling dynamic symbolic execution of java programs," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, 2011, pp. 33–42. DOI: 10.1109/ASE.2011.6100071.

[28] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," *SIGPLAN Not.*, vol. 47, no. 6, pp. 193–204, Jun. 2012, ISSN: 0362-1340. DOI: 10.1145/2345156.2254088. [Online]. Available: https://doi.org/10.1145/2345156.2254088.

[29] 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.

[30] 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.

[31] S. Anand, P. Godefroid, and N. Tillmann, "Demand-driven compositional symbolic execution," 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. 367–381, ISBN: 978-3-540-78800-3.

[32] 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.

[33] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani, "Darwin: An approach for debugging evolving programs," in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09, Amsterdam, The Netherlands: Association for Computing Machinery, 2009, pp. 33–42, ISBN: 9781605580012. DOI: 10.1145/1595696.1595704. [Online]. Available: https://doi.org/10.1145/1595696.1595704.

[34] K. Böttinger and C. Eckert, "Deepfuzz: Triggering vulnerabilities deeply hidden in binaries," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, J. Caballero, U. Zurutuza, and R. J. Rodríguez, Eds., Cham: Springer International Publishing, 2016, pp. 25–34, ISBN: 978-3-319-40667-1.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] J. Kim and J. Yun, "Poster: Directed hybrid fuzzing on binary code," 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. 2637–2639, ISBN: 9781450367479. DOI: 10.1145/3319535.3363275. [Online]. Available: https://doi.org/10.1145/3319535.3363275.

[40] 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.

[41] 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.

[42] L. Borzacchiello, E. Coppa, and C. Demetrescu, "Fuzzolic: Mixing fuzzing and concolic execution," *Computers & Security*, vol. 108, p. 102368, 2021, ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2021.102368. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404821001929.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] P. Godefroid, "Higher-order test generation," *SIGPLAN Not.*, vol. 46, no. 6, pp. 258–269, Jun. 2011, ISSN: 0362-1340. DOI: 10.1145/1993316.1993529. [Online]. Available: https://doi.org/10.1145/1993316.1993529.

[50] B. S. Pak, "Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution," M.S. thesis, School of Computer Science Carnegie Mellon University, May 2012.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold, "Test-suite augmentation for evolving software," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008, pp. 218–227. DOI: 10.1109/ASE.2008.32.

[59] C. S. Păsăreanu, N. Rungta, and W. Visser, "Symbolic execution with mixed concrete-symbolic solving," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11, Toronto, Ontario, Canada: Association for Computing Machinery, 2011, pp. 34–44, ISBN: 9781450305624. DOI: 10.1145/2001420.2001425. [Online]. Available: https://doi.org/10.1145/2001420.2001425.

[60] Y. Chen, M. Ahmadi, R. M. farkhani, B. Wang, and L. Lu, "MEUZZ: Smart seed scheduling for hybrid fuzzing," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, San Sebastian: USENIX Association, Oct. 2020, pp. 77–92, ISBN: 978-1-939133-18-2. [Online]. Available: https://www.usenix.org/conference/raid2020/presentation/chen.

[61] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *2012 IEEE Symposium on Security and Privacy*, 2012, pp. 380–394. DOI: 10.1109/SP.2012.31.

[62] 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.

[63] 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.

[64] 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.

[65] N. Bjørner, N. Tillmann, and A. Voronkov, "Path feasibility analysis for string-manipulating programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, S. Kowalewski and A. Philippou, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 307–321, ISBN: 978-3-642-00768-2.

[66] 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.

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] 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.

[72] 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.

[73] 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.

[74] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: A platform for in-vivo multi-path analysis of software systems," *SIGPLAN Not.*, vol. 46, no. 3, pp. 265–278, Mar. 2011, ISSN: 0362-1340. DOI: 10.1145/1961296.1950396. [Online]. Available: https://doi.org/10.1145/1961296.1950396.

[75] 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/.

[76] 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.

[77] 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.

[78] 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.

[79] R. Majumdar and I. Saha, "Symbolic robustness analysis," in *2009 30th IEEE Real-Time Systems Symposium*, 2009, pp. 355–363. DOI: 10.1109/RTSS.2009.17.

[80] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 725–741. DOI: 10.1109/SP.2015.50.

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

[82] S. Anand, A. Orso, and M. J. Harrold, "Type-dependence analysis and program transformation for symbolic execution," in *Tools and Algorithms for the Construction and Analysis of Systems*, O. Grumberg and M. Huth, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 117–133, ISBN: 978-3-540-71209-1.

