Demystify the Fuzzing Methods: A Comprehensive Survey

SANOOP MALLISSERY and YU-SUNG WU, Department of Computer Science, National Yang Ming Chiao Tung University (NYCU), Taiwan

Massive software applications possess complex data structures or parse complex data structures; in such cases, vulnerabilities in the software become inevitable. The vulnerabilities are the source of cyber-security threats, and discovering this before the software deployment is challenging. Fuzzing is a vulnerability discovery solution that resonates with random-mutation, feedback-driven, coverage-guided, constraint-guided, seed-scheduling, and target-oriented strategies. Each technique is wrapped beneath the black-, white-, and grey-box fuzzers to uncover diverse vulnerabilities. It consists of methods such as identifying structural information about the test cases to detect security vulnerabilities, symbolic and concrete program states to explore the unexplored locations, and full semantics of code coverage to create new test cases. We methodically examine each kind of fuzzers and contemporary fuzzers with a profound observation that addresses various research questions and systematically reviews and analyze the gaps and their solutions. Our survey comprised the recent related works on fuzzing techniques to demystify the fuzzing methods concerning the application domains and the target that, in turn, achieves higher code coverage and sound vulnerability detection.

CCS Concepts: • **Security and privacy** → **Software security engineering**; **Software reverse engineering**; *Distributed systems security*; *Information flow control*; • **Software and its engineering** → *Distributed programming languages*; *Parsers;*

Additional Key Words and Phrases: Automated testing, fuzzing, code inspection, vulnerability discovery

ACM Reference format:
Sanoop Mallissery and Yu-Sung Wu. 2023. Demystify the Fuzzing Methods: A Comprehensive Survey. *ACM Comput. Surv.* 56, 3, Article 71 (October 2023), 38 pages.
https://doi.org/10.1145/3623375

1 INTRODUCTION

Software vulnerabilities or bugs are always critical and have widespread attention that attackers can exploit. It can become the root cause of the company's financial loss and reputation damage—for example, the WannaCry ransomware attack [144]. The attack startled the global economy by hitting its impact on around 230K–300K computers in about 150 countries, leading to an estimated substantial financial impact of US $4–$8 billion worldwide [96]. The attack happened because the users had not updated their system with the security patch released for the Microsoft Windows

This study is supported by the National Science and Technology Council of the Republic of China under grant number 111-2628-E-A49-007-MY2 and 112-2634-F-A49-001-MBK.
Authors' address: S. Mallissery and Y.-S. Wu (Corresponding author), Department of Computer Science, National Yang Ming Chiao Tung University (NYCU), No. 1001, Daxue Rd. East Dist., Hsinchu City 30010, Taiwan; e-mails: sanoopmallissery@gmail.com, ysw@nycu.edu.tw.

0360-0300/2023/10-ART71 $15.00
https://doi.org/10.1145/3623375

operating system. This patch removed the vulnerability that EternalBlue [6] exploited to infect computers with WannaCry ransomware.

Another example is the Heartbleed vulnerability in OpenSSL [157]. Seventeen percent of SSL servers worldwide, around 500K servers, were vulnerable to Heartbleed vulnerability [157]. The current version of OpenSSL does not have Heartbleed exposure. However, it still exists in many servers and systems, since some could not upgrade the patched version of OpenSSL. The root cause of any such attacks is vulnerable code designs. Therefore, it is necessary to use some techniques to discover vulnerabilities well before releasing the software. Fuzzing is one of the most promising practices that discover security vulnerabilities by repeatedly testing the software with altered or fuzzed inputs [104].

Most topline companies and organizations utilize fuzzing to ensure quality control and cybersecurity operations. For example, Google uses fuzzing to verify and ensure that the millions of **Lines of Code (LOC)** in Google Chrome are bug-free [67]. It was challenging to admit that Google could find 20K vulnerabilities in Chrome using fuzz testing [67]. The dominant software from Microsoft has to pass the fuzz test stage in the software development cycle to ensure no code vulnerabilities and confirm stability [136]. The DoD Enterprise DevSecOps Reference Design document [40] from the United States has mentioned that continuous testing across the software development cycle is necessary for the test tools support. Therefore, it is essential to use fuzzing to discover Distributed Denial of Service attacks and malware exploit possibilities, validate system security, and reduce the risk of system degradation [40]. Cisco concentrates on searching and identifying vulnerabilities in network applications. They use a network fuzzer that performs replaying the network traffic, and Cisco uses the support of a mutational fuzzer [84]. Adobe Reader from Adobe surpassed the bugs or vulnerabilities in every version release; hence, Yoav Alon et al. [176] have tried using WinAFL. WinAFL, a fork of **American Fuzzy Lop (AFL)** [122] for Windows, was a game-changer for Adobe to discover effective file format bugs, especially in compressed binary formats—images, videos, and archives. Google and Adobe had announced the importance of continuous fuzzing integration in the software development cycle and had released new open source security tools [47].

Google has developed the ClusterFuzz-based continuous fuzzing solution known as the ClusterFuzzLite [68]. It runs as part of Continuous Integration workflows to help users uncover vulnerabilities before committing to the source code. Adobe has released the **Living-off-the-land (LotL)** fuzzer to detect living-off-the-land attacks where malicious code leverage legitimate software to avoid detection [1]. ClusterFuzzLite offered the same feature as ClusterFuzz [33], such as test corpus management, sanitization, integrated and continuous fuzzing, and adequate coverage reports regarding the crash and hangs. Apart from that, ClusterFuzzLite provides ease of use for the end-users. Clusterfuzz identifies more than 6.5K vulnerabilities with 21K bugs from more than 500 open source projects, and that was an eye-opener for most of the open source development team. Despite that, the LotL classifier from Adobe's LotL project leverages **Machine Learning (ML)** abilities such as feature extraction from various prospects to generate a series of labels based on patterns, binary paths, network paths, keywords, and similarities from the logs.

1.1 The Fundamentals of Fuzzing

Fuzzing or fuzz testing is a software testing technique; alternatively, a fuzzer is a type of software that tests a target software [104]. A fuzzing engine works with a few periodic steps (Figure 1).

① System readiness is the primary step toward fuzzing (supplying initial seeds, setting iterations, memory allocation, etc.).
② Automate the fuzzing process and instrument wherever required.
③ Generated test cases will take to the target software for performing an execution using the delivered test case.
④ Continue with new program states or check for potential crashes in the software while following the execution path.
⑤ Report all the fuzz progress (bugs/crashes/hangs/paths/edges) to the fuzz engine and the user.
⑥ User involvement/backtracking is optional to improve fuzzing and explore all the unvisited program control flow paths.

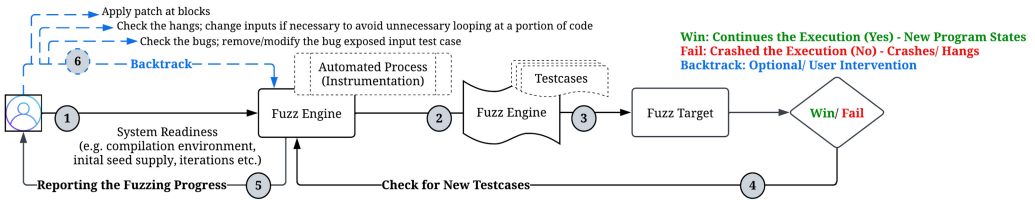

Fig. 1. Steps to automated vulnerability discovery and optional user involvement for better fuzzing progress.

All the steps (①–⑥) are essential in harnessing fuzzing software to become a successful bug/crash detection software. The primary purpose of fuzzing techniques is not only to test the software functionalities but also to explore and uncover software source code bugs and vulnerabilities such as coding errors, buffer overflow vulnerabilities, the likelihood of **Denial of Service (DoS)**, and SIG errors (segmentation faults while accessing a memory location, bugs due to dereferencing a null pointer, program termination with signal SIGSEGV, etc.) in the code by using unpredictable, distorted, random data called as fuzz as program inputs. Therefore, fuzzing tries to crash the software to expect the unexpected behavior of the software for various combinations and blends of inputs.

Upon selecting a fuzzing target, input data can be generated via dump or smart fuzzing [117]. Dump data fuzzing involves generating, creating, or selecting random data to serve as input for the target. However, in the case of smart fuzzing, existing valid input data or the specific input data type accepted by the target is employed. Subsequently, these data can undergo mutation to yield new inputs. As a result, dump fuzzing involves analysis with a collection of malformed input data without an exhaustive understanding. In contrast, smart fuzzing is conducted with a comprehension of the underlying data structure. It is better to utilize smart fuzzing to get an immediate result on a potential security vulnerability in the target.

The selection of the input set is irrespective of whether the target is a source code, a binary, a kernel, a firmware, or an instrumented one. This selection of valid input sets is based on observing the working of target software or by examining its source code. Thus, we can divide this creation of a valid input set into generating a valid mutation guidance model and use this model to produce the well-formed fuzzed data. The random generation of inputs is time-consuming, especially for the targets that accept complex inputs, and there is no surety that it can find a potential vulnerability. For example, a target program that accepts a network packet with a typical **Type of Service (TOS)** field. In such a case, generating a packet with precisely the TOS field the target program expects may take time. In the worst case, it may not generate such a TOS field.

Fuzzers are typically categorized into generation-, mutation-, or evolution-based depending on the target-fuzzing methods.

(1) *Generation fuzzers* use random input data to a specific format the target expects. We take valid input data and do some transformations by breaking those data into bits and bytes and then fuzzing each of these bits and bytes randomly. The input data structure is maintained but only to fuzz the selected parts. Therefore, it follows a proper specification or format of the input [137]. The input test cases get generated with the knowledge of the file formats or network protocols and pass the validation of programs without much difficulty. However, generation fuzzers require support to generate realistic inputs that match the complexity of the target application to fuzz. It will lead to a combinatorial explosion, making it challenging to achieve complete code coverage and limiting its effectiveness in testing sophisticated software systems.
(2) In *mutation fuzzers*, it mutates the input data or seeds without understanding the format or structure of the data. Bit/byte flipping or appending random bits to the input can be mutation fuzzers. The mutation model ensures that the input data structure supplied meets the target's expectations. For example, mutation Fuzzing on file formats or network protocols saves the sampled inputs, then replays after mutation. Though it sounds skeptical, Man in the Middle or Proxy in mutation fuzzing where the fuzzer is placed in the middle of two communicating parties, thus allowing the fuzzer to intercept and alter the messages passed in the network [101]. In the security realm, fuzzing is an effective way to identify corner-case bugs and vulnerabilities. Still, mutation fuzzers need help identifying corner cases or rare scenarios leading to bugs or vulnerabilities.

(3) In *evolutionary fuzzers*, genetic programming converges toward finding the potential security vulnerability in the target [46]. So it is a continuous process of input set creation that depends on the fuzzing framework and the response from the target. Therefore, it uses metrics such as a score that defines the goodness of the input dataset, discarding the lowest-scored input dataset, mutating the remaining good-scored input datasets, and combining the input datasets with the highest score to achieve optimal code coverage. Nevertheless, there could be a considerable struggle while generating the test cases, since there are chances for generating test cases similar to previously generated inputs, leading to redundancy in testing. Additionally, enhancing the ability to identify vulnerabilities or bugs requires more extensive program behavior exploration.

As a whole, fuzzing is an effective technique for identifying vulnerabilities or bugs in software. However, the limitations must be considered when selecting a fuzzing process for a particular software system. We can also classify fuzzers based on the degree of program analysis concerning the target and classified as black-, white-, and grey-box.

(1) *Black-box fuzzing* randomly mutates well-formed input data and then supplies the modified input data to the target [12]. We can transform input data with the help of mutation or generation-based generators. A detailed log report during the process of execution stores the test cases used and the corresponding behavior of the target application.

(2) *White-box fuzzing* also starts with well-formed input data and then symbolically executes the program dynamically [65]. The symbolic execution technique is based on program analysis and constraint solvers to capture the control flow path of the program. It uses **Satisfiability Modulo Theory (SMT)** [10, 39] solvers (for example, the Z3 solver from Microsoft [38]) and quantifier-free first-order logic formulas. Therefore, the quantifier-free, fixed size, and bit-vector logic backed by SMT solvers directly support the accurate description of arithmetic at the assembly level. It helps to perform direct translation from the instructions executed to the primitives provided by the SMT solvers. The SMT formulas represent the path condition of the program under test. We gather all the constraints on the inputs from the conditional branches on the execution path. Later, we negate all the constraints encountered during the execution path and solve those constraints using a constraint solver. All the new solutions obtained via the constraint solver are considered as the new inputs and expect those inputs to explore new paths unvisited so far. The seeds get generated to cover all the constraint sets, including the unsatisfied constraints obtained so far. Figure 2 shows an example of a constraint-guided fuzzing process. It captures unsatisfied constraints and branches not traversed during its execution and generates new seeds for further traversing of the source code.

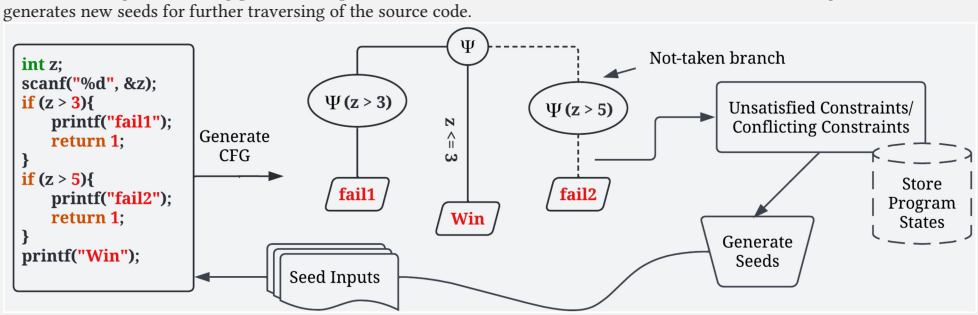

Fig. 2. Constraint-guided fuzzing process.

(3) *Grey-box fuzzing* instruments the target program on a few control locations [141]. Instrumentation is performed during the compile-time and provides an initial seed as a test case. The generated seeds intend to cover the code block on the appropriate control locations [66, 109, 122] and are known as coverage-based fuzzing. Grey-box fuzzing claims more coverage on the program code, and the code can have basic blocks that form the tree's nodes, and edges are control paths that connect the block codes [141]. This kind of graph-based approach is well suited for search-based software testing. Furthermore, constraint-guided directed grey-box fuzzing also targets a few control locations. It generates a sequence of constraints to traverse through each code block. Each constraint, coupled with certain data conditions, gets mapped to its target site. The input seeds generated will cover all the constraint sets and the unsatisfied constraints obtained during the execution.

The generation and selection of suitable test cases are critical in fuzzing. The selection of the best test cases has received much attention nowadays, and generating test cases has many shortcomings. For example, suppose the generation of test cases uses a random approach; then recycling *interesting* test cases is not happening. Recycling *interesting* test cases must reuse the previously generated test cases identified as *interesting* to improve the fuzzing process's effectiveness. It involves the selection of suitable test cases that have triggered interesting behavior, such as crashes or hangs, and using them as starting points for subsequent fuzzing iterations. Even if we use sequence strategy, that is, generating a sequence of test cases with variations in input values or data

structures related to the initial *interesting* test case, recycling of *interesting* test cases is not occurring though it tests each test case. However, the challenge in identifying the most promising *interesting* test cases to use as starting points and generating and selecting related test cases is time-consuming and resource intensive. Additionally, there is a risk of overfitting to specific sequences of input values or data structures, which can limit the ability of the fuzzing process to explore the complete scope of application behavior. The test case generation for a fuzzer could be random or grammar-based.

(1) *Random test case generation* [59, 104, 109, 114, 169] is a prevalent technique used for testing an application by injecting random and unpredictable inputs to determine vulnerabilities and bugs. It helps to generate new test cases that have yet to be explored during the evolution or testing phase of the code. Nevertheless, the boundary is restricted in its ability to generate test cases that surpass the capacity of its mutation capabilities. Therefore, it can only generate test cases that are within the boundary of the existing seed corpus. Suppose the seed corpus does not contain a specific input pattern or data structure. In that case, it will not be able to generate test cases that can explore the behavior of an application under those specific conditions. However, mutation-based techniques such as block-based mutation can manipulate input data at a higher level of abstraction. It can generate new test cases by transforming the input data structure (addition/removal) instead randomly changing individual bytes. Furthermore, fuzzers [24, 89, 114] also give prevalence to construct test cases based on the scheduling behavior of an application that implies real-time related features like scheduling, timing, and priority. The **Application Under Test (AUT)** gets analyzed to apprehend its scheduling behavior, such as resource allocation and task/event scheduling, thread scheduling, interrupt handling, time-slicing, locking, and synchronization. For instance, PrIntFuzz [115] generates test cases that intrigue to interrupt handler code bugs in device driver and kernel code. A state-aware task generation fuzzer Rtkaller [153] initializes tasks as test cases to identify bugs linked with code concurrency intensity, shared memory synchronization, resource allocation, and task/event scheduling in RTOS. AutoInter [99] identifies the concurrency bugs, likely referring to the systematic control of thread interleavings during the execution. UltraFuzz [186], a centralized dynamic scheduling-scheme fuzzer, evaluate seeds and dispatch tasks in a centralized manner and checks for bugs related to resource allocation. It also applies smart scheduling that prioritizes fuzzing tasks based on factors like code coverage, crash severity, or the likelihood of finding bugs. Undoubtedly, such fuzzers are used to fuzz operating systems, including RTOS, device drivers, and network or security protocols, since all these software or applications have complex and challenging scheduling behavior. It can effectively generate test cases specifically tailored to schedule the behavior of AUT. However, it can also be challenging and computationally resource intensive, as it requires significant knowledge of the scheduling behavior of the AUT and hence demands the utilization of resources intensely to generate the test cases.
(2) An extended thought was to provide a better capability for generation and mutation-based fuzzer using *grammar-based test case generation* [3, 46, 64, 116, 154, 164]. The grammar specifies the rules for valid and invalid inputs, such as the structure of input files, the type of characters allowed, and the length of inputs. The parse grammars usage into the fuzzer models helps evolve various test cases. Therefore, the critical point is to generate syntactically correct new test cases from the parsed grammar. A syntactically correctly generated test case can help the fuzzer explore the target's less-visited paths; however, it can contain unexpected or malicious content. Since the grammar of the input language is used to generate the set of rules that define the valid syntax and structure of the input, and hence the rules can be used to generate new test cases by randomly selecting valid sequences of tokens that adhere to the grammar. Moreover, inlining code snippets into grammatical structures helps explore complex paths in the code and perform complex actions by making precise decisions that cannot carry out using context-free grammar. However, it requires significant knowledge of the input language grammar and may require the development of custom grammar for complex or proprietary input languages. Additionally, the generated test cases may not always contain unexpected or malicious content, and further mutation or generation is required to increase the likelihood of finding vulnerabilities and bugs.

Furthermore, the techniques accompanied in fuzzing to find potential security vulnerabilities include coverage-guided, symbolic, and concolic execution. These techniques have to augment the identification of severe bugs and explore the deeper code blocks in the target.

(1) *Coverage-guided* is a well-versed fuzzing technique that helps analyze the code coverage metric deeply, and it aids in measuring the quality of tests that decides areas of code blocks that require a more profound test [109, 127]. Besides, it automatically generates test cases or inputs that trigger bugs or potential security threats in the target. We can call it a coverage-guided mutational fuzzing test, since test cases are generated by mutating the existing inputs. The mutation operations are carried out at the bit or byte level, flipping, deleting, duplicating, or adding random subsequences of bytes into the inputs. The mutant version's new inputs become *interesting* if it covers new code blocks or paths in the target. We save these *interesting* mutant inputs and restart mutation operations on these *interesting* ones. Therefore, we must save this new coverage mutant input, termed coverage-guided in its algorithm. However, if the input space is simple, then high code coverage is assured, but it will miss the bugs that require complex inputs to trigger. Furthermore, fuzzing may get stuck at the local minima of the code coverage, which means some paths will not get explored and limit its input space exploration. If the input space is complex, then converging on a bug will take a long time, since it may have to create many test cases to trigger that bug. Moreover, complex data structures, such as nested or linked lists, require complex inputs to trigger the bug.

(2) Another exciting technique in fuzzing is *symbolic execution* to gain maximal code coverage. In this case, we symbolically execute the inputs and then collect the constraints (symbolic) on that inputs [9, 155, 173, 180]. All the collected constraints get negated to generate new inputs that help cover new code blocks that have not been observed so far during the execution of the target. It generates expressions that define the program manipulation over the symbolic inputs that resort to SMT solver queries [38]. It checks for both the directions from a branch condition and sets one path that must hold *true* in each program state (*true* branch), and the negated condition gets added to the program state that holds *false* in each program state (*false* branch). However, symbolic execution is computationally expensive, since many constraints could be solved while exploring multiple paths, leading to path explosion. Also, the limited support to solve the floating point arithmetic operations restrain symbolic execution in fuzzing. Additionally, symbolic execution creates conflicts while handling system calls, since it does not support modeling all possible system calls and inter-process communication, such as pipes or sockets. Likewise, the non-deterministic behavior of system calls complicates the generation of inputs that consistently trigger specific paths.
(3) In the case of *concolic execution*, the symbolic constraints get analyzed with concrete values when the execution paths taken by the program run over the specific code blocks [9, 30, 155, 177]. It is powerful, since it uses concrete values. However, it can also suffer path exploration problems, since branch conditions have two paths (*true* and *false* branch or taken and not-taken branch conditions). Hence, the program state has to fork both paths and demands exploration, which may go to an infinite state. However, path explosion can severely limit the scalability and handling of floating-point arithmetic in concolic execution, leading to inaccuracies during constraint solving.

1.2 Paper Organization

The rest of the article is organized as follows: Section 2 outlines the existing fuzzing categorizations and surveys conducted based on those categorizations, followed by the research gaps identified, our contributions, and the research questions phrased. Sections 3 and 4 discuss contemporary research on fuzzing in the absence of source code and directed fuzzing in conjunction with static analysis. We have addressed the research questions phrased and explained the concepts with the support of recent research works in fuzzing. In Section 5, we have conducted an in-depth case study by shedding light on symbolic and concolic execution fuzzing. Section 6 reviews various fuzzers used for interface and environment fuzz, and Section 7 inspects fuzzing for improving finding and fixing instrumentation errors. Later, in Sections 8 and 9, we discussed summary, challenges, and potential future directions.

2 ORGANIZATION OF EXISTING FUZZING WORK

For the last several years, fuzzing has been categorized as follows, and the state-of-the-art fuzzers are grouped into each of the following classes. However, we can observe intriguing overlaps between these fuzzers irrespective of their categorizations. Furthermore, we have extensively surveyed fuzzers based on the methods, types, or techniques accustomed to and listed them in Table 1.

(1) *Influence-oriented fuzzers*: white-box such as SAGE [65], KLEE [19], S2E [29], Black-box such as BeSTORM [12], Autodafé [119], Choronzon [22], and Dharma [124] and grey-box such as AFL [122], libFuzzer [109], and Honggfuzz [66].
(2) *Mutation-oriented fuzzers*: AFL [122], libFuzzer [109], and Honggfuzz [66].
(3) *Semantic or Grammer-oriented fuzzers*: GWF [64], NAUTILUS [3], EVOGFUZZ [46], Superion [164], Pythia [5], G-EVOSUITE [130], GramFuzz [73], and Dharma [124].
(4) *Feedback-oriented fuzzers*: directed-fuzzers such as DGF [14], Zipr [76], TaintScope [167], and Coverage-guided fuzzers such as AFL [122], libFuzzer [109], and Honggfuzz [66].

2.1 Outline and Survey Overview

We have summarized those surveys based on their applications, benefits and drawbacks, techniques, different fuzzing tools, and methods as outlined in Table 1.

- The survey conducted by James Fell et al. [49] has examined various software vulnerabilities and detection methods, such as static and dynamic analysis on software source code, and provided insight into fuzzing and the form of instrumentation used to the target application.
- J. B. Crawford [83] has pointed out classic security vulnerabilities/crashes due to arbitrary code execution and discussed the evaluation of software to discover the stability and security issues.

- The survey [104] summarized the recent advances in fuzzing and its improvement over the past couple of years and discussed its types, giving importance to coverage-based fuzzing.
- The review [107] discussed the generic fuzzing framework and its categorizations such as black-, white-, and grey-box. Furthermore, it provided an overview about the state-of-the-art fuzzing mechanisms by exploring typical fuzzers and their application areas.
- Yan Zhang et al. [181] have provided a quick overview of the influence of symbolic execution in fuzzing and a short analysis of directed grey-box fuzzing tools.
- Reference [117] focused on the taxonomy of fuzzer concerning black-, white-, and grey-box with an insight toward the instrumentation, test corpus selection, trimming, and generation.
- Gary J. Saavedra et al. [148] have explored leveraging ML algorithms such as unsupervised, reinforcement, and deep learning in generation-, mutation-, and evolutionary-based fuzzers.
- Reference [166] focused on a deep understanding of the grey-box fuzzing technique. It analyzes the metrics such as edge coverage, seed prioritization, optimization, exploration and exploitation, mutation scheduling, and dataflow graphs analysis.
- Reference [63] presented fuzzing as a hunting software for discovering vulnerabilities by using black-box random fuzzing, grammar-based fuzzing, and white-box fuzzing. The effectiveness of all the fuzzers is tested by analyzing the applications fuzzed using test corpuses formats such as jpeg, png, manually written grammars, and XML or JSON dialects.
- M. Böhme et al. [13] summarized the challenges in black- and grey-box fuzzers, since they struggle to generate inputs to bypass frequent paths visited and discusssed the problems in white-box (symbolic) with constraint solvers using SMT [38].
- The survey [168] systematically classifies and categorizes the ML algorithm distribution in fuzzing using traditional ML, deep learning, and reinforcement learning, employing preprocessing methods from Natural Language Processing to transform input data into vector representations.
- Reference [180] has conducted a short survey that presents the importance of hybrid fuzzing for automated vulnerability mining in coverage-oriented fuzzing.
- The survey propounds by Xiaogang Zhu et al. [188] has provided an in-depth discussion on fuzzing, its features, and various components that identify the gaps in defect detection by narrowing down the gap between test case generation techniques.

While those surveys focus on theories/methodology concerning fuzzer types, we envisage and explore the dimensions of fuzzing in a broader aspect concerning the application domains, techniques, and the target. For simplicity, we use some of the recent fuzzing research works to address the research questions raised.

2.2 Research Gaps Identified

Despite the effectiveness of various methods, types, test case generation, and techniques in fuzzing, several research gaps still need to be addressed. Below are some technical research gaps in fuzzing:

(1) Instrumentation modifies the application's source code or binary to accumulate supplementary data during runtime. However, most fuzzers are not concerned with deserting the unreachable paths. Additionally, instrumenting code or binary may be difficult or impossible for specific software applications, such as those written in low-level languages like assembly.
(2) Generate appropriate inputs that trigger a complex bug in the code, for example, the bugs related to memory errors in the code. However, we still encounter research gaps concerning identifying common coding mistakes, such as memory errors, buffer overflows, format string vulnerabilities, and integer overflows, which are all potential targets for fuzzing.
(3) Improvising coverage paths by visiting the maximum paths irrespective of code deepness with targeted input generation.
(4) Despite having generic fuzzers, specific application fuzzers must concentrate on sensitive code blocks based on their service. For example, fuzzers to test firmware and emulation/OS kernel code must capture bugs or vulnerabilities during memory interactions, interconnections with sockets, pipes, and so on. However, fuzzing such interaction code hits several limitations, missing the bugs and earning reduced code coverage.
(5) Errors in instrumentation code introduce additional overhead. Hence, we need to detect/correct instrumentation code errors.

Table 1. Summary on Prior Survey Articles

Ref. }: **R1** [49], **R2** [83], **R3** [104], **R4** [107], **R5** [181], **R6** [117], **R7** [148], **R8** [166], **R9** [63], **R10** [13], **R11** [168], **R12** [180], **R13** [188]

Fuzzers }: **1**-AFL [122], **2**-SPIKE [37], **3**-Honggfuzz [66], **4**-libFuzzer [109], **5**-Clusterfuzz [33], **6**-PEACH [137], **7**-Tscope [167], **8**-AFLFast [17], **9**-AFLGo [14], **10**-REDQUEEN [4], **11**-Driller [155], **12**-QSYM [177], **13**-T-Fuzz [140], **14**-Angora [26], **15**-Miller [158], **16**-VUzzer [146], **17**-KLEE [19], **18**-SAGE [65], **19**-Sulley [138], **20**-FAIRFUZZ [103], **21**-KATCH [118], **22**-BUGREDUX [92], **23**-MLFuzz [120], **24**-S2E [29], **25**-Sym. Path finder [145], **26**-BFuzz [61], **27**-Token-based [64], **28**-CESI [116], **29**-GANFuzz [79], **30**-NEUZZ [152], **31**-DEEPFUZZ [110], **32**-VFuzz [106], **33**-Skyfire [163], **34**-PANGOLIN [81].

Ref.	Year, and Highlight	Grouping of Fuzzers	Fuzzers Discussed	Major Viewpoints
R1	2017 Code coverage	MG	1	■Software vulnerabilities ■Static analysis methods ■Genetic algorithms for fuzzing
		GW	2	
R3	2018 Security threats	GW	2, 6	■Vulnerability discovery ■Smart fuzzing ■Testcase generation ■Fuzzing OS kernels
		MW	15	
		MG	1, 7, 11, 16	
		MB	4, 18	
R5	2018 Integrated fuzzing	GB	6	■Heavyweight program analysis ■Directed white- & grey-box ■Integration of fuzzers ■Based on symbolic execution ■Access-control vulnerability
		CG	1	
		DS	7	
R7	2019 ML tools to overcome fuzzing challenges	MNAF	6	■Awarness of input formats or not ■Gnerate interesting input testcases ■Scheduling of input testcases ■Use of unsupervised learning ▸ Genetic algorithm, deep learning ■Use of supervised learning ▸ Reinforcement learning
		GAF	6, 19	
		EMF	1, 3, 4	
R9	2020 Effectiveness of fuzzing	RMB	1	■Security vulnerabilities ■Classifications based on: ▸ Combination of fuzzing methods ■Effectiveness of combination of: ▸ Various fuzzing techniques ■Structuring the input formats ■Applications require fuzzing ■Challenges in fuzzing ▸ Automation of input formats ▸ Exhaustive symbolic testing ▸ Cost-effective test approaches ▸ Fuzzing cloud services
		IG	2, 6, 19	
		WIW	17, 18, 24, 25	
		HGeBW	1*	
		HG	26*	
		HBGwW	11, 12	
		HGW	27, 28	
R11	2020 ML techniques for fuzzing	GMMLG	29, 30, 31, 32	■The use of ML techniques ■Examination based on: ▸ Seed selection, Testcase generation ▸ Filters, Mutation methodology ▸ Fitness metrics, Exploitability analysis
		CMLG	30, 31, 32, 33	
R2	2018 Statbility security	MG	1, 3, 4	■Security vulnerabilities ■Automated fuzzing ■Test instrumentation
		MGB	5	
R4	2018 Correctness security	GB	6	■Software bugs ■Generation of testcases ■Fuzzing techniques ■Open problems in fuzzing
		GW	18	
		CG	1	
R6	2019 System modeling of fuzzers	MG	1, 3, 4, 8, 9, 10, 11, 12, 13, 14	■Taxonomy of fuzzers ■Modeling of fuzzers ■Testcase validation, minimization, and types of mutation ■Fuzz testing algorithms ■Pre-processing techniques ■Scheduling techniques
		Mo/GrB	6, 15	
		CTW	11, 12, 13, 16, 17, 18	
		TAG	7, 14, 16	
R8	2020 Grey-box fuzzing coverage	CG	1, 20	■Directed grey-box fuzzing ■Bugs and vulnerabilities ■Target location reachability ■Challenges in directed grey-box ■In-depth analysis on: ▸ 32 starte-of-the-art fuzzers
		DS	17, 21, 22	
		HS	23	
		PCD	9	
R10	2020 Summarization of open challenges in fuzzing	M/GwB	6	■Heavyweight program analysis ■Directed white- & grey-box ■Integration of fuzzers ■Based on symbolic execution ■Access-control vulnerability
		CFG	1, 3, 4	
		SEW	17, 18	
R12	2020 Automated software vulnerability mining	HG	1, 4	■Combination of: ▸ Constraint solving ▸ Coverage techniques ■Combination of fuzzing tools
		MG + MW	11, 12, 34	
R13	2022 Defect detection in applications and software	ISR	Fuzzers form 2009-2021	■Input space reduction: ▸ Dynamic taint analysis ▸ Seed selection ▸ Schedule mutation operations ▸ Program transformation ▸ Format inference ▸ Concolic execution ▸ Dependency inference
		MG + MW	Fuzzers from 2009-2021	

MG-Mutation (Grey-box), **MB**-Mutation (Black-box), **MW**-Mutation (White-box), **GW**-Generation (White-box), **GB**-Generation (Black-box), **MGB**-Mutation (Grey- & Black-box), **CG**-Coverage (Grey-box), **Mo/GrB**-Model/ Grammar (Black-box), **CTW**-Constraint (White-box), **TAG**-Taint Analysis (Guided-fuzz), **DS**-Directed (Symbolic), **MNAF**-Mutation (No awareness on expected input format), **GAF**-Generation (Awareness on expected input format), **EMF**-Evolutionary (Built on top of mutation fuzzers), **HS**-Heuristic (Search-based), **PCD**-Path Coverage (Directed Grey-box), **RMB**-Random Mutation (Black-box), **IG**-Input Grammar (Grammar-based), **WIW**-Well-informed Input (White-box), **HGeBW**-Hybrid (Grey-box extends Black-box with White-box), **HG**-Hybrid (Grey-box), **HBGwW**-Hybrid (Black-box or Grey-box with White-box), **HGW**-Hybrid (Grammar-based with White-box), **M/GwB**-Mutational or Generational (Black-box), **CFG**-Coverage-feedback (Grey-box), **SEW**-Symbolic Execution (White-box), **GMMLG**-Generation or Mutation (ML-based Grey-box), **CMLG**-Coverage (ML-based Grey-box), **ISR**-Input Space (Reduction)

2.3 Scope, Contributions, and Research Questions

In light of the existing surveys and the research gaps identified, we present a deep and thorough survey of fuzzing and its counterparts by taking a holistic direction on various approaches. The following are the contributions of this article:

(1) We have presented this survey by endorsing different methods of fuzzing based on their application domains and techniques.
(2) We differentiate the fuzzing approaches starting with *in the absence of source code* and *in conjunction with static source code analysis.*
(3) We have explored *symbolic and concolic execution* by taking a fuzzer as an example to investigate the bright and dark shades of fuzzing on real-world applications in real-time.
(4) Later, we examined the specific target fuzzing domains, such as *interface and environment issues* in firmware/kernel code. Additionally, we researched fuzzers designed to address instrumentation errors.
(5) Furthermore, we have researched and examined some of the associated challenges.

We have framed the following Research Questions (RQ) based on categorizations in (2), (3), and (4).

RQ1: (a) What if the software source code is unavailable for instrumentation, and (b) if the commodity software is with closed source code, then how do we conduct instrumentation? (c) Besides, if the applications with complex build systems do not support recompilation or recompilation tends to cause multiple errors, how can it get fuzzed?
RQ2: (a) How does fuzzing work in conjunction with static analysis, and (b) comprehend the feasibility of using coverage-guided or mutation-based fuzzing with static code analysis? (c) If yes, then what are the benefits/limitations compared to existing methods?

RQ3: (a) How do we blend fuzzing with symbolic and concolic execution, and (b) realize the use of preconstraints to add constraints for symbolic execution? (c) Once the constraints are added, how is it solved to explore the state space of the target source code?
RQ4: (a) Can modern fuzzers address the runtime interface and environment dependencies of fuzzing targets? and (b) if yes, any challenges?
RQ5: (a) Who will check the correctness of the instrumentation code, and (b) what if it possesses some errors (missed/redundant locations)? (c) Will these instrumentation code errors affect the coverage feedback and accuracy, and (d) if yes, how profound is it, what are the ways to fix it, and whether fixing benefits fuzzing?

We address each of the above research questions phrased in the following sections based on the categorizations that we have articulated. To the best of our knowledge, this is the first article covering the survey on fuzzing techniques by endorsing where it is getting applied.

3 FUZZING IN THE ABSENCE OF SOURCE CODE

In this section, we delve into the realm of fuzzing scenarios where access to the source code of the target program is limited or unavailable. This perspective aligns with black-box and grey-box fuzzing methodologies, which involve testing software behavior without complete source code knowledge. Notably, the effectiveness of black-box fuzzing is recognized to have inherent limitations. As we explore strategies within this context, we focus on leveraging coverage information to enhance the efficiency and effectiveness of fuzzing on binary code, even when full access to source code is constrained. By adopting this approach, we aim to showcase the potential for achieving significant results through fuzzing, even when more than comprehensive source code understanding is feasible.

Directly fuzzing the binary allows for far more accurate testing of the actual code that will run in production. Additionally, the available debug information helps map the binary back to the code, often included in binaries, making it easier to perform grey-box fuzzing on the binary level. Besides, fuzzing the binary provides better protection against reverse engineering. Also, the state-of-the-art fuzzing tools, such as AFL [122] and libFuzzer [109], are designed to work with binary rather than code and ensure better coverage of program behavior, including interactions with the operating system and other external resources. However, we also have techniques that allow for grey-box fuzzing on the source code level, such as instrumentation-based or code coverage-guided fuzzing. Nevertheless, these techniques may require additional setup and may not be as widely used as binary-level fuzzing. To address RQ1(a), we seek binary rewriting or instrumentation (instrumentation code inserted directly into the binary). In the case of commodity software with closed source as in RQ1(b), we can follow binary instrumentation at different levels; for example:

- *Hardware-assisted* such as PTfuzz [179], Intel PT [82], PTrix [27], and kAFL [150]; *dynamic binary instrumentation* such as PIN [113], QEMU [7], DynamoRio [123]; *Static binary rewriting* such as AFL-Dyninst [11, 159], E9Patch [45], and E9AFL [62].
- Exploring a scenario involving hardware-assisted fuzzing, we gather runtime traces while recording information about every executed state. It captures the runtime execution information using dedicated hardware, and all the runtime captured data get through post-process to obtain detailed coverage information. However, collecting runtime traces is expensive, requiring a minute strategic mechanism and substantial effort to capture individual blocks at a high rate. In dynamic binary rewriting, the instrumentation gets performed during the execution. An example scenario is using AFL-QEMU [122], one of the best attractive solutions for researchers to rewrite the binary of complex code commodity software on the fly. So we have to trap the execution of individual basic blocks and proceed with binary writing while it gets executed. It is a sound technique for binary instrumentation but at the cost of significant overhead due to heavyweight emulator configurations using PIN [113] and QEMU [7]. In the case of static binary rewriting, it disassembles and rewrites the binary before it gets executed. For example, AFL-dyninst [159] gets performed offline, where we rewrite the binary before it starts execution. Since it is static, we can utilize complex source code analysis techniques. Optimization methods can perform in memory and reduce the runtime overhead, similarly to compiler-level optimations for the source code instrumentation. Unfortunately, trampoline-based approaches in static binary rewriting have challenges, such as the separation of code data interleaving and inlined data.

Section 3.1 will discuss various recent binary fuzzers that address RQ1(a), (b), and (c), which claim to have betterness in identifying bugs, coverage of codes (block or edge), hit counts, and so on. To address RQ1(c), we resonate with the thoughts related to instrumenting a program at its source code level. In that case, creating a matching compilation environment and building the

source code with the modified compiler (afl-gcc, afl-clang, or afl-clang-fast) is essential. The ability to recompile the target application with instrumentation or other modifications is often helpful for enabling certain types of fuzzing techniques. However, creating a matching compilation environment is a considerable task for applications with a complex build process, such as Firefox or Chrome. Also, modifying the build system can lead to multiple fallacies. Likewise, there are situations where the target application may have a complex build system, or, recompiling the application, introduce errors, making recompilation difficult or impossible. For example, in the case of binary instrumentation, recompilation is impossible. However, binary instrumentation can modify the binary, enabling coverage-guided fuzzing or other advanced techniques. Furthermore, suppose the target application has a well-defined protocol, API, or interface. In that case, protocol and API-level fuzzing can be used to test the application without recompiling or modifying it. We have detailed the answers for the research questions and showed specifically that RQ1(c) is better explainable using those techniques based on the user's need or fuzzer performance expected.

3.1 Recent Binary Rewriting Fuzzers

BEACON [80]. There is a presumption that performing reachability analysis on a specified target can improve the fuzzer to explore the bugs and vulnerabilities in a binary. However, the statement mentioned is partially correct if we use the concept of directed white-box [75] or grey-box fuzzing [14] but at the cost of computational resources. This contradiction arises, since directed fuzzing does not prune the paths when it hits an instruction that cannot reach the target. Furthermore, the paths that are reachable to the target at the cost of solving an unsatisfied path condition also occur in directed fuzzing. Relying on symbolic [118] and concolic execution [19] may reach the target. Still, it is equally expensive, since many constraints are to be solved to acquire the target.

However, direct grey-box fuzzing relies on prioritizing the seeds based on the likelihood of reaching a target, which depends on some meta-heuristics. Hence, it is essential to prune the do-not-contribute paths in directed fuzzing. BEACON explored the essentiality of using static code analysis as pre-processing, which analyzes the source code and computes abstract pre-conditions. At the same time, it is essential to ensure that static code analysis will not create a significant overhead. They have carried out selective instrumentation that instruments only two kinds of statements: branch and variable-defining. It reduces a significant amount of overhead that could have occurred during static instrumentation on each LOC to identify the pre-conditions.

Furthermore, deciding on an interval domain is suitable, since it is the most affordable method achieved through lightweight static code analysis. For that, strategies called relationship preservation and bounded disjunction are used. The former helps preserve the relationship among the variables to achieve better pre-conditions and thus prune more do-not-contribute paths to the target. The latter provides a better boundary regarding the values to be considered for a variable, avoiding unnecessary infinity-bound values during exhaustive path merges. BEACON has been evaluated against benchmark programs such as libpng, libjpeg, lrzip, libxml, binutils, and so on. The observation is that BEACON had outperformed the state-of-the-art conventional coverage-guided fuzzers such as AFL [122], AFL++ [52], and MOPT [114] concerning the reproduction of bugs, speedup, and identification of exciting **Common Vulnerability Enumeration (CVE)** bugs.

E9AFL [62]. As discussed above, we go for binary rewriting if we do not have the source code to perform instrumentation. E9Patch [45] is a state-of-the-art static fuzzing system that rewrites the binary. The trampoline-based rewriting methodology combined with instruction punning gets used in E9Patch. Trampoline-based approaches select instructions and replace them with jumps to trampolines. Instruction punning takes care of initialization and memory management without disturbing the application's address space during instrumentation. However, trampolines-based

approaches break the code's continuity, leading to significant fuzzing runtime overhead (extra jumps to/from trampolines) and, later, to unnecessary page faults. To overcome such issues, E9AFL is built on top of E9Patch, which introduces additional mechanisms such as trampoline ordering, instruction selection, and bad basic block eliminations.

We must make the trampoline memory contiguous to reduce page faults and remove unnecessary trampolines. The proposed model initially allocates the trampolines correctly to achieve such an optimization, and it will, in turn, help to address the page faults due to the trampoline break. Later, the model only selects the best instructions for better trampoline ordering. The last step is eliminating all the bad blocks by identifying the redundant instructions likely to cause page faults. For contiguous allocations of the trampoline, allocate trampolines in the same order as in the patched instructions. This detailing helps map the code regions in the same trampoline memory and reduce page faults. Regarding instruction selection, we know that AFL [122] instrument the code at the start of each basic block by default. However, irrespective of considering the beginning of a basic block, we can insert the instrumentation code anywhere inside the basic block. Such an effort will maintain the functionality of the basic block. Hence, E9AFL uses an instruction selection algorithm that finds the instructions greater than five bytes in a basic block, thus allowing trampoline ordering in many basic blocks. If the instructions are less than five bytes, then such basic blocks are considered bad, so trampoline ordering will not apply. All the optimized and unoptimized basic blocks get represented with the help of **Control Flow Graphs (CFG)**. Those with indirect jumps or calls to targets are treated as bad basic blocks and eliminated. It achieves the best performance of afl-gcc with comparable speed and code coverage on various fuzzing benchmarks. Furthermore, E9AFL fuzzed the Google Chrome binary to demonstrate its effectiveness and scalability.

STOCHFUZZ [182]. Static binary rewriting techniques assume that there are no inlined data. We may disable inline data during compilation for performance analysis, since inlining can make data difficult to interpret. However, such a default assumption leads to missing necessary code blocks if the compiler copies the code from one block to another during a call instead of creating a separate set of instructions in the memory. Also, the indirect jumps can overlap with other instructions. The virtual address space layout for the instrumented binary in E9Patch [45] exposed enormous cache misses and extra overhead in process forking.

In most cases, static symbolization in static binary rewriting is challenging, since it may have to convert address-related immediate values in the binary to symbols much more frequently. Since fuzzing is a repetitive process, it would be suitable to collaborate on an incremental and stochastical approach that piggybacks the fuzzing routines. Here it relies on probabilities to model the uncertainty regarding the separation of code, data, and inlined data. The model does not require an initial binary analysis in such a case, and instead, the model conducts rewriting the binary based on the uncertain results. Since the technique encourages stochastic modeling, it is not worried about the problems occurring in the initial stage of binary rewriting. Because in the repetitive fuzzing process, the problematic code sections get identified, and the model will rectify them in each subsequent run. A probabilistic random determination approach earns the control to correct issues in the code sections based on the sample generated in each run. Rewriting the bytes at some addresses is determined based on the possibility that the address is an instruction or not. Therefore, the model checks and computes the likelihood of each instruction address, indicating whether it is data or code. However, it is evident that if too much rewriting is required, then fuzzing accuracy will decrease. So it was specific that there should be little rewriting; otherwise, the fuzzing process can become incompetent. Progression during this repetitive process is phenomenal in incremental and stochastic fuzzing processes to identify vulnerabilities. STOCHFUZZ outperformed state-of-the-art binary-only fuzzers, which were backward in soundness and overhead.

ZAFL [126]. In source-code-level fuzzers, it is transparent that the coverage gets collected via the instrumentation code inserted in the target source code [66, 109, 122]. Hence, the coverage information invariably includes the instrumented codes in the basic blocks. In most cases, such compiler-level instrumentations produced high throughput with low or high overhead and improved the finding of many new bugs [66, 109, 122]. However, compiler-level instrumentation is impossible if the target source code is unavailable. Thus, the option left behind is to perform binary-level instrumentation. Even though we have many binary-level instrumentation fuzzers, it lacks performance because of the difficulty of instrumenting the binary, high overhead, and the complexity of learning semantics [7, 113]. So far, binary-level fuzzing techniques concentrate on hardware-assisted fuzzing, which is incapable of modifying the binary, and thus fails to achieve fuzzing enhancement on program transformation [60, 165]. Another option suggested is dynamic or static binary rewriting, which fails to achieve less overhead. Analysis infers that the average overhead of dynamic binary rewriting tools such as AFL-QEMU [122] is more than 600%, and DynamoRio [123] and PIN [113] reported an overhead between 10× to 100×, respectively. The static binary rewriting tool DynInst [11] possesses an overhead of 500%, and RetroWrite [41] relies on AFL's assembly-time instrumentation, which is 10–100% slower than compile-time instrumentation. Besides, RetroWrite does not support any program transformation. Therefore, an open window exists that expects performance enhancement fuzzers that help improve program transformations at the binary level, maintaining the compiler-level performance achieved in source code fuzzers.

The program transformation through ZAFL for fuzzing enhancement on real-world binaries are based on contrasting factors, scope, complexity, and platform. ZAFL extracts the **Intermediate Representation (IR)** code from the binary and performs optimization, which scans the target binary's CFG to determine points of interest and apply IR-level transformations. The altered control flow of the IR demands analysis, and hence ZAFL checks the liveliness of the registers involved and flow-demands of the IR meeting the desired target point or not. Later, determine the instrumentation points and apply compiler-based techniques on the IR. The current prototype of ZAFL has achieved more unique crashes and test cases with a drastic reduction in overhead.

HEXCITE [127]. Most fuzzers neglect the parameters such as edge coverage and hit counts, since they mainly target coverage of the basic blocks and identification of bugs. However, in fuzzing, the need for edge coverage and hit counts have equal importance as basic block coverage [125].

> A basic block is a single entry and exit point between the instruction sequences representing the start and endpoint during the control flow transfer. In contrast, an edge represents the transition from one basic block to another and the number of respective paths covered during those transitions. Regarding hit counts, it conveys the frequency in executing a block or an edge, which helps to observe whether the condition inside the basic block or the edge transition leads to state exploration (for example, AFL [122] use bucketed hit count method). Therefore the term coverage-preserving should guide basic block coverage, edge coverage, and hit count coverage while fuzzing a binary. For that, the researchers leverage the coverage-guided tracing [41, 52, 125] technique in fuzzer, which assures high throughput by reducing the cost of coverage tracing if and only if a possibility of new coverage exists. However, ensuring coverage-preserved fuzzing guidance on binaries is not as easy as coverage achievement on software source code. Integration of coverage-guided routines to a binary is complex due to its semantically inadequate nature.

HEXCITE tries to improve this by giving adequate prominence to coverage preservation and tracing. Coverage-guided tracing improves binary-oriented fuzzing by reducing coverage tracing expense to fewer test cases that meet new coverage. HEXCITE proposed a concept, Jump mistargeting, to support edge coverage. Usually, the fuzzers split the critical edges with some dummy blocks. Thus, additional instrumentation with too many new instructions to be processed in each execution. This situation leads to high overhead, and fuzzing becomes slower on the binary. Therefore, statically alter the *jump* instruction associated with the edge and redirect the edge to some other points. Hence the unnecessary interrupt insertion in coverage-guided tracing gets bypassed, and it permits signaling of the critical edge coverage without any need for many dummy instructions.

The bucketed hit count method of AFL [122] of libFuzzer [109] consumes much time by calculating hit counts for every loop block. The new hit count coverage gets localized to loops such as *while* and *for*; hence, an iteration count is tracked by monitoring its induction variable and leads to expensiveness in terms of time consumption. Therefore, a novel technique called *bucket unrolling* in HEXCITE augments each loop header with sequential condition statements that weigh the loop induction variable against the expected hit count bucket range. Therefore, each conditional block gets assigned an interrupt to support coverage-guided tracing. Thus the target *jump* instruction inside the conditional block gets directed to the loop body. It ensures that there is no change in the current bucket range allowed. Also, the next sequential interrupt will signal an advancement to the next bucket. So HEXCITE follows the AFL-style bucketed hit count method but can achieve acceptable performance, since it has just one instrumentation location per loop. HEXCITE has evaluated against UnTracer [125], Retrowrite [41], and Dyninst [11] with 12 real-world binaries and gained better block and edge coverage and uncovered many known bugs.

The subsequent sections (Sections 4 and 5) will delve into specific methodologies that harness coverage information to overcome the challenges posed by limited source code access. Notably, we will highlight the performance upper bounds of the works referenced in this section to underscore the need for further discussions on these techniques. Through this progression, we aim to comprehensively explore the strategies that enhance fuzzing outcomes, contributing to a more nuanced understanding of effective fuzzing in diverse contexts.

4 DIRECTED FUZZING IN CONJUNCTION WITH STATIC ANALYSIS

Directed fuzzing with static source code analysis is a technique that incorporates the advantages of two testing techniques to determine potential software bugs and vulnerabilities. In directed fuzzing, the fuzzer generates inputs from the seed test cases. Some of them could be very specific to distinct functionalities of the target code rather than random inputs. Indeed, these input values generated are used to analyze the code and identify potential vulnerabilities concealed or missed—the vast generation of inputs aimed to improve the accuracy and effectiveness of static code analysis. Moreover, providing targeted inputs helps uncover deeply hidden code vulnerabilities.

The answer to RQ2(a) addresses directed fuzzing and enhancement possibilities while using fuzzing with static source code analysis. As mentioned earlier, directed fuzzing generates various input values conceived to target specific functionalities and components of the source code, unlike traditional fuzzing, which involves generating random inputs to test the target software. Directed fuzzing can enhance static code analysis by providing a targeted approach to testing such specific or sensitive components and functionalities that are likely to be vulnerable. Two types of fuzzing techniques that can be used in conjunction with static source code analysis are coverage-guided and mutation-based fuzzing.

Coverage-guided fuzzing helps generate inputs that maximize code coverage by targeting different parts of the code with different inputs. The predominance of coverage-guided fuzzing has opened a broad window to enhance the path coverage in fuzzing and identify new bugs and vulnerabilities [17]. Improvisation on such fuzzers has been incredibly well, and good to continue finding the basic blocks or paths uncovered in the source code examined. However, many state-of-the-art coverage-guided fuzzing methods become ineffective after a while [14, 17, 26, 52, 109, 140]. The significant difficulty experienced is covering the paths that are hard to trigger code with the seed inputs provided or the existing mutation strategies. Here, we must be tricky enough to elude ineffective mutation bytes and inputs that do not contribute to path coverage. Therefore, Identify the areas not well covered by existing tests and then use static analysis to identify potential

vulnerabilities using various inputs generated by the fuzzing technique, whereas mutation-based fuzzing helps make minor modifications to existing inputs to test sensitive components and functionalities of the code. Mutation-based grey-box fuzzers use evolutionary algorithms to prioritize the *interesting* test cases that uncover better path coverage [17, 109]. However, a concern to be raised here is regarding the mutation process of inputs. The mutated input generation technique is purely random, limiting the generation of *interesting* test cases. We have many general-purpose mutation-based fuzzers, such as Angora [26], Honggfuzz [66], QSYM [177], AFL [122], AFL++ [52], and so on, that have limitations in testing language processors. These fuzzers are unaware of the input format, and mutation happens randomly using mathematical or logical operations on the bits and bytes. Here such fuzzers will miss the ingenuity toward generating correct syntax-oriented inputs. Yet, fuzzers such as LangFuzz [78] and Superion [164] add value toward high mutation in the Abstract Syntax Tree or the IR code of the programming language, which guarantees the syntactic correctness but misses the semantic correctness. However, by modifying inputs, we can identify potential vulnerabilities that may not be detected otherwise. Then we use static analysis to identify potential vulnerabilities using inputs generated by fuzzing.

The usage of coverage-guided or mutation-based source code fuzzing with static code analysis is feasible. However, it requires careful consideration of several factors, and by answering this, we can address RQ2(b). One key factor is the selection of appropriate fuzzers, and static code analysis tools are essential to ensure that the results are accurate and reliable. We have various state-of-the-art coverage-guided and mutation-based fuzzers that help us integrate with static source code analysis to improve the identification of bugs and vulnerabilities. Fuzzers such as AFL [122], libFuzzer [109], go-fuzz for Go [42], cargo-fuzz for Rust [128], JQF for Java [131], jsfuzz [57], and jsfunfuzz for Javascript [88], pythonfuzz for Python [58], javafuzz for Java [56] that concentrates on a particular programming language. The research community is also open to customizing fuzzers concerning different programming languages to validate semantic correctness. We can also call it the language processor fuzzer, such as CSmith [175], which performs heavy analyses to generate valid C programs without undefined behaviors. C4 [172] is a compiler concurrency checker with a C4f fuzzer that adds random nontrivial control flow and redundant atomic actions to concurrent test cases. For Javascript engines, we have DIE [135], with two mutation strategies, structure and type preservation, using a lightweight static and dynamic analysis technique. FuzzIL [71] is another guided fuzzing approach for JavaScript interpreters. SQUIRREL [184] fuzzer identifies the data dependency of SQL to generate valid queries to test database management systems.

There are various static source code analysis tools in conjunction with fuzzing for static analysis. Tools such as AFLSmart [141] are designed to act as a coverage-guided fuzzer and for static analysis of C/C++ source code. CodeSonar [161] is a commercially viable static analysis tool that detects memory errors, null pointer deferences, dataflow errors, and many intriguing bugs. We can use it with fuzzing by generating various inputs for identifying these bugs. Frama-C [35], a static analysis framework for the C programming language that can detect potential security vulnerabilities, programming errors, and other issues in code. Frama-C analysis plugins can be used in conjunction with fuzzing to identify potential issues. We also have LLVM [102], which acts as a collection of modular and reusable compiler and toolchain technologies for static analysis of C/C++ code and includes several tools in conjunction with fuzzing, including Clang, a C/C++ compiler, and libFuzzer [109], a coverage-guided fuzzer. Another aspect is the time and resources required to create the input values used in the fuzzing tests. It could be time-consuming and resource intensive, particularly for mutation-based fuzzing, which involves modifying existing input values. Additionally, the effectiveness of coverage-guided or mutation-based fuzzing may be limited by the complexity of the target being tested.

To address RQ2(c), we have considered various latest source code fuzzers in Section 4.1. Those works answer the issues related to identifying bugs and getting better coverage. Furthermore, the benefits of using directed fuzzing with static code analysis comprise targeting sensitive and specific functionalities of the source code that are likely to be vulnerable. Besides, directed fuzzing can generate many input values to test different functionalities of the targeted based on the strategies rendered for each functionality. However, limitations such as time and resources required to create the input values are computationally expensive and intensive. Furthermore, directed fuzzing

is only efficacious for identifying known or suspected vulnerabilities and may need to be more effective for identifying new or unknown vulnerabilities.

Section 4.1 will discuss various recent source code fuzzers that address RQ2(a), (b), and (c), which claim to have betterness in identifying bugs, coverage of codes (block or edge), hit counts, and so on. We have detailed their methodology and showed specifically that RQ2(c) is better explainable using those techniques based on the user's need or fuzzer performance expected.

4.1 Recent Fuzzers Based on Directed Fuzzing with Static Analysis

ATTUZZ [187]. It highlights the importance of combining static source code analysis with fuzzing. It, in turn, aids in improving the effectiveness of unknown vulnerability detection in the source code. Likewise, it equips fuzzing to help generate better test cases that are more likely to trigger vulnerabilities. Later, static code analyses can identify critical variables and data structures that must be targeted during fuzzing. Furthermore, the work highlights the potential benefits of using dynamic program analysis with static analysis and fuzzing. The work also lists the importance of static analysis as a complementary technique to fuzzing, which can help identify critical code areas to target and generate more effective test cases. To achieve this, ATTUZZ focused on rewarding the efficient seeds contributing to path coverage and bug identification. Similarly, the mutation strategy evaluates, where the system continuously selects and mutates the seeds by utilizing more relevant bytes to explore previously uncovered paths. Indeed, such a strategy is expected to enhance fuzzing effectiveness, even as it may gradually deteriorate over time.

To contribute to such an effort, ATTUZZ used the concept of global source code analysis (a kind of lightweight dynamic analysis) with the help of a deep learning model with an attention mechanism in ML. The proposed method quantifies the reward of covering the basic block through this global analysis, and assigning the reward is based on the probability of covering uncovered branches and their count. The attention model helps determine the coverage of significant bytes, providing practical guidance on future mutations. They have, furthermore, trained the corresponding mutators by using the attention model to learn and update the fuzzing data periodically. ATTUZZ is written on top of AFL [122] by using the same techniques as AFL to generate inputs and record the seeds. Once the fuzzer gets stuck or loops through the same paths after a dedicated time, its model gets activated. It then adopts deep learning with the attention mechanism to train the model and predict whether a particular change in a significant byte during mutation can achieve any uncovered basic blocks. Nevertheless, it is well aware of the expenses due to many uncovered paths and trains a model on each seed input to find the significant byte. However, to address that challenge, an abstraction of the source code is created well in advance using a labeled discrete-time Markov chain. Later, they found the critical basic blocks and prepared separate fuzzing data. The newly created fuzzing data aims to get the heat maps of the seed file under different mutators to guide the selection of the more significant bytes in a seed file. ATTUZZ tested against many state-of-the-art coverage-guided fuzzers and detects new bugs in real-world programs such as mupdf, libjpeg, harfbuzz, libxml, and so on, and LAVA benchmarks [36].

AFLTeam [142]. Coverage grey-box fuzzing has received much attention in the research community of fuzzers as discussed above. The reason behind such a statement arose from the increase in path coverage, meaning it can uncover many unexplored or complex to trigger paths during the fuzzing process [13, 33, 109, 117, 122]. Still, exploration in coverage grey-box fuzzing has more concentration toward single-mode fuzzing. The importance of parallel-mode fuzzing is explored in various fuzzers, but the performance is not up to the mark [17, 103]. For example, consider the case of AFL's default parallel fuzzing mode (AFL-P) [122]. AFL-P employs collaborative parallel fuzzing that uses the shared seed corpus directory. Suppose we have three instances of AFL running

in parallel mode. Let us assume that fuzzing instance 1 found an *interesting* seed corpus. The term *interesting* means the mutated seed could find some unexplored path in the source code during the fuzzing process. At the same time, the other fuzzing instances 2 and 3 will also be periodically checking the shared seed corpus directory. The fuzzing instances search for new *interesting* inputs in the seed corpus directory. If the fuzzing instances 2 and 3 find the newly mutated *interesting* seed, then they will copy that to their seed corpus queue. Here, some issues will arise as follows:

- The dynamic information (favored/covered paths, new edges on, count coverage, etc.) supports single-mode fuzzing. Such data are not synchronized with parallel-mode or, say, not simultaneously synchronized with other fuzzing instances.
- Task conflicts due to the same seed corpus usage by different fuzzing instances.
- If using the same seed corpus, then fuzzing instances 2 and 3 visit the same paths explored or search for bugs in the program space.
- Using the same seed corpus produces identical test cases and causes deterioration in the performance of fuzzing processes.

Interestingly, the issues led to the importance of explicit task allocation strategies, dividing the tasks and assigning them to each fuzzing instance. With a vision of addressing the issues mentioned, AFLTeam is proposed and built on top of AFL [122]. Here, static analysis gets into conjunction while guiding the allocation of fuzzing tasks in a collaborative parallel fuzzing system. A framework for distributed fuzzing employs static analysis to partition the target program into smaller, more manageable code units. Multiple workers can then fuzz these units concurrently while identifying all potential code paths susceptible to bugs or vulnerabilities. Subsequently, these paths are subdivided and allocated to individual workers for fuzzing. We can have an even workload distribution by dividing the code into smaller units, and it reduces the risk of overloading worker nodes and slowing the overall fuzzing process. Again, it helps determine which input types should be used to fuzz each code unit. To attain this, a graph-partitioning task allocation mechanism produces an attribute call graph of the program, which gets fuzzed. Later, the graphs get divided into many subgraphs, and each subgraph will be assigned to each fuzzing instance. Each subgraph gets treated as a task, hence named Task Division. For example, each fuzzing instance will focus on tasks such as handling the file header, reading the data chunks, mutating the data chunks, and so on. Once a task is completed, the instance may merge with other tasks that are getting processed. Furthermore, if some tasks consist of many branches of functions that are complicated, then it gets divided into sub-tasks if necessary. Later, using a Task Dispatcher, they dispatch the completed tasks and initiate new fuzzing instances. All the fuzzing instances know the tasks assigned (branches of functions). Using a monitoring algorithm, track the statistical information such as the number of fuzzing instances, define the moment where to stop the instances, and request Task Division to assign new tasks. AFLTeam leverages graph-partitioning and search methods to improve the path coverage while using collaborative coverage grey-box parallel fuzzing and solving the issues present in the AFL's default parallel fuzzing mode. They achieved higher code coverage on benchmarks such as PNGImage, dJPEG, and discovered new zero-day bugs in FFmpeg and JasPer toolkits.

PATA [108]. As mentioned in Section 4, the mutated input generation technique is purely random, limiting the generation of *interesting* test cases. PATA leveraged the benefits of taint analysis. It is already well proven that taint analysis can observe the influencing input bytes and solve complex constraints in the source code. They have employed fuzzing in conjunction with static analysis to perform path-sensitive taint analysis. By analyzing the code in advance and identifying potential paths that can get tainted, the framework can generate test cases more likely to trigger vulnerabilities in those paths. For this, PATA utilized taint analysis to identify the relevant data sources to each path and then performed input generation for the fuzzing. Additionally, static analysis guides fuzzing by prioritizing the test cases more likely to trigger vulnerabilities.

In contrast, taint analysis has drawbacks such as over-tainting and under-tainting. These flaws can occur in fuzzing, which limits the performance of fuzzing. For example, consider using taint analysis in fuzzing as propagation based [26], leading to over-tainting. Then each input byte will taint using different labels and propagate these labels during the program's execution. Here the propagation will fail to understand the multiple occurrences of the same constraint variable, and hence it will visit the given constraint numerous times. Thus fuzzer needs clarification on when to change the repetitious labels. Furthermore, in such a case, all the input bytes may get marked as influencing bytes, and it will not make any progression in mutation. If we consider taint analysis using inference based [4, 59, 103, 177], then it may lead to under-tainting. Randomly carrying out byte mutations may alter the execution path and visit the same constraint multiple times. In the worst-case scenario, it might never encounter that constraint. Additionally, it may miss certain bytes, affecting specific occurrences because it captures the constraint value only once.

The importance of path-aware taint analysis is to overcome the above-mentioned drawbacks. The concept of path awareness will distinguish between multiple occurrences of the same constraint in an execution path. PATA takes care of path awareness during taint analysis to proceed for better mutation in fuzzing. It ensures that all the representative variables in the constraints are collected and monitors the value changes when the inputs are mutated. PATA mutates the influencing input bytes to solve constraints in an execution path. PATA was tested on the Google fuzzer-test-suite [69] and the LAVA benchmarks [36]. It outperformed state-of-the-art fuzzers such as Angora [26] and GREYONE [59] in finding the unique paths and coverage of basic blocks and discovered 40 new bugs with 12 bugs confirmed as CVEs and listed many new bugs on the Google fuzzer-test-suite.

POLYGLOT [28]. It neutralizes the discrepancy in syntax and semantics of programming languages and can fuzz language processors of nine programming languages. The framework employs fuzzing in conjunction with static analysis to perform semantic validation of the source code. Semantic validation involves analyzing the code to ensure that it conforms to the expected behavior of the language processor that is being tested. Hence, we analyze the syntax and structure of the code and the behavior of the language processor itself. By performing semantic validation, the framework can identify potential vulnerabilities and, later, generate more effective test cases and use them for fuzzing the target software. It is achieved along with semantic analysis, which helps identify the specific features and behavior of the language processor relevant to each test case. To balance the differences between each language, they have designed an IR to map the language-specific features related to the syntax and semantics. The IR generates a uniform format for any programming languages tested in this work. For that, it is a must to know the BNF grammar of the language picked up and translate the source code to the IR format with the support of the BNF format. The IR structure keeps track of the syntactic format of the source code; hence reverse translation to the source code is feasible. To understand the semantics, an annotation format is designed that describes the scope of functions, methods, and variables and their types and argument types. The description generated using this annotation format gets encoded into the IR's semantic properties and helps POLYGLOT repair the semantic error during translation. Once POLYGLOT captures the language-grammar-specific syntactic and semantic features using the IR, then mutation can be carried out using the generated test cases irrespective of the programming languages. To ensure the syntactic correctness of the generated test case, it is necessary to mutate the IR based on the types acquired for the IR grammar structure and later use it for the semantic validation of the unmutated section of the code. Therefore, the constrained mutator mutates the IR code to generate new IR code, which might contain semantic errors. The semantic validator then fixes the new IR code-generated semantic errors. Eventually, the fuzzer runs reverse-translated validated source code to identify bugs or vulnerabilities. POLYGLOT applied to different real-world programming languages such as C, C++, Javascript, R, PHP, SQL, LUA, SOLIDITY, and PASCAL. Furthermore, POLYGLOT is compared with AFL [122], QSYM [177], NAUTILUS [3], CSmith [175], and DIE [135] fuzzers.

5 FUZZING WITH SYMBOLIC AND CONCOLIC EXECUTION ENGINE

Symbolic and concolic execution are prevalent techniques used in fuzzing to explore more paths in the source code. We represent the program inputs and variables as symbolic expressions rather than concrete values. Later, we explore all possible paths using the provided set of symbolic inputs. In contrast, concolic execution in fuzzing combines both concrete and symbolic execution. In this approach, we begin code execution with concrete inputs to collect program behavior information. Subsequently, we generate symbolic inputs based on the gathered information, enabling us to explore new paths. It helps realize complex condition branches and to explore new branches. However, it is computationally expensive, since it explores all the possible paths could be time-consuming. Additionally, rendering concrete inputs demands specialized knowledge and expertise to generate satisfactory symbolic inputs. We have some specific fuzzers that use symbolic and concolic.

- QSYM [177], a state-of-the-art symbolic execution engine developed to improve the performance fuzzing by analyzing the application's binary code and generating a symbolic execution tree that represents all possible paths through the program. QSYM could handle complex input formats such as images, network protocols, and compressed archives. Therefore, QSYM is a versatile symbolic execution engine. However, it possesses high-computational overhead, since it highly augments the use of computational resources and limited compatibility, since it can run only binaries that can compile on x86 or x86-64 processors.
- AFLSmart [141] is an extension of the AFL [122] that contemplates a symbolic execution to explore new paths. However, it is computationally expensive, since it requires analyzing the inputs generated and thus yields symbolic inputs.
- In contrast, the KLEE [19] symbolic engine built on top of LLVM [102] lacks the models for system environments and sockets or multi-threading programs. KLEE [19] requires LLVM IR [102] code to work with, and we cannot use it directly on binaries. We have tools such as S2E [29], which works on binaries but possesses high computational overhead and requires more memory and CPU cycles to perform symbolic execution and exploration. Therefore, scaling up will be challenging. Another tool is the FuzzBALL [54, 55], a binary symbolic execution engine based on VINE and BitBlaze that works on the VEX-based IR of binaries. Nevertheless, using VEX to translate binary code into an IR is complex and computationally expensive. Besides, it lacks support to handle external libraries and system calls, limiting its ability to explore paths or identify vulnerabilities related to these interactions.
- SAGE [65] is also a symbolic execution engine, which uses a hybrid model consisting of dynamic taint analysis and symbolic execution to explore program paths based on symbolic inputs. Hence, it has increased computational complexity.

Though techniques such as symbolic and concolic execution are heavyweight or suffer the path exploration problem, it is well explored in the vision of hybrid fuzzing. One of the best examples in this category is Driller [155], which uses selective concolic execution. Driller is a hybrid vulnerability excavation tool that leverages fuzzing and selective concolic execution to help explore new paths and discover deeper bugs. Also, it explores the paths found as *interesting* and generates inputs for the conditional branches that the fuzzer cannot satisfy. For that, Driller leverages concolic execution, which reaches the deeper paths of a conditional branch and uses a feedback-driven process to mitigate the path explosion problem. Therefore, Driller helps achieve higher code coverage than any other symbolic and concolic engine that exists and maintains high scalability and speed. However, all these fuzzers suffer from path explosion, which occurs when the number of paths to explore during symbolic execution grows exponentially with the program's complexity. This can result in computational overhead and limit vulnerability identification effectiveness.

In this section, we use Driller as our testbed to address the research questions. To address RQ3(a), we must generate *interesting* input test cases that satisfy the complex branch condition checks. It uses a concolic execution engine such as angr.[1] The constraint-solving technique (Z3 solver based on SMT [38]) generates inputs to explore the state space of the target for better code coverage and to identify potential security vulnerabilities. Figure 3 shows the workflow of Driller [155] with AFL [122] and angr. AFL generates an initial seed and places it in the input directory. Each time a seed gets produced, its recording is saved in bitmap fuzz. During the execution of AFL, it gets on to a *stuck* phase, from where the AFL fails to create any *interesting* paths or diversions. At the same time, Driller can be invoked parallelly by providing the *stuck seed* and the current bitmap to the Driller. The Driller generator gets activated and starts its concolic execution. It initiates drilling

[1] https://angr.io

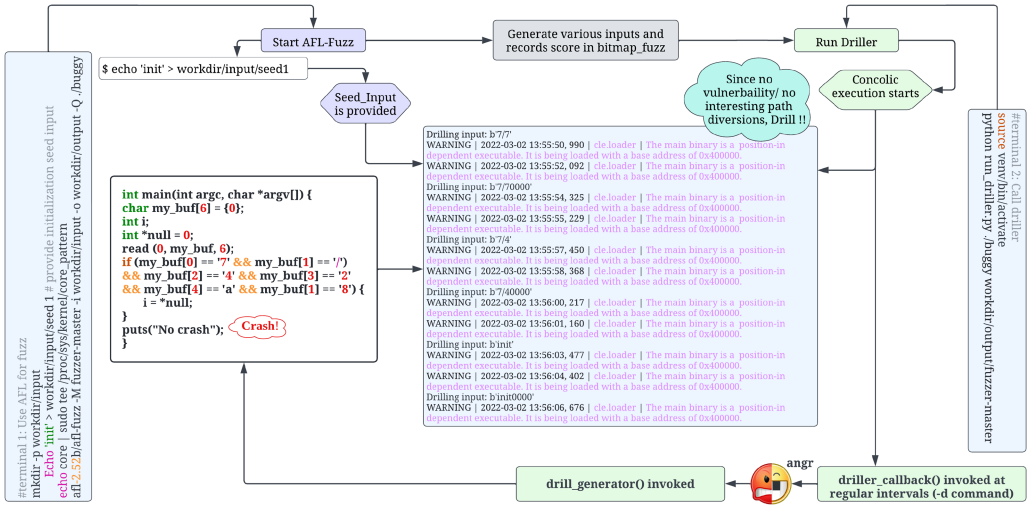

Fig. 3. Driller workflow with AFL and angr.

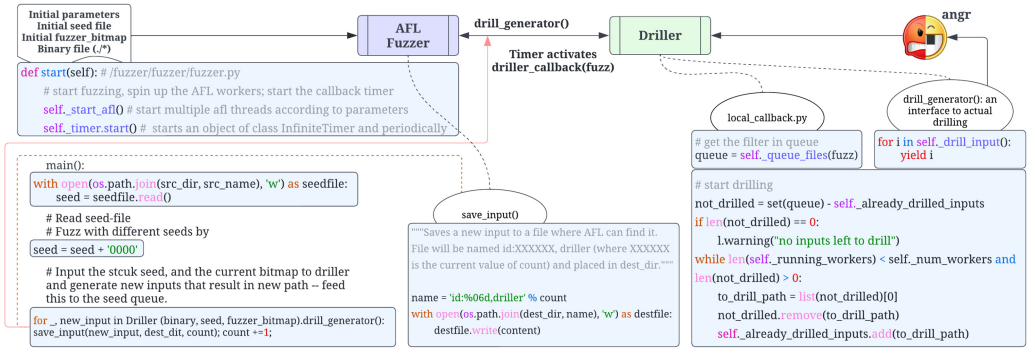

Fig. 4. Walkthrough the Driller source code.

with *b'7/7'* and then generates seeds by mutating and adding redundant bits. Seeds are saved in a seed queue. As we can see from the figure, the second drilling input is *b'7/70000'*. Driller used *-d* to drill repeatedly at regular intervals. We realize that to find the final crash in *b'7/42a8'*, Driller must receive an input seed *b'7/42a'*. Though it propagates back and forth for a long time, it will reach the expected input for identifying the crash.

The Driller [155] source code for executing these parallel processes is shown in Figure 4. The seed, binary executable, and bitmap files are provided as parameters to the AFL [122]. The *main()* function of AFL reads the seed file and generates seeds by execution of the statement *seed=seed+'0000'*. Whenever the fuzzer fails in traversing *interesting* or new paths, *drill_generator()* function is called along with *save_input()* function to save the seeds that the Driller would generate. Also, there is a *driller_callback()* mechanism at a regular time interval. The angr tool aids AFL in invoking Driller and analyzes the symbolic state of the path conditions. It creates preconstraints for the evaluation against *SimState* and compares the result with the current valid input state. angr's solver engine helps generate concrete values (concolic) and provides it as input for further drilling. To address RQ3(b) and (c), we have to apprehend the role of concolic execution and preconstraints.

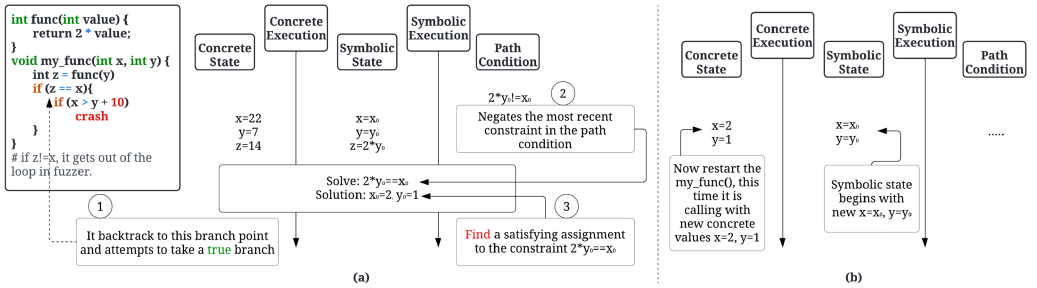

Fig. 5. Example scenario with concolic and symbolic execution.

Let us take an example[2] code in Figure 5(a). The key idea is to run the code concretely and record all the instruction traces achieved. Next, we use a preconstrainer to add constraints for symbolic execution in advance. The constraints added by the preconstrainer can get removed later. The preconstrainer method sets the constraints for the file and sets the state.posix.stdin (i.e., the input in symbolic execution) to its input (i.e., the test case passed to Driller). In this way, the test case is used as input to execute during subsequent execution, and the execution path of the test case itself is determined. For example, *generated_from_concrete=state.posix.stdin.load(0,state.posix.stdin.pos)*, which generates *x{generated_from_concrete}=22*, and *y{generated_from_concrete}=7*. These values get provided as a preconstrain for symbolic execution. In the Driller source code, it accepts as *s.preconstrainer.preconstrain_file(self.input, s.posix.stdin, True)*, where *s* is the state. These recorded concrete values get evaluated against the inputs obtained from the AFL [122] to find *interesting* paths. Then Driller symbolically analyzes the trace, chooses one of the branches, and negates it. This procedure will help generate new inputs, and we can restart fuzzing using the new inputs found. When Driller comes upon a conditional control flow transfer, it checks for any deviation in the path recorded from the fuzzer. It then expands its capability by inverting the condition that would result in discovering a new state transition. If it will, then Driller produces an example input that will drive execution through the new state transition instead of the original control flow, $x = 2$ and $y = 1$. After producing the input, Driller follows the matching path to find additional new state transitions. When new possible basic block transitions get discovered, Driller removes the preconstrainer. Thus, it solves for an input that would deviate into that state transition (Figure 5(b)).

As mentioned above, to solve the constraints in the example code in Figure 5 and to generate inputs, Driller [155] executes the statement *generated_from_concrete=state.posix.stdin.load(0,state.posix.stdin.pos)*. It will result in generating concrete values *x{generated_from_concrete}=22* and *y{generated_from_concrete}=7*. These values are saved in a preconstrain file called *SimfileStream* and further provided for symbolic execution. Driller's concrete results get evaluated against the inputs obtained from the AFL [122]. As mentioned earlier, Driller extends its capability by negating the most recent path condition and drills to find an input that will drive execution through the new state transition instead of the original control flow. Driller escalates its efficiency in solving constraints by coupling angr into it. If any new *interesting* path or new basic block transitions, then it gets recorded in the bitmap and removes the preconstrain. Figure 6 illustrates the storage of preconstrain to SimFileStream and shows the entry update in AFL-bitmap.

5.1 Recent Fuzzers Based on Symbolic and Concolic Execution Engines

CONFETTI [100]. This is a tool that uses concolic execution and taint tracking to generate complex inputs and detect hidden bugs in program logic through parametric fuzzing. In the traditional

[2] https://www.cis.upenn.edu/~mhnaik/edu/cis700/lessons/symbolic_execution.pdf

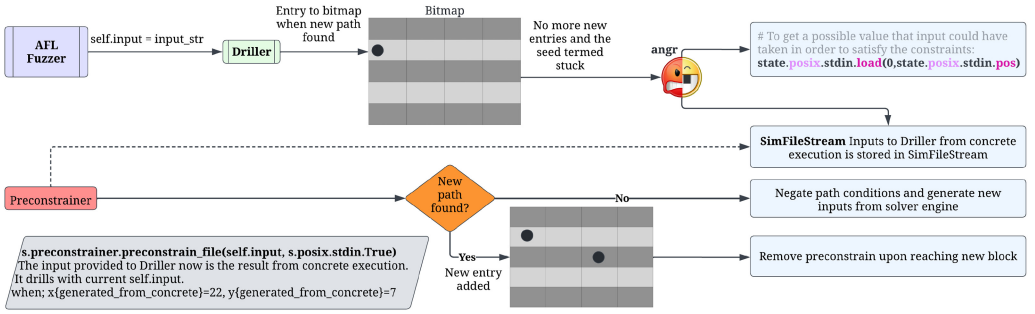

Fig. 6. Driller finding new entries and adding to bitmap.

fuzzing approach, we integrate white-box guidance with fuzzing using targeted hints, which directs the fuzzer to place specific bytes at specific locations in specific inputs. CONFETTI also uses targeted hints but introduces the concept of global hints, which help overcome the limitations of dynamic taint tracking. While CONFETTI is not tied to any particular programming language, it is implemented in Java. It can target applications written in languages that target the JVM, such as Java, Scala, Kotlin, Groovy, and Clojure. CONFETTI claims it is more effective than any other fuzzing approach, including a state-of-the-art grey-box Java fuzzer, covering more branches and finding more bugs. CONFETTI fuzzed several open source projects, including Apache Ant, BCEL and Maven, Google's Closure Compiler, and Mozilla's Rhino engine. However, the performance overhead occurring in CONFETTI feels the heaviness of concolic execution and taint tracking—also, the high-performance overhead limits its usage. The increased complexity due to global hints makes it less attractive. It adds complexity to the fuzzing process and makes it harder to integrate or maintain it with traditional fuzzers. Moreover, the limited applicability of CONFETTI limits its target that uses JVM. CONFETTI is still a research prototype, meaning it must be thoroughly tested or optimized for all scenarios. It may also need some of the features and stability of more established fuzzing tools.

FUZZOLIC [15]. It improved the efficiency of hybrid fuzzing by expediting different phases performed by concolic executors. FUZZOLIC, the concolic framework, is built on top of the binary translator QEMU [7]. The framework enhanced the emulation performance and versatility compared to the state-of-the-art binary concolic executor QSYM [177]. On the reasoning side, an approximate solver is introduced, FUZZY-SAT, which tests the satisfiability of symbolic queries generated by concolic engines without relying on expensive SMT solvers [38]. Instead, FUZZY-SAT use fuzzing domain and performs informed mutations on the expressions in a symbolic query to generate *interesting* inputs. Consequently, the approximate solver replaces classic SMT solvers in hybrid fuzzing. To showcase the potential of FUZZY-SAT, they have integrated it into two binary-based concolic frameworks, FUZZOLIC and QSYM, and into the source-based concolic executor SYMCC [143]. FUZZOLIC was tested on 12 applications and LAVA-M benchmarks [36] and identified new bugs.

6 INTERFACE AND ENVIRONMENT ISSUES

The runtime dependencies of fuzzing targets consist of interface and environment dependencies. For example, a device driver or firmware acts as an interface between the application and hardware and successively has access to the user-space data. Direct access to a user-space pointer can lead to incorrect behavior. It depends on the architecture, and the user-space pointer may not be valid or mapped to environment (kernel) space. Direct access can also point to a kernel oops, where the user-mode pointer can refer to a non-resident memory area, or it may lead to security

issues. Therefore directly accessing the user-space data is inconceivable; hence, the interface's and environment's issues should be attended to with utmost priority and made impeccable.

> The number of IoT devices is increasing exponentially nowadays. However, many security issues arose along with the increase in interfaces used in IoT devices. Recent botnet attack Mirai [94] is one example of accessing the live CCTV camera footage. The FDA had confirmed that medical devices used for the cardio examination could be hacked, and risk lead to the recall of the pacemakers [53]. Therefore, we can undoubtedly say that IoT devices are vulnerable to attacks, since they communicate with the outside world through the Internet. Consequently, the attack surface becomes more extensive than the traditional devices. It is necessary to have an assisting mechanism that tests the firmware code in detail to identify its flaws. Realizing the importance of coverage-guided feedback fuzzing on critical system applications has got enough attention in the research community. In that case, we can indeed question the security related to the environment, such as a kernel. For example, the Linux kernel consists of around 28M LOC. The Linux coding team uses the handwritten test suite called the Linux-test-project [16] to identify and eliminate the bugs in the kernel. However, manual elimination of bugs on such a large-sized code is nearly impossible due to the rapid increase in the code size and the complexity possessed in the code. Furthermore, any bugs present in the kernel code of an operating system will lead to severe security breaches, such as the blue screen of death, privilege escalation, information disclosure, and so on. Therefore, finding bugs in the kernel code is intense, since most fuzzers fail to do so. When we analyze the reason for such failures, we can undoubtedly say that finding kernel code bugs is coupled with understanding *syscall* types and their dependencies.

The answer to RQ4(a) is yes, and we can address the runtime interface and environment dependencies of fuzzing targets and find the bugs and vulnerabilities present in them. Let us consider firmware as the interface and kernel as the environment. For example, a race condition can occur if the code has multiple concurrently executing processes that try to access shared data, leading to a circular lock due to synchronization issues. Such a code behavior can direct to an unresponsive kernel and result in a deadlock situation. It will trigger a bug in the code, which could be a DoS or an escalation attack due to the priority assigned for the access. These bugs are prevalent in kernel code and easily identified using kernel fuzzing. However, as we say in RQ4(b), we face many challenges, and it is inevitable. We cannot use state-of-the-art fuzzing software such as AFL [122], libFuzzer [109], Honggfuzz [66], and so on, since it faces some challenges while testing the interfaces in IoT devices' firmware. The main reason concerns the generation of input test cases. The message format requirements of IoT devices vary much from standard desktop applications. Therefore, the input test cases will get rejected at a very early stage of fuzzing, since it fails at the input validation and sanitizations stage. In such a case, the fuzzing process cannot even enter the functionality stage of IoT device workflow. Also, it poses a challenge to insert the instrumentation code, since the firmware needs the support of the hardware. Thus, it is impossible to obtain coverage information to evaluate the performance of the fuzzer. Furthermore, it is challenging to use traditional grammar-based fuzzers such as GANFuzz [79], Skyfire [163], LangFuzz [78], Gramatron [154], GramFuzz [73], and so on, on IoT devices firmware or kernels. It needs many learning materials to support the extraction of grammar rules from the available firmware or kernel material. It can only support the available/known grammar format depending on the learning materials collected, and in the case of IoT device firmware or kernels, it is a complex task.

> In contrast, in environment fuzzing (for example, kernel fuzzing), we require some adaptions in the fuzzer executions. Let us take the example of the state-of-the-art fuzzer AFL [122]. AFL has to perform fuzzing, adapting the features of QEMU [7]. QEMU can emulate a whole machine in software without hardware virtualization support, and QEMU can also provide userspace API virtualization for Linux and BSD kernel interfaces. Applying the QEMU patch, AFL++ [52] and TriforceAFL [87] have performed coverage-feedback fuzzing on the binary that QEMU can run. The patch correspondingly managed to have an uplift from the user space to the kernel space, and hence it runs in the privileged mode and has direct access to the machine hardware support. The combination of QEMU/AFL++ and QEMU/TriforceAFL has found several critical bugs in the Linux kernel. The agents communicate with the virtual machine, check the messages received from the fuzzer, decode them to the system calls and make it interactive with the kernel or direct generation of system calls with the initial parameter values also helps to get the test cases. The *interesting* test cases get saved for another mutation from the execution trace.

While fuzzing the interface of an IoT device, many researchers have quoted the challenges as shown below [21, 32, 50, 74, 93].

(1) Fuzzing a firmware should be more transparent, since cohesion exists in the model's peripheral behavior and the driver code [50, 74, 93]. Essentiality lies here to leverage the existing debugging tools of the embedded software development and thus to construct a suitable fuzzing framework that shows the dependency on hardware and IoT firmware. For example, an emulation environment may yield the highest throughput, since IoT devices' processing is slower than desktop environments. The real-time IoT device performance is far beyond the emulation environment compared with a complete system emulation.
(2) The unavailability of obtaining and emulating IoT firmware makes fuzzing a firmware arduous [21, 32, 50]. Therefore, we experience *restricted access to the firmware* $\propto$ *retrieving the IoT device's internal execution trace* to influence the fuzzing process.
(3) In terms of fuzzing, we need a feedback-driven approach, since we miss the feedback from the IoT device and, therefore, optimize the fuzzing seed generation process [21, 50, 74]. Thus, retrieving the execution trace from the IoT device is not possible, and hence it is very challenging to fuzz the IoT device firmware using network communication.
(4) The diversified message formats in IoT device firmware are used to exchange network messages [21, 50, 74, 93]. IoT devices follow the strict grammatical syntax for communication messages and use forms such as JSON, string, SOAP, key-value pairs, and custom bytes. Therefore, IoT fuzzing software has to send random inputs or random formats of messages to the target IoT device, since it does not know the type of communication message. The random mutation strategy will break the expected syntax rule of the input format, and inputs will get rejected at the syntax validation stage.
(5) Therefore, the mutation strategy becomes expensive, time-consuming, and potentially misses the relevant ones. Furthermore, IoT devices possess extra elements in the communication messages such as nonce, timestamps, signatures, and built-in module formats. These additional elements in messages increase the randomness and hence lose the effectiveness of mutations.

In environment fuzzing, we have fuzzers such as Triforce Linux Syscall Fuzzer [86], Google's Syzkaller [43], Moonshine [133], SYZVEGAS [162], and StateFuzz [183]. All of them are kernel code testing fuzzers based on coverage-guided feedback. Despite some drawbacks, it is feasible to use coverage-guided feedback-based fuzzers to track the kernel states more extensively than the non-coverage-guided fuzzers. Triforce Linux Syscall Fuzzer decoded the inputs from the fuzzer and communicated with the kernel. Syzkaller tracks the system call description and arranges it in a sequence based on the appearance of system calls in the kernel code. Hence the sequence of system call invocation is tracked and used for feedback analysis to refine the seed corpus in the upcoming iterations. In contrast, Moonshine tried to extract the critical system calls based on the relevance of those mentioned in the handwritten test suite called the Linux-test project. Then Moonshine guessed the applicability of system calls' read-write dependencies. Using this assumption, Moonshine generated the test cases and further mutated them to create *interesting* ones.

Indeed, recently there have been so many amendments to Google's Syzkaller [43] by leveraging its unsupervised coverage-guided expertise to fuzz the Linux kernel.

(1) SYZVEGAS [162] is a *dynamic and adaptive fuzzer* designed to improve kernel fuzzing efficiency. Traditional fuzzers rely on fine-tuned and hard-coded parameters, limiting their adaptability to different environments and targets. SYZVEGAS addresses this issue by automatically adjusting two critical decision points in Syzkaller, namely task selection and seed selection, using Multi-Armed Bandit algorithms and a novel reward assessment model.
(2) StateFuzz [183] addresses limitations by proposing a *state-aware fuzzing* solution. StateFuzz models program states using state variables and employs static analysis to recognize such variables. Target programs are instrumented to track the values of these variables and infer program state transitions at runtime. State information is then used to prioritize test cases that can trigger new states, and a three-dimensional feedback mechanism fine-tunes the evolutionary direction of coverage-guided fuzzers. The objective of SYZVEGAS is to enhance kernel fuzzing efficiency and bug detection by learning effective strategies over time with reinforcement learning, whereas StateFuzz discovers bugs and vulnerabilities in Linux drivers by realizing the impact of system call inputs on the driver's state.
(3) Furthermore, we have PrIntFuzz [115], which is an engaging, efficient, and *universal fuzzing framework* designed to fuzz the Linux driver code and extend the support to test IoT firmware, e.g., simulating peripherals of IoT devices and rehosting the firmware in an emulator (e.g., QEMU) that can test the overlooked driver code, including the PRobing code and INTerrupt handlers. It addresses the primary challenges of fuzzing IoT device drivers, which stem from limited hardware support and the black-box nature of IoT software. PrIntFuzz extends DIFUZE [34] to work on the latest kernel and collects system calls from driver interfaces. Thus, it comprises different methods like user interactions (system calls), interrupt injection, and data injection. By orchestrating these methods in a specific order, it thoroughly explores the attack surface of Linux drivers and the drivers in IoT devices. This approach helps uncover vulnerabilities and potential security issues hidden in the black-box.

However, none have discussed the influence relation between the system calls. It has yet to contribute to analyzing the influence of one system call with the execution behavior of the subsequent system calls while generating and mutating the inputs. All the concerns mentioned above by various researchers have been addressed with the implementation of HEALER [156]. They have analyzed the influence relations between the subsequent system calls for generating *interesting*

test cases. Sections 6.1 and 6.2 discusses recent firmware and kernel fuzzers that address RQ4(a) and (b).

6.1 Recent Firmware Fuzzers

μAFL [105]. It is a feedback-driven fuzzing on ARM-based MCU devices' firmware to locate bugs in the low-level peripheral drivers. The development of such a fuzzing solution is in the light of vulnerability identification in ESP8266 and ESP32 communication, WiFi co-processors, and FreeRTOS TCP/IP stack leading to remote code execution and stealing private data. μAFL gave prominence to the concern of rehosting, that is, solutions that expect a shift in their infrastructure services that cannot model the peripheral behavior and thus cannot fuzz the driver code. Existing fuzzing solutions face challenges when applied to embedded device firmware. Most of them concentrated on emulation-based rehosting methods, but it will become inaccurate when modeling the diverse peripheral behavior. Recent works, such as Jetset [93], PRETENDER [74], μEmu [185], DICE [121], P^2IM [50], Laelaps [21], HALucinator [32], Ninja [129], USBFuzz [139], and Avatar [178], depend on symbolic execution, pattern matching, and ML to learn embedded firmware's extensive peripheral model behavior. However, inaccuracy exists on rehosting-based solutions, since we cannot boot firmware that possesses peripherals such as USB or complex hardware devices.

This discrepancy is due to generating different execution traces on an emulator than booting it on an actual device. It occurs when emulators use hardware-independent code related to device-firmware passes through the booting process. μAFL runs the firmware directly on the target device. Therefore, it supports a full-stack testing environment that assures high fidelity and overcomes the issues of a rehosting-based solution. A fuzzing manager on the user's system will synchronize with fuzzing using a debug dongle. It acts as an intermediator between the target device and the system where the fuzzing manager exists. The fuzzing manager retrieves all the execution traces and decouples the execution engine from AFL [122]. Thus, it uses the target-embedded board to execute the test cases. The seed test cases are submitted to the target device board during execution through the fuzzing manager, pulling all the code coverage and crash information.

Furthermore, μAFL leverages Embedded Trace Macrocell [2], which transparently traces every executed instruction and provides unmatched insight into the target board's activities. Hence, μAFL is free from code instrumentation, which helps μAFL to be free from rewriting the target device binary. Evaluation of μAFL conducted on NXP and a STMicroelectronics board identified three zero-day bugs and eight CVE assigned bugs.

SNIPUZZ [51]. Existing IoT fuzzing methods rely heavily on obtaining the firmware of the IoT device. Irrespective of whether we use static or dynamic analysis, we must have the IoT firmware, but it is challenging to obtain. In IoT device workflow, once it receives the input, it will send it to the sanitizer to perform input validation and then forward it to the function switch, which triggers the corresponding functionalities. Then the related functions will get evoked from the replier module, which will send the status of the execution back to the caller. SNIPUZZ acts as a client that communicates with the IoT devices in real time and deduces the message snippets (consecutive bytes). These deduced message snippets mutate to produce more input test cases based on the responses from the IoT device. In SNIPUZZ, the initial input seeds are collected from the IoT device's API documentation and their official website. Later, they mutate the seed byte-by-byte, send it to messenger, and communicate with the IoT device to collect the responses.

Upon the reception of each new response, categorization has been performed. Therefore, the inputs that make interesting responses have been added to the seed pool. SNIPUZZ use the response from the IoT device as feedback to guide the fuzzing process. However, it is impossible to directly extract the relationship between the firmware execution and the response messages. Hence, if two

inputs get two different response messages, then the information flow happens to two different firmware execution paths. The randomness in the communication messages is due to the tokens or timestamps. It shows that the response messages get varied in their presentation. Therefore, it relies on similarity algorithms to circumvent the randomness in the response messages. They calculate the similarity score based on the added distance, and then later, it gets normalized.

> Dealing with numerous message formats in the IoT device responses, all input messages are considered byte by byte and mutated byte by byte. Then the snippets are merged according to the responses. Concurrently, mutations on two different byte positions receive the same response. In that case, the two-byte positions may be highly likely related to the same functionality in the firmware. Therefore, those consecutive bytes with the same response will merge into one snippet. Thus, each snippet in SNIPUZZ is considered a block of consecutive bytes, and this mutation strategy based on snippets will help narrow down the search space to alter the probe messages. If two responses are semantically identical, then they will be classified into one category, and hierarchical clustering will convert the response message into feature vectors and track the features from the responses.

SNIPUZZ is compared with four state-of-the-art IoT fuzzers, such as Doona [48], IOTFUZZER [25], BooFuzz [95], and NEMESYS [98], and SNIPUZZ discovered many categories that exposed the most number of bugs, including five zero-day bugs.

6.2 Recent Kernel Code Fuzzers

SyzScope [189]. A known fact is that fuzzers can tirelessly discover bugs and vulnerabilities in an application. However, coverage-guided fuzzing explores more code blocks and paths, but negligence exists to analyze the security impact of the bugs found. For example, oss-fuzz software [70] continuously fuzzes the user-space applications and reports new bugs in day-to-day software [43, 70]. Although we could find more bugs, the challenge is the high rate of bug discovery compared to the rate of bug fixes. SyzScope addresses such a concern profoundly and uses the bug report as an input to build, extract, and distill out the kernel, PoC, and system cal bugs separately. It provide these bugs as targets to a kernel fuzzer to identify the read/write impacts of each PoCs using bug analysis and then performs taint analysis to verify the bugs. Later, with the help of symbolic execution, it explores the paths to each bug, which has a profound impact and validates it to produce the new bug report based on the predicted impact. Thus, it evaluates the impacts of the bugs identified and facilitates prioritized bug fixing with the help of static analysis and symbolic execution. For example, the UAF and heap OOB bugs may lead only to a read-primitive. The bugs belonging to the warning, info, or GPF also got classified in the low-risk impact category.

HEALER [156]. The concerns mentioned in RQ4(b) have been addressed more promptly in implementing HEALER. It analyzes the influence relations between the subsequent system calls for generating *interesting* test cases. An example of the influence relationship between system calls is in the case of the system call *bind* and the subsequent system call *listen*. We realize that '*bind*' assigns the address to the socket, and '*listen*' is employed to await a connection on that socket. If '*listen*' terminates prematurely with an error, it signifies an improper bind socket connection due to the absence of the initial '*bind*' system call. This issue arises because the execution of one system call can alter the execution path of a subsequent system call. Therefore, an influence relation learning algorithm assesses the relationships between system calls within the kernel code. This awareness is injected into the fuzzer to get an acquaintance on which a system call is present in the execution path of another call. A static routine algorithm analyzes the relationship between the system calls based on the initial input parameter types. Also, it checks the return types associated with the system calls. Furthermore, a dynamic-routine check algorithm finds the influence relations between the system calls. For that, it generates a minimized set of system call sequences and then runs the fuzzer to get feedback from each system call. The sole purpose of this minimization is to discard the system calls that do not contribute to generating *interesting* test cases. The execution trace is stored and used as feedback for further iterations to evaluate the influence

relation between the system calls. The dynamic-routine learning continuously updated the influence relations table with the information not captured in the system call descriptions.

HEALER generates *interesting* test cases that lean toward new bug discovery in the branch where system calls are present. Later, HEALER uses the information regarding the influence relations to guide the system call sequence mutation and generation. A guided call selection algorithm decides whether to use the influence relation table and understand which calls to select based on information in the relations table. HEALER's branch coverage shows an outstanding improvement with an effective speedup compared to the state-of-the-art kernel fuzzers Syzkaller [43] and Moonshine [133]. Furthermore, it resulted in identifying 33 new bugs in the kernel code.

NTFUZZ [31]. The dependency between the system calls and nested behavior precludes the fuzzers from generating meaningful test cases.

> To an extent, Linux kernel fuzzers such as SyzScope [189], HEALER [156], kAFL [150], pe-AFL [112], Triforce AFL [87], Triforce Linux Syscall [86], Google's Syzkaller [43], and Moonshine [133] are doing a good job of finding bugs from the kernel code. Nevertheless, the stringency will increase if the kernel is a Windows kernel, and none of the kernel fuzzers mentioned earlier are suitable for fuzzing the Windows kernel code. An exception is Google's Syzkaller, which has limited support for fuzzing the Windows kernel. However, Syzkaller can perform only on Windows API functions but not on the system calls. This is because of the private and the undocumented nature of the system calls in Windows.

NTFUZZ used a static binary analyzer that deduces the details about the Windows systems calls present in the "system binaries" at a large scale. "System binaries" represents the binaries that have documentation about the Windows API functions, and their implementation is present in built-in core system libraries (DLL files) such as *ntdll.dll*, *kernel32.dll*, *kernelbase.dll*, *win32u.dll*, *gdi32.dll*, *gdi32full.dll*, *user32.dll*, and *dxcore.dll*. "System binaries" get converted to IR code and create the CFGs and parse all the Windows API functions based on the type information provided. They have used static analyzer to find and analyze the *syscall* types by tracking their reachability to the register and memory states. Therefore, the static analyzer analyzes the binaries such as *kernel32.dll*, *kernelbase.dll*, *ntdll.dll*, and so on, and invokes the *syscalls*. Then it deduces the information about the system call that includes its argument types. It bridges the information gap between documented and undocumented functions in Windows, known Kernel API functions, and the system calls by propagating the information between the interfaces. NTFUZZ is compared with IOCTL-Fuzzer [44] and NTCall64 fuzzer [77]. However, both the fuzzers had required code modification, since the development of NTFUZZ is to fuzz Windows 10. There is a significant impact over finding bugs while fuzzing Windows 10 system binaries with NTFUZZ.

HFL [97]. As discussed in the works mentioned above, kernel fuzzing is challenging, since we must consider the indirect control flow transfers during *syscalls*, internal state matching via *syscall*, and tracking the nested arguments as and when *syscall* invocation.

> We know that random fuzzing gets stuck when branch conditions are complex [140]—the savior in such a situation incorporates symbolic execution [9, 20]. Likewise, symbolic executions suffer state explosion, since it has to explore both taken and not taken branch branches, and the complexity grows exponentially [19, 29, 155]. It will lead to a state explosion issue. In such a case, random fuzzing will act as a savior by pinpointing the fuzzer to follow a specific exploration path. Thus both of these methods stay hand-to-hand to mitigate the concerns suffered. Therefore, HFL, i.e., Hybrid Fuzzing, combines both random fuzzing and symbolic execution to assist kernel fuzzing. It is apparent that fuzzing kernels are not similar to fuzzing an application using the hybrid fuzzing technique. Some existing kernel fuzzers, such as RAZZER [85], Digtool [134], and kAFL [150], ignored the challenges mentioned above and used random fuzzing and symbolic execution as distinct methods. Using static analysis, kernel fuzzers such as DIFUZE [34] and Moonshine [133] have only handled some challenges.

To tackle indirect control flows in the Linux kernel (for example, polymorphism due to many devices and feature support, kernel relies on a function pointer table), HFL first converts indirect to direct control flows. Here we may argue that random fuzzing can understand indirect control transfers. However, random fuzzing will only understand the indexing associated with the function pointer table if we are trying to fuzz a kernel with many indirect control transfers to support

polymorphism. For the conversion, HFL uses an offline translator, which will not alter the semantics of conditional branches, i.e., the underlying code blocks are reachable through direct control flows. It tracks the system call parameters propagation, since the translator is not altering the index variable related to a particular *syscall* parameter in the function pointer table. Consequently, the translator is performing the inter-procedural dataflow analysis intact. HFL also performs a branch transformation by inserting a conditional branch jumping to a corresponding function pointer, ensuring it will never miss tracking a branch. HFL outperformed the kernel fuzzers, such as Moonshine [133], Syzkaller [43], TriforceAFL [87], kAFL [150], and S2E [29], by achieving a higher coverage and discovered 24 new bugs.

KRACE [174]. It is arduous to realize the influence of multiple thread synchronization while accessing the shared memory due to the high concurrency nature of kernel file system design. It is prominent in the data race scenario when multiple threads try to access the shared memory location without proper synchronization. However, detecting data race is difficult, since it shows fewer thread interleavings, triggering a complicated memory crash in kernel file systems. KRACE concentrates on data race or thread interleavings that can happen due to multi-threaded system calls. KRACE design proposed alias instruction pair coverage, focusing on the concurrency domain. It used RAZZER's [85] conventional branch coverage approach to explore the code sequentially.

> Developers commonly use the stress test to detect data race situations in source code [16]. Yet the increased complexity in kernel code due to its intensive workloads triggers distinctive thread interleavings, and the number of data races eventually increases. Fuzzer, like RAZZER [85], has proved the prominence of kernel code fuzzing and reported that data race in a kernel is an erroneous bug, which critically affects the reliability and security of the underlying system. RAZZER relies on static analysis and deterministic thread interleaving techniques to detect data race scenarios but is concerned about the coverage issues rather than the data race due to concurrency. The sequential aspect of source code was treated well in RAZZER but focused on something other than the concurrent execution aspect in source code. That means it focused more on single-thread system call executions than the multi-threaded system call sequences, which may trigger many thread interleavings.

To comprehend the concurrent execution scenarios, it tracked all pairs of memory instructions, knowing that threads may make interleavings in one another during execution. The sole responsibility of alias instruction pair coverage was to count and track all covered interleaving locations. KRACE retains the details about the shared memory access and ensures that at least one thread is a memory write access in a pair of threads. According to the KRACE design, scheduling the threads is significant, since it is pertinent to explore unexplored coverage paths. During compilation, KRACE adds annotations to the kernel code. An LLVM instrumentation pass [102] and KRACE library compilations extract the coverage path's details and track the runtime. Furthermore, KRACE generates a set of multi-threaded seeds and input test cases, executed in a QEMU emulator [7] and virtualizer environment to augment fuzzing and detect the data races scenarios in kernel file system code. KRACE has uncovered 23 unexplored data races bugs in ext4/ btrfs file systems.

7 INSTRUMENTATION ERRORS

As discussed in all the recent works, the researchers have tried either to improve fuzzing coverage or the performance of fuzzers to identify new bugs. There is always a need for software like fuzzers, since bugs are inevitable in manually written programs. Those bugs or vulnerabilities in a software source code can open up the attack surface for attackers.

7.1 Recent Fuzzer to Deal with Instrumentation Errors

InstruGuard [111]. This is a stand-alone tool that finds and fixes instrumentation errors during fuzzing. To address RQ5(a), we know that instrumentation or coverage feedback is paramount for grey-box fuzzers. Unfortunately, existing works believe that the instrumentation for getting coverage feedback is thoroughgoing and precise. InstruGuard proved that that is a misconception. The developers must validate their belief in their instrumentation completeness and

accuracy. To answer RQ5(b), a thorough understanding of fuzzing instrumentation completeness and accuracy is required. However, it is evident that if the instrumentation code possesses some errors, then it will reflect in the fuzzed program and lead to unawareness of the hard-to-trigger coverage paths and missing the bugs hidden in those blocks. We must have some novel methods to address missed/redundant instrumentation locations. In InstruGuard, if instrumentation misses some necessary instrumentation locations, then the fuzzer will not discover the missing part and bypass the chance of finding vulnerabilities in the lost fragment of code. Second, if redundant instrumentation happens on a particular block, then the path coverage depth will confuse the fuzzer depending on its execution time and depth. For example, in AFL [122], the bitmap size is 64 KB, where it stores the coverage feedback information. If the path depths increase due to the redundant instrumentation location, then bitmap collision occurs. For addressing RQ5(c), let us take the scenarios of compiler-level instrumentation. The IR code is optimized by default during such instrumentations (for example, LLVM [102] compiler-level instrumentation). Such optimizations usually merge some basic blocks. For instance, consider the below-mentioned cases:

afl-clang-fast: Instrument the code only at the beginning of a basic block. Due to the merge, the instrumentation misses some chances of code instrumentation at some basic blocks. It leads to missed bugs or crashes at that location, affecting the coverage feedback due to the missed or redundant instrumentation at that basic block. If it is assembly-level instrumentation, then the binary created must have a modified assembly-level basic block and miss the instrumentation due to the error. The assembly-level instrumentation (*afl-gcc* or *afl-clang*) rule in AFL [122] deals with function/branch destination labels or conditional jump instructions that adds instrumentation. Furthermore, AFL adds instrumentation at the actual assembly code of a program (*text* section) or leaves the instructions below *.p2align* to reduce unnecessary instrumentation and affect the coverage-feedback. From all these, we can address RQ5(d) that instrumentation errors turn to faulty coverage feedback and damage the fuzzing process.

InstruGuard disassembles the given program to the assembly code, checks each basic block, and instruments their code if any missing or redundant instrumentation in the basic block is compared with the original IR or assembly code. InstruGaurd rewrote the binary by adding or deleting the required code instead of the vanilla binary created by the original instrumentation. The experimentation used six coverage-based grey-box and white-box state-of-the-art fuzzers.

There is a significant difference in the instrumentation error rate and locations in the above-mentioned fuzzers. AFL [122] produces the least number of instrumentation errors in compiler and assembly-level instrumentation, and most errors are related to missing instrumentation. MemLock [170] modifies AFL code to acquire memory information, leading to more redundant instrumentation errors. Angora [26] also edits the original code by splitting the basic blocks with conditional statements and generating two basic blocks. This strategy can reduce the overhead while solving the constraints. Still, it causes many instrumentation errors, since only one basic block will collect the instructions of the conditional statement. Hence redundancy and missing instrumentation errors are increased in Angora [26]. AFL++ [52] employs a different strategy, such as adding new basic blocks to the original code while compiling the code. The newly added block gets loaded with *mov* and *jmp* instructions to handle loading edge coverage identification from its global memory. This attempt will cause many redundancy instrumentation code sequences in many locations, leading to unnecessary instrumentation errors and, later, affecting the coverage.

REZZAN [8] and FIFUZZ [90]. Explicitly seeking, InstruGaurd [111] is the only fuzzer designed specifically to check for instrumentation errors. However, fuzzers such as REZZAN and FIFUZZ (applies static analysis) check for memory errors and error-handling code by combining memory error sanitizer with fuzz testing. It detects "silent" memory errors that often go unnoticed by the program, as it does not always cause immediate crashes or other apparent failures. Instead, it leads to subtle, hard-to-detect bugs that may only manifest themselves under certain conditions or after a significant amount of time has passed. For example, memory errors such as out-of-bounds array access, null pointer dereference, use-after-free, uninitialized memory access, and memory leaks

go undetected in most cases. The memory error sanitizer design in REZZAN is optimized for fork-mode grey-box fuzzing. We also have FIFUZZ, which tests the error-handling code and detects bugs. FIFUZZ applies static analysis on code to pinpoint potential error sites. Users can select those most likely to fail from these sites and trigger the error handling code. During runtime testing, FIFUZZ incorporates a context-sensitive Software Fault Injection–based fuzzing [90].

We have not related these fuzzers that checks for instrumentation errors. However, it is always applicable to check for memory errors, since instrumenting foreign code may attract memory errors [111]. In such a case, tools such as REZZAN and FIFUZZ have prominence and can point to the research question RQ5(d) to fix the instrumentation errors for the benefit of better fuzzing results.

In addition to fuzzing scenarios, instrumentation techniques find applications in various domains, such as software testing, dynamic analysis, program profiling, and performance monitoring. Consequently, mitigating instrumentation errors becomes a major concern beyond the scope of fuzzing. Several research efforts have been made to address these errors in different contexts. While the detailed exploration of these works is beyond the scope of this survey, we briefly highlight some relevant studies:

(1) Researchers have focused on mitigating instrumentation-induced overhead in software performance monitoring tools. The proposed techniques reduces the impact of instrumentation on the overall system performance [72, 147, 171].
(2) In the context of dynamic analysis, some studies have addressed the challenges of instrumentation errors when monitoring program behavior. They focuses on statistical approaches to identify and correct inaccuracies in instrumentation [18, 149].
(3) Some research investigates instrumentation errors in the context of software testing and presents an automated method to detect and rectify inconsistencies introduced during code coverage measurement [23, 91, 151, 160].

While these studies demonstrate the importance of identifying the significance of robust instrumentation and addressing instrumentation errors in different application domains, we emphasize the need for further research and innovation to ensure reliable instrumentation across various contexts. We stress the necessity for continued research and innovation in the realm of instrumentation error detection and prevention. The studies we have reviewed provide valuable insights, but they also highlight the dynamic and evolving nature of instrumentation vulnerabilities.

8 SUMMARY OF RECENT ARTICLES ON FUZZING

Table 2 shows an overview of the recent articles and a detailing of their classifications. We have also mentioned the directions in those articles for improving fuzzing performance.

The codes mentioned in the core techniques in Table 2 are as follows. ⨷(a): Coverage guided fuzzer + Dynamic analysis + Mutation based deep learning network with Attention mechanism, ⨷(b): Coverage based grey-box fuzzer + Parallel fuzzing + Dynamic task allocation + Task aware fuzzing, ⨷(c): Mutation based grey-box fuzzer + Path awareness + Propagation and inference based taint analysis, ⨷(d): Language-specific fuzzer + Grammar awareness + Constrained mutation + Semantic validation, ⨷(e): Coverage-guided fuzzer + Bug impact awareness + LLVM based, ⨷(f): Coverage-guided fuzzer + *Syscall* influence relation identification + Relationship learning algorithm + Static and dynamic learning, ⨷(g): Generation based + Coverage-guided + Type aware fuzzer + Windows kernel fuzzing + Static binary analysis, ⨷(h): Coverage-guided fuzzer + Symbolic execution + Indirect control flow transfer between Syscalls + Kernel sys call fuzzing + Static analysis, ⨷(i): Coverage-guided fuzzer + Data race (concurrency) detection + Dynamic analysis + Test case generation, ⨷(j): Directed grey-box fuzzer + Static and precondition analysis + LLVM based, ⨷(k): AFL + E9Patch + Static binary rewriting + Trampoline-based rewriting, ⨷(l): Coverage-based grey-box fuzzer (AFL) + Stochastic binary rewriting + Probabilistic inference problem, ⨷(m): Coverage-based grey-box fuzzer (AFL) + Static binary rewriting + ZAX transformation + Feedback enhancement, ⨷(n): Coverage-guided tracing + Coverage preservation + Static and dynamic binary analysis + LLVM based, ⨷(o): AFL + Full stack testing + Dynamic binary rewriting (Online trace collector) + Offline trace analyzer, ⨷(p): Black-box fuzzer + Grammar based fuzzer + Syntax inference mechanism + Dynamic binary analysis, ⨷(q): White-box, Coverage-based grey-box fuzzer + hybrid analysis + taint tracking, ⨷(r): White-box, Coverage-based grey-box fuzzer + hybrid analysis + JIT compilation + path prioritization + optimal search strategy, ⨷(s): Coverage-based grey-box fuzzer + Instrumentation error find and fix + Static binary analysis. The fuzzing analysis performed in each of the recent fuzzers are as ⨶(S), ⨶(K), ⨶(B), ⨶(F), ⨶(H), and ⨶(E) represent source code, kernel, binary, firmware, hybrid (symbolic + concolic), and instrumentation errors, respectively. The numbers circled in the direction of these fuzzers are as follows. ❶*time and cost of analysis*, ❷*coverage or reachability analysis*, ❸*runtime overhead*, ❹*pre-condition inference*, ❺*bug detection*, ❻*complex bugs*, ❼*seed generation*, ❽*instrumentation errors*, ❾*parallel fuzzing*, and ❿*soundness*. Furthermore, we have mentioned various benchmarks used in these approaches.

9 CHALLENGES AND FUTURE DIRECTIONS

While it is possible to identify bugs and vulnerabilities using a fuzzer, drawing definitive conclusions regarding its effectiveness in detecting privilege escalation, remote code execution, and potential side-channel attacks requires further investigation [13].

Table 2. Summary of Recent Fuzzers Categorized by Technique, Publication, and the Fuzzing Approach

Fuzzer	Article and Year of Publication	Core Techniques	Fuzzing Analysis	Open Source	Direction	Compared Fuzzers
ATTuzz [187]	CoRR'21	(a)	(S)	✗	❶❻	AFL [122], Driller [155], Vuzzer [146], AFLFast [17], NEUZZ [152]
AFLTeam [142]	IEEE/ACM ASE'21	(b)	(S)	✓	❶❷❺❾	AFL-P (Parallel) [122]
PATA [108]	IEEE S&P'22	(c)	(S)	✗	❷❻	AFL [122], Vuzzer [146], MOpt [114], Angora [26], Redqueen [4], GREYONE [59], TortoiseFuzz [169]
PolyGlot [28]	IEEE S&P'21	(d)	(S)	✓	❷❺❼	AFL [122], Qsym [177], Nautilus [3], CSmith [175], DIE [135]
SyzScope [189]	USENIX'22	(e)	(K)	✓	❻	No comparison fuzzers (Comparison of Bug effectiveness (low or high risk))
HEALER [156]	ACM SOSP'21	(f)	(K)	✓	❶❷❻	Syskaller [43], Moonshine [133]
NtFuzz [31]	IEEE S&P'21	(g)	(K)	✓	❺❿	No comparison fuzzers (Finding bugs from windows core .dll libraries)
HFL [97]	NDSS'20	(h)	(K)	✗	❷❻	kAFL [150], Moonshine [133], S2E [29], TriforceAFL [87], Syskaller [43]
Krace [174]	IEEE S&P'20	(i)	(K)	✓	❷❻❼	Syskaller [43], Razzer [85]
Beacon [80]	IEEE S&P'**22**	(j)	(B)	✓	❶❷❸❹	AFL [122], AFLGo [14], AFL++ [52], Hawkeye [24], MOpt [114]
E9AFL [62]	IEEE/ACM ASE'21	(k)	(B)	✓	❶❷	AFL-gcc [122], AFL-dyninst [11], AFL-QEMU [122]
StochFuzz [182]	IEEE S&P'21	(l)	(B)	✓	❺❿	AFL-gcc [122], AFL-clang-fast [122], AFL-QEMU [122], PTFuzzer [179], E9Patch [45], RetroWrite [41]
Zafl [126]	USENIX'21	(m)	(B)	✗	❶❷❻	AFL-dyninst [11], AFL-QEMU [122], DynamoRio [123], Intel PT [82], PIN [113], RetroWrite [41]
HExcite [127]	ACM CCS'21	(n)	(B)	✓	❶❷❻	AFL[122], AFL-dyninst [11], RetroWrite [41]
μAFL [105]	IEEE/ACM ICSE'22	(o)	(F)	✗	❶❷❻	P^2IM [50], Avatar [178], μEmu [185]
Snipuzz [51]	ACM CCS'21	(p)	(F)	✓	❺❼	IoTFuzzer, BooFuzz, Doona [48], NEMESYS [98]
Confetti [100]	IEEE/ACM ICSE'22	(q)	(H)	✓	❷❻❻❿	JQF-Zest [132]
Fuzzolic [15]	Comput. & Secur.'21	(r)	(H)	✓	❷❺❻❼❿	AFL++, Eclipser, Qsym [48], SymCC [143]
InstruGuard [111]	IEEE/ACM ASE'21	(s)	(E)	✓	❷❽	AFL [122], AFL++ [52], MemLock [170], Angora [26], FairFuzz [103], MOpt [114]

- *Smart detection.* We want to pinpoint the sophisticated approach to detecting bugs rather than simple program crashes or anomalies. Thus we identify specific types of bugs, such as memory leaks, buffer overflows, or other vulnerabilities that attackers could exploit. The methods discussed in this work, for example, code coverage analysis, symbolic execution, and mutation/grammar-based fuzzing, can be considered *smart*. We address *smart* for generating test inputs and the method used for detecting bugs. Generally, a *smart* fuzzing approach will combine both. A fuzzer is efficient if it can find vulnerabilities by causing a crash. Nevertheless, it is required that fuzzing be intelligent to detect all possible unusual behaviors of source code under analysis. While directed grey-box fuzzing is used for detecting and reproducing bugs along with patch testing, it suffers from performance reduction, since additional instrumentation and data analysis are required. Seeds in directed grey-box fuzzing are prioritized based on their proximity to the target sites, and randomness in seed generation is another concern. However, when multiple targets exist, fuzzing does not address the interrelationship between the targets. For instance, the spatial, state, and interleaving relationships that govern the positions, program states, state transition maps, and shared threads of the target site variables are crucial while fuzzing multiple targets. Nonetheless, this approach will achieve better code coverage by reducing execution time, energy consumption, and memory consumption [166].
- *Interesting paths.* Similarly, in the urge to find *interesting* paths, and coverage-based grey-box fuzzing, their power schedules assign more energy than needed. In general, the generation of initial seeds has always been a concern for mutation-based fuzzers. Furthermore, ML techniques are used to auto-generate seeds for analyzing programs that are GUI based and take pdf and HTML as input; this is currently an important area of research. Another research approach is utilizing reinforcement learning techniques to dynamically optimize fuzzing tasks and seed inputs in the context of kernel fuzzing [162]. This process shows promising results in improving code coverage and crash detection while maintaining a low-performance overhead. White-box fuzzing provides full path coverage, but program statements involving pointer arithmetic calls to the operating system and library functions will partially make the path remain unexplored, reducing the fuzzer's efficacy. SMT solvers [38] generate effective formulas, but the computational overhead to solving the formulas is significantly high. However, complex constraints are still challenging to solve and computationally expensive.
- *Identifying bugs.* In an application, memory access errors, such as *use before initialization*, *use after free*, or *stack* or *heap overflow*, are considered dangerous bugs. Similarly, bugs related to concurrency issues, specifically non-deadlock bugs and their categories on atomicity and order bugs, are more common. Furthermore, a performance-sensitive code may possess bugs related to algorithmic complexity. Fuzzing is one of the best approaches to finding all the above-mentioned bugs, but identifying the bugs associated with logical and privilege escalation is still difficult. An empirical study needs to be carried out in fuzzing as such unexplored distributed security vulnerabilities remain. Furthermore, there is a need to extend the fuzzing capability to all programming languages as most of the present-day fuzzers revolve around C/C++ programming languages. The use of mutational analysis or mutational score can have effectiveness instead of structural coverage measure to address the aforementioned challenges effectively [13]. Currently, there is no efficient mutational framework with optimized computational expense. Likewise, the frequent mutational score gets affected by redundant/duplicate mutants and equivalent mutants, making the count of killable mutants high, making it difficult to evaluate the fuzzer. The concept of an appropriate time budget based on the number of mutants can be considered to evaluate the tradeoff between efficiency and effectiveness.

Assessing residual security risks after an unsuccessful fuzzing has been difficult. Thus, future research should develop methodical probabilistic frameworks to quantitatively measure accuracy

in assessing such risks that exist in black-, white-, and grey-box fuzzers with limitations in finding residual errors. Qualitative analysis of fuzzing is conducted by feeding synthetic faults to the system. However, the relativeness of synthetic bugs with real ones and the probability that it addresses all kinds of bugs must be well studied.

REFERENCES

[1] Adobe 2021. Living off the Land (LotL) Classifier. Retrieved from https://github.com/adobe/libLOL

[2] armDeveloper 2011. Embedded Trace Macrocell Architecture Specification in versions of the ARM architecture: ETMv1.0 to ETMv3.5. Retrieved from https://developer.arm.com/documentation/ihi0014/latest

[3] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, A.-R. Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for deep bugs with grammars. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'19)*. 1–15.

[4] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with input-to-state correspondence. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'19)*. 1–15.

[5] Vaggelis Atlidakis, Roxana Geambasu, Patrice Godefroid, Marina Polishchuk, and Baishakhi Ray. 2020. Pythia: Grammar-based fuzzing of REST APIs with coverage-guided feedback and learning-based mutations. *CoRR* abs/2005.11498 (2020). arXiv:2005.11498. https://arxiv.org/abs/2005.11498

[6] Avast 2021. What Is EternalBlue? Retrieved from https://www.avast.com/c-eternalblue

[7] Fabrice B. 2005. QEMU, a fast & portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference (ATC'05)*. 41–46.

[8] Jinsheng Ba, Gregory J. Duck, and Abhik Roychoudhury. 2022. Efficient greybox fuzzing to detect memory errors. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE'22)*. IEEE/ACM, 1–12.

[9] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2019. A survey of symbolic execution techniques. *ACM Comput. Surv.* 51, 3 (2019), 1–39.

[10] Mislav Balunović, Pavol Bielik, and Martin Vechev. 2018. Learning to solve SMT formulas. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS'18)*. 10338–10349.

[11] Andrew R. Bernat and Barton P. Miller. 2011. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools (PASTE'11)*. ACM, 9–16.

[12] Beyond Security 2018. beSTORM Black-box. Retrieved from https://beyondsecurity.com/fuzzer-bestorm-whitepaper-2.html

[13] Marcel Boehme, Cristian Cadar, and Abhik Roychoudhury. 2021. Fuzzing: Challenges & reflections. *IEEE Softw.* 38, 3 (2021), 79–86.

[14] Marcel Bohme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*. ACM, 2329–2344.

[15] Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. 2021. FUZZOLIC: Mixing fuzzing and concolic execution. *Comput. Secur.* 108, C (Sep 2021), 26 pages. https://doi.org/10.1016/j.cose.2021.102368

[16] Bull SGI, OSD 2022. Linux Test Project. Retrieved from https://linux-test-project.github.io

[17] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2019. Coverage-based greybox fuzzing as markov chain. *IEEE Trans. Softw. Eng.* 45, 5 (2019), 489–506.

[18] Boyuan Chen and Zhen Ming Jiang. 2021. A survey of software log instrumentation. *ACM Comput. Surv.* 54, 4 (2021), 1–34.

[19] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and automatic generation of high-coverage tests. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. 209–224.

[20] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing. *Commun. ACM* 56, 2 (2013), 82–90.

[21] Chen Cao, Le Guan, Jiang Ming, and Peng Liu. 2020. Device-agnostic firmware execution is possible: A concolic execution approach. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC'20)*. IEEE, 746–759.

[22] Census Labs 2016. Choronzon: An Evolutionary Knowledge-based Fuzzer. Retrieved from https://github.com/CENSUS/choronzon

[23] Boyuan Chen, Jian Song, Peng Xu, Xing Hu, and Zhen Ming Jack Jiang. 2018. An automated approach to estimating code coverage measures via execution logs. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE'18)*. IEEE/ACM, 305–316.

[24] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*. IEEE, 2095–2108.

[25] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, Xiaofeng Wang, W. Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. 2018. IOTFUZZER: Discovering memory corruptions in IoT through App fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'18)*. 1–15.

[26] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'18)*. IEEE, 711–725.

[27] Yaohui Chen, Dongliang Mu, Jun Xu, Zhichuang Sun, Wenbo Shen, Xinyu Xing, Long Lu, and Bing Mao. 2019. PTRIX: Efficient hardware-assisted fuzzing for COTS binary. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (Asia CCS'19)*. ACM, 633–645.

[28] Yongheng Chen, Rui Zhong, Hong Hu, Hangfan Zhang, Yupeng Yang, Dinghao Wu, and Wenke Lee. 2021. One engine to fuzz 'em all: Generic language processor testing with semantic validation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'21)*. IEEE, 642–658.

[29] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A platform for in-vivo multi-path analysis of software systems. *ACM SIGARCH Comput. Archit. News* 39, 1 (2011), 265–278.

[30] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. 2019. Grey-box concolic testing on binary code. In *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE'19)*. IEEE/ACM, 736–747.

[31] Jaeseung Choi, Kangsu Kim, Daejin Lee, and Sang Kil Cha. 2021. NTFUZZ: Enabling type-aware kernel fuzzing on windows with static binary analysis. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'21)*. IEEE, 677–693.

[32] Abraham A. Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. 2020. HALucinator: Firmware re-hosting through abstraction layer emulation. In *Proceedings of the 29th USENIX Security Symposium* 1201–1218.

[33] ClusterFuzz 2021. ClusterFuzz. Retrieved from https://google.github.io/clusterfuzz/

[34] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Krügel, and Giovanni Vigna. 2017. DIFUZE: Interface aware fuzzing for kernel drivers. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*. ACM, 2123–2138.

[35] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-C. In *Proceedings of the Software Engineering and Formal Methods (SEFM'12)*. Springer, 233–247.

[36] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. 2016. LAVA: Large-scale automated vulnerability addition. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'16)*. IEEE, 110–121.

[37] Dave Aitel 2002. Block-Based Protocol Analysis. Retrieved from https://www.immunitysec.com/downloads/

[38] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*. 337–340.

[39] Leonardo De Moura and Nikolaj Bjørner. 2011. SAT: Introduction & applications. *Commun. ACM* 54, 9 (2011), 69–77.

[40] Department of Defense (DoD) 2019. DevSecOps Reference Design. Retrieved from https://dodcio.defense.gov/Portals/0/Documents/

[41] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. RetroWrite: Statically instrumenting COTS binaries for fuzzing and sanitization. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'20)*. IEEE, 1497–1511.

[42] Dmitry Vyukov 2021. Randomized Testing for Go. Retrieved from https://github.com/dvyukov/go-fuzz

[43] Dmitry Vyukov and Andrey Konovalov 2022. Syzkaller. Retrieved from https://github.com/google/syzkaller

[44] Dmytro Oleksiuk 2011. Ioctl Fuzzer. Retrieved from https://github.com/Cr4sh/ioctlfuzzer/

[45] Gregory J. Duck, Xiang Gao, and Abhik Roychoudhury. 2020. Binary rewriting without control flow recovery. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'20)*. ACM, 151–163.

[46] Martin Eberlein, Yannic Noller, Thomas Vogel, and Lars Grunske. 2020. Evolutionary grammar-based fuzzing. In *Proceedings of the 12th International Symposium of Search-Based Software Engineering (SSBSE'20)*. Springer-Verlag, 105–120.

[47] Eduard Kovacs 2021. Open Source Security Tools. Retrieved from https://google-adobe-announce-new-open-source-security-tools

[48] Eldar Marcussen 2019. Doona—Network Based Protocol Fuzzer. Retrieved from https://github.com/wireghoul/doona

[49] James Fell. 2017. A Review of Fuzzing Tools and Methods. Retrieved from https://tinyurl.com/4r6pkuha

[50] Bo Feng, Alejandro Mera, and Long Lu. 2020. P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *Proceedings of the 29th USENIX Security Symposium*, 1237–1254.

[51] Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu, Minhui Xue, Sheng Wen, Dongxi Liu, Surya Nepal, and Yang Xiang. 2021. Snipuzz: Black-box fuzzing of IoT firmware via message snippet inference. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS'21)*. ACM, 337–350.
[52] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT'20)*. 1–12.
[53] United States Food & Drug Administration. 2018. Efforts to Strengthen the Agency's Medical Device Cybersecurity Program. Retrieved from https://www.fda.gov/news-events/
[54] FuzzBall 2021. FuzzBALL Symbolic Engine. Retrieved from https://github.com/bitblaze-fuzzball/fuzzball
[55] FuzzBallVine 2021. FuzzBALL: Vine-based Binary Symbolic Execution. Retrieved from http://bitblaze.cs.berkeley.edu/fuzzball.html
[56] Fuzzit 2020. Coverage Guided Fuzz Testing for Java. Retrieved from https://github.com/fuzzitdev/javafuzz
[57] Fuzzit 2020. Coverage Guided Fuzz Testing for Javascript. Retrieved from https://github.com/fuzzitdev/jsfuzz
[58] Fuzzit 2020. Coverage Guided Fuzz Testing for Python. Retrieved from https://github.com/fuzzitdev/pythonfuzz
[59] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. GREYONE: Data flow sensitive fuzzing. In *Proceedings of the 29th USENIX Security Symposium*, 2577–2594.
[60] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path sensitive fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'18)*. IEEE, 679–696.
[61] Vijay Ganesh, Tim Leek, and Martin Rinard. 2009. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*. IEEE, 474–484.
[62] Xiang Gao, Gregory J. Duck, and Abhik Roychoudhury. 2021. Scalable fuzzing of program binaries with E9AFL. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE'21)*. IEEE/ACM, 1247–1251.
[63] Patrice Godefroid. 2020. Fuzzing: Hack, art, and science. *Commun. ACM* 63, 2 (2020), 70–76.
[64] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-based whitebox fuzzing. *ACM SIGPLAN Not.* 43, 6 (2008), 206–215.
[65] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox fuzzing for security testing: SAGE has had a remarkable impact at microsoft. *Queue* 10, 1 (2012), 20–27.
[66] Google 2015. Honggfuzz. Retrieved from https://github.com/google/honggfuzz
[67] Google 2016. Fuzzing Chrome. Retrieved from https://security.googleblog.com/2016/08/guided-in-process-fuzzing-of-chrome.html
[68] Google 2021. ClusterFuzzLite. Retrieved from https://security.googleblog.com/2021/11/clusterfuzzlite-continuous-fuzzing-for.html
[69] Google 2021. Fuzzer-test-suite. Retrieved from https://github.com/google/fuzzer-test-suite
[70] Google 2022. OSS-fuzz. Retrieved from https://github.com/google/oss-fuzz
[71] Samuel Groß. 2018. *FuzzIL: Coverage Guided Fuzzing for JavaScript Engines*. Master's thesis. Institute of Theoretical Informatics, Competence Center for Applied Security Technology.
[72] Binfa Gui, Wei Song, Hailong Xiong, and Jeff Huang. 2022. Automated use-after-free detection and exploit mitigation: How far have we gone? *IEEE Trans. Softw.Eng.* 48, 11 (2022), 4569–4589.
[73] Tao Guo, Puhan Zhang, Xin Wang, and Qiang Wei. 2013. GramFuzz: Fuzzing testing of web browsers based on grammar analysis and structural mutation. In *Proceedings of the International Conference on Informatics Applications*. IEEE, 212–215.
[74] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christophe Kruegel, and Giovanni Vigna. 2019. Toward the analysis of embedded firmware through automated re-hosting. In *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID'19)*. USENIX, 135–150.
[75] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. 2013. Dowsing for Overflows: A guided fuzzer to find buffer boundary violations. In *Proceedings of the 22nd USENIX Security Symposium*, 49–64.
[76] William H Hawkins, Jason D Hiser, Michele Co, Anh Nguyen-Tuong, and Jack W. Davidson. 2017. Zipr: Efficient static binary rewriting for security. In *Proceedings of the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'17)*. IEEE, 559–566.
[77] Hfiref0x 2021. NtCall64. Retrieved from https://github.com/hfiref0x/NtCall64
[78] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *Proceedings of the 21st USENIX Security Symposium*, 445–458.
[79] Zhicheng Hu, Jianqi Shi, YanHong Huang, Jiawen Xiong, and Xiangxing Bu. 2018. GANFuzz: A GAN-based industrial network protocol fuzzing framework. In *Proceedings of the 15th ACM International Conference on Computing Frontiers (CF'18)*. ACM, 138–145.

[80] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2022. BEACON: Directed grey-box fuzzing with provable path pruning. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'22)*. IEEE, 104–118.

[81] Heqing Huang, Peisen Yao, Rongxin Wu, Qingkai Shi, and Charles Zhang. 2020. Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'20)*. IEEE, 1613–1627.

[82] Intel 2020. Processor Tracing. Retrieved from https://software.intel.com/content/www/us/en/develop/blogs/processor-tracing.html

[83] J. B. Crawford 2018. A Survey of Some Free Fuzzing Tools. Retrieved from https://lwn.net/Articles/744269/

[84] James Spadaro and Lilith Wyatt 2017. Mutiny Fuzzing Framework and Decept Proxy. Retrieved from http://blog.talosintelligence.com/

[85] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. 2019. Razzer: Finding kernel race bugs through fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'19)*. IEEE, 754–768.

[86] Jesse Hertz 2017. Triforce Linux Syscall Fuzzer. Retrieved from https://github.com/nccgroup/TriforceLinuxSyscallFuzzer

[87] Jesse Hertz 2017. TriforceAFL. Retrieved from https://github.com/nccgroup/TriforceAFL

[88] Jesse Ruderman 2021. jsfunfuzz: Test JS Engine in Firefox. Retrieved from https://github.com/MozillaSecurity/funfuzz/

[89] Tiantian Ji, Zhongru Wang, Zhihong Tian, Binxing Fang, Qiang Ruan, Haichen Wang, and Wei Shi. 2020. AFLPro: Direction sensitive fuzzing. *J. Inf. Secur. Appl.* 54 (2020), 1–14.

[90] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. 2020. Fuzzing error handling code using context-sensitive software fault injection. In *Proceedings of the 29th USENIX Security Symposium*, 2595–2612.

[91] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. 2022. Context-Sensitive and directional concurrency fuzzing for data-race detection. In *Proc. of the Net. & Distr. Sec. Symp. (NDSS)*. 1–18.

[92] Wei Jin and Alessandro Orso. 2012. BugRedux: Reproducing field failures for in-house debugging. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*. IEEE, 474–484.

[93] Evan Johnson, Maxwell Bland, YiFei Zhu, Joshua Mason, Stephen Checkoway, Stefan Savage, and Kirill Levchenko. 2021. Jetset: Targeted firmware rehosting for embedded systems. In *Proceedings of the 30th USENIX Security Symposium*, 321–338.

[94] Josh Fruhlinger 2018. The Mirai Botnet Explained. Retrieved from https://www.csoonline.com/article/

[95] Joshua Pereyda 2017. boofuzz: Network Protocol Fuzzing for Humans. Retrieved from https://boofuzz.readthedocs.io/en/stable/

[96] Kaspersky 2017. WannaCry. Retrieved from https://www.kaspersky.com/resource-center/threats/ransomware-wannacry

[97] Kyungtae Kim, Dae R. Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2017. HFL: Hybrid fuzzing on the linux kernel. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'17)*. 1–17.

[98] Stephan Kleber, Henning Kopp, and Frank Kargl. 2018. NEMESYS: Network message syntax reverse engineering by analysis of the intrinsic structure of individual messages. In *Proceedings of 12th USENIX Workshop on Offensive Technologies (WOOT'18)*. 1–13.

[99] Youngjoo Ko, Bin Zhu, and Jong Kim. 2022. Fuzzing with automatically controlled interleavings to detect concurrency bugs. *J. Syst. Softw.* 191 (2022), 111379.

[100] James Kukucka, Luís Pina, Paul Ammann, and Jonathan Bell. 2022. CONFETTI: Amplifying concolic guidance for fuzzers. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (ICSE'22)*. IEEE/ACM, 438–450.

[101] ZZUF 0.15 (latest). 2017. ZZUF: Multi-purpose Fuzzer. Retrieved February 1, 2022 from http://caca.zoy.org/wiki/zzuf

[102] C. Lattner and V. Adve. 2004. LLVM: A compilation framework for lifelong program analysis transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04)*. IEEE, 75–86.

[103] Caroline Lemieux and Koushik Sen. 2017. FairFuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage. *CoRR* abs/1709.07101 (2017). arXiv:1709.07101. http://arxiv.org/abs/1709.07101

[104] Jun Li, Bodong Zhao, and Chao Zhang. 2018. Fuzzing: A survey. *Cybersecurity* 1, 6 (2018), 1–13.

[105] Wenqiang Li, Jiameng Shi, Fengjun Li, Jingqiang Lin, Wei Wang, and Le Guan. 2022. μAFL: Non-intrusive feedback-driven fuzzing for microcontroller firmware. In *Proc. of the 44th International Conference on Software Engineering (ICSE)*. ACM, 1–12.

[106] Yuwei Li, Shouling Ji, Chenyang Lyu, Yuan Chen, Jianhai Chen, Qinchen Gu, Chunming Wu, and Raheem Beyah. 2022. V-Fuzz: Vulnerability prediction-assisted evolutionary fuzzing f or binary programs. *IEEE Trans. Cybern.* 52, 5 (2022), 3745–3756.

[107] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. 2018. Fuzzing: State of the art. *IEEE Trans. Reliabil.* 67, 3 (2018), 1199–1218.
[108] Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Yu Jiang, Jianzhong Liu, Zhe Liu, and Jiaguang Sun. 2022. PATA: Fuzzing with path aware taint analysis. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'22)*. IEEE, 154–170.
[109] LibFuzzer 2017. libFuzzer—A Library for Coverage-guided Fuzz Testing. Retrieved from https://llvm.org/docs/LibFuzzer.html
[110] Xiao Liu, Xiaoting Li, Rupesh P, and Dinghao Wu. 2019. DeepFuzz: Automatic generation of syntax valid c programs for fuzz testing. *Proc. AAAI Conf. Artif. Intell.* 33, 1 (2019), 1044–1051.
[111] Yuwei Liu, Yanhao Wang, Purui Su, Yuanping Yu, and Xiangkun Jia. 2021. InstruGuard: Find and fix instrumentation errors for coverage-based greybox fuzzing. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE'21)*. IEEE/ACM, 568–580.
[112] Lucas Leong. 2019. Make Static Instrumentation Great Again: High Performance Fuzzing for Windows System. Retrieved from https://msrnd-cdn-stor.azureedge.net/bluehat/bluehatil/2019/
[113] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. *ACM SIGPLAN Not.* 40, 6 (2005), 190–200.
[114] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized mutation scheduling for fuzzers. In *Proceedings of the 28th USENIX Security Symposium*, 1949–1966.
[115] Zheyu Ma, Bodong Zhao, Letu Ren, Zheming Li, Siqi Ma, Xiapu Luo, and Chao Zhang. 2022. PrIntFuzz: Fuzzing linux drivers via automated virtual device simulation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'22)*. ACM, 404–416.
[116] Rupak Majumdar and Ru-Gang Xu. 2007. Directed test generation using symbolic grammars. In *Proceedings of the 6th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'07)*. ACM, 553–556.
[117] Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. S, and Maverick Woo. 2021. The art, science, and engineering of fuzzing: A survey. *IEEE Trans. Softw. Eng.* 47, 11 (2021), 2312–2331.
[118] Paul Dan Marinescu and Cristian Cadar. 2020. KATCH: High-coverage testing of software patches. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'20)*. ACM, 235–245.
[119] Martin Vuagnoux 2005. Autodafe: An Act of Software Tortur. Retrieved from https://infoscience.epfl.ch/record/140525?ln=en
[120] Phil McMinn. 2004. Search-based software test data generation: A survey. *Softw. Test Verif. Reliabil.* 14, 2 (2004), 105–156.
[121] Alejandro Mera, Bo Feng, Long Lu, and Engin Kirda. 2021. DICE: Automatic emulation of DMA input channels for dynamic firmware analysis. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'21)*. IEEE, 1938–1954.
[122] Michał Zalewski. 2013. American Fuzzy Lop (AFL). Retrieved from https://lcamtuf.coredump.cx/afl/
[123] MIT and Hewlett-Packard. 2009. DynamoRIO. Retrieved from https://dynamorio.org/page_home.html
[124] Mozilla Security. 2020. Dharma: Context-free Grammar Fuzzer. Retrieved from https://github.com/mozillasecurity/dharma
[125] Stefan Nagy and Matthew Hicks. 2019. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'19)*. IEEE, 787–802.
[126] Stefan Nagy, Anh Nguyen-Tuong, Jason D. Hiser, Jack W. Davidson, and Matthew Hicks. 2021. Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing. In *Proceedings of the 30th USENIX Security Symposium* 1683–1700.
[127] Stefan Nagy, Anh Nguyen-Tuong, Jason D. Hiser, Jack W. Davidson, and Matthew Hicks. 2021. Same coverage, less bloat: Accelerating binary-only fuzzing with coverage-preserving coverage-guided tracing. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS'21)*. ACM, 351–365.
[128] Nick Fitzgerald 2022. Cargo-Fuzz for Rust-code. Retrieved from https://rust-fuzz.github.io/book/cargo-fuzz/guide.html
[129] Zhenyu Ning and Fengwei Zhang. 2017. Ninja: Towards transparent tracing and debugging on ARM. In *Proceedings of the 26th USENIX Security Symposium*. 33–49.
[130] Mitchell Olsthoorn, Arie van Deursen, and Annibale Panichella. 2020. Generating highly-structured input data by combining search-based testing and grammar-based fuzzing. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE'20)*. IEEE/ACM, 1224–1228.
[131] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. JQF: Coverage-guided property-based testing in Java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'19)*. ACM, 398–401.

[132] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'19)*. ACM, 329–340.

[133] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. MoonShine: Optimizing OS fuzzer seed selection with trace distillation. In *Proceedings of the 27th USENIX Security Symposium*, 729–743.

[134] Jianfeng Pan, Guanglu Yan, and Xiaocao Fan. 2017. Digtool: A virtualization-based framework for detecting kernel vulnerabilities. In *Proceedings of the 26th USENIX Security Symposium*, 149–165.

[135] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. 2020. Fuzzing javascript engines with aspect-preserving mutation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'20)*. IEEE, 1628–1642.

[136] Patrice Godefroid 2020. Fuzzing and Why It's an Important Tool for Developers. Retrieved from https://www.microsoft.com/en-us/research/blog/a-brief-introduction-to-fuzzing-and-why-its-an-important-tool-for-developers/

[137] PEACH. 2014. PEACH Fuzzer. Retrieved from https://gitlab.com/peachtech/peach-fuzzer-community

[138] Pedram Amini and Aaron Portnoy 2015. Sulley: Fuzzing Framework. Retrieved from https://github.com/OpenRCE/sulley

[139] Hui Peng and Mathias Payer. 2020. USBFuzz: A framework for fuzzing USB drivers by device emulation. In *Proceedings of the 29th USENIX Security Symposium*, 2559–2575.

[140] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: Fuzzing by program transformation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'18)*. IEEE, 697–710.

[141] Van-Thuan Pham, Marcel Böhme, Andrew E. Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. 2021. Smart greybox fuzzing. *IEEE Trans. Softw. Eng.* 47, 9 (2021), 1908–1997.

[142] Van-Thuan Pham, Manh-Dung Nguyen, Quang-Trung Ta, Toby Murray, and Benjamin I.P. Rubinstein. 2021. Towards systematic and dynamic task allocation for collaborative parallel fuzzing. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE'21)*. IEEE/ACM, 1337–1341.

[143] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile! In *Proceedings of the 29th USENIX Security Symposium*, 181–198.

[144] Maria F. Prevezianou. 2021. *WannaCry as a Creeping Crisis*. Springer International Publishing.

[145] Corina S. Păsăreanu and Neha Rungta. 2010. Symbolic PathFinder: Symbolic execution of Java bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'10)*. IEEE/ACM, 179–180.

[146] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'17)*. 1–14.

[147] David Georg Reichelt, Stefan Kühne, and Wilhelm Hasselbring. 2023. Towards solving the challenge of minimal overhead monitoring. In *Proceedings of the ACM/SPEC International Conference on Performance Engg (ICPE'23)*. ACM, 381–388.

[148] Gary J. Saavedra, Kathryn N. Rodhouse, Daniel M. Dunlavy, and W. Philip Kegelmeyer. 2019. A review of machine learning applications in fuzzing. *CoRR* abs/1906.11133 (2019). arXiv:1906.11133. http://arxiv.org/abs/1906.11133

[149] César Sánchez, Gerardo Schneider, Wolfgang Ahrendt, Ezio Bartocci, Domenico Bianculli, Christian Colombo, Yliès Falcone, Adrian Francalanza, Srdan Krstic, João M. Lourenço, Dejan Nickovic, Gordon J. Pace, José Rufino, Julien Signoles, Dmitriy Traytel, and Alexander Weiss. 2019. A survey of challenges for runtime verification from advanced application domains (beyond software). *Formal Methods Syst. Des.* 54, 3 (2019).

[150] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-assisted feedback fuzzing for OS kernels. In *Proceedings of the 26th USENIX Security Symposium*, 167–182.

[151] Amanda Schwartz, Daniel Puckett, Ying Meng, and Gregory Gay. 2018. Investigating faults missed by test suites achieving high code coverage. *J. Syst. Softw.* 144 (2018), 106–120.

[152] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2019. NEUZZ: Efficient fuzzing with neural program smoothing. In *Proc. of the IEEE Symp. on Security and Privacy (S&P)*. IEEE, 803–817.

[153] Yuheng Shen, Hao Sun, Yu Jiang, Heyuan Shi, Yixiao Yang, and Wanli Chang. 2021. Rtkaller: State-aware task generation for RTOS fuzzing. *ACM Trans. Embed. Comput. Syst.* 20, 5s (2021), 1–22.

[154] Prashast Srivastava and Mathias Payer. 2021. Gramatron: Effective grammar-aware fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'21)*. ACM, 244–256.

[155] Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'16)*. 1–16.

[156] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. 2021. HEALER: Relation learning guided kernel fuzzing. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. ACM, 344–358.

[157] Synopsys. 2020. The Heartbleed Bug. Retrieved from https://heartbleed.com
[158] Ari Takanen, Jared Demott, Charles Miller, and Atte Kettunen. 2008. *Fuzzing for Software Security Testing and Quality Assurance*. Artech.
[159] Talos-vulndev. 2018. AFL-Dyninst. Retrieved from https://github.com/talos-vulndev/afl-dyninst
[160] Dries Vanoverberghe, Jonathan de Halleux, Nikolai Tillmann, and Frank Piessens. 2012. State Coverage: Software validation metrics beyond code coverage. In *Proceedings of the Theory and Practice of Computer Science (SOFSEM'12)*. Springer, 542–553.
[161] David Vitek. 2016. Auditing code for security vulnerabilities with CodeSonar. In *Proceedings of the IEEE Cybersecurity Development (SecDev'16)*. 154–154.
[162] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V.K., and Nael Abu-Ghazaleh. 2021. SyzVegas: Beating kernel fuzzing odds with reinforcement learning. In *Proceedings of the 30th USENIX Security Symposium* 2741–2758.
[163] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-driven seed generation for fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'17)*. IEEE, 579–594.
[164] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-aware greybox fuzzing. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering (ICSE'19)*. IEEE/ACM, 724–735.
[165] Mingzhe Wang, Jie Liang, Chijin Zhou, Yu Jiang, Rui Wang, Chengnian Sun, and Jiaguang Sun. 2021. RIFF: Reduced instruction footprint for coverage-guided fuzzing. In *Proceedings of the USENIX Annual Technical Conference (ATC'21)*. 147–159.
[166] Pengfei Wang and Xu Zhou. 2020. SoK: The progress, challenges, and perspectives of directed greybox fuzzing. *CoRR* abs/2005.11907 (2020). https://arxiv.org/abs/2005.11907
[167] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A checksum-aware directed fuzzing tool for software vulnerability detection. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'10)*. IEEE, 497–512.
[168] Yan Wang, Peng Jia, Luping Liu, Cheng Huang, and Zhonglin Liu. 2020. A systematic review of fuzzing based on machine learning techniques. *PLOS ONE* 15, 8 (2020), 1–37.
[169] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. 2020. not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'20)*. 1–15.
[170] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. 2020. MemLock: Memory usage guided fuzzing. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE'20