[83] C. Zhang, A. Groce, and M. A. Alipour, "Using test case reduction and prioritization to improve symbolic execution," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014, San Jose, CA, USA: Association for Computing Machinery, 2014, pp. 160–170, ISBN: 9781450326452. DOI: 10.1145/2610384.2610392. [Online]. Available: https://doi.org/10.1145/2610384.2610392.

[84] 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.

[85] 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.

[86] 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.

[87] 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.

[88] 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.

[89] 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.

[90] 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.

[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] 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.

[93] 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.

[94] "Web of science." (2023), [Online]. Available: https://www.webofscience.com (visited on Dec. 1, 2023).

[95] "Scopus." (2023), [Online]. Available: https://www.scopus.com (visited on Dec. 1, 2023).

[96] "Google scholar." (2023), [Online]. Available: https://scholar.google.com (visited on Dec. 1, 2023).

[97] P. Oehlert, "Violating assumptions with fuzzing," *IEEE Security & Privacy*, vol. 3, no. 2, pp. 58–62, 2005. DOI: 10.1109/MSP.2005.55.

[98] C. S. Păsăreanu and W. Visser, "A survey of new trends in symbolic execution for software testing and analysis," *International Journal on Software Tools for Technology Transfer*, vol. 11, no. 4, pp. 339–353, Oct. 2009, ISSN: 1433-2787. DOI: 10.1007/s10009-009-0118-1. [Online]. Available: https://doi.org/10.1007/s10009-009-0118-1.

[99] 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.

[100] 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.

[101] 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.

[102] S. Ognawala, A. Petrovska, and K. Beckers, *An exploratory survey of hybrid testing techniques involving symbolic execution and fuzzing*, Dec. 2017. arXiv: 1712.06843 [cs.SE].

[103] 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.

[104] 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, Jul. 2021. DOI: 10.1109/JIOT.2021.3056179.

[105] 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.

[106] 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.

[107] 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.

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

[109] 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, Apr. 2022. DOI: 10.1109/ACCESS.2022.3169902.

[110] 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, Dec. 2021, ISBN: 9781450389853. DOI: 10.1145/3487075.3487107. [Online]. Available: https://doi.org/10.1145/3487075.3487107.

[111] 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.

[112] 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.

[113] 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.

[114] 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.

[115] 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.

[116] 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.

[117] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 474–484. DOI: 10.1109/ICSE.2009.5070546.

[118] 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.

[119] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, Vienna, Austria: Association for Computing Machinery, 2016, pp. 1032–1043, ISBN: 9781450341394. DOI: 10.1145/2976749.2978428. [Online]. Available: https://doi.org/10.1145/2976749.2978428.

[120] 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.

[121] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta, "Symbolic pathfinder: Integrating symbolic execution with model checking for java bytecode analysis," *Automated Software Engineering*, vol. 20, no. 3, pp. 391–425, Sep. 2013, ISSN: 1573-7535. DOI: 10.1007/s10515-013-0122-2. [Online]. Available: https://doi.org/10.1007/s10515-013-0122-2.

[122] 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.

[123] 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.

[124] 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.

[125] 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.

[126] 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.

[127] 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.

[128] 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.

[129] 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.

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

[131] B. Dutertre and L. de Moura, "A fast linear-arithmetic solver for dpll(t)," in *Computer Aided Verification*, T. Ball and R. B. Jones, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 81–94, ISBN: 978-3-540-37411-4.

[132] 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.

[133] 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.

[134] J. Edvardsson, "A survey on automatic test data generation," in *Proceedings of the 2nd Conference on Computer Science and Engineering*, 1999, pp. 21–28.

[135] M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[136] I. van Sprundel, "Fuzzing: Breaking software in an automated fashion," in *Proceedings of the 22nd Chaos Communication Congress*, 2005.

[137] T. Chen, X.-s. Zhang, S.-z. Guo, H.-y. Li, and Y. Wu, "State of the art: Dynamic symbolic execution for automated test generation," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1758–1773, Sep. 2013, Including Special sections: Cyber-enabled Distributed Computing for Ubiquitous Cloud and Network Services & Cloud Computing and Scientific Applications — Big Data, Scalable Analytics, and Beyond, ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2012.02.006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X12000398.

[138] 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.

[139] 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.

[140] L. Y. Araki and L. M. Peres, "A systematic review of concolic testing with aplication of test criteria.," *ICEIS (2)*, pp. 121–132, 2018.

[141] 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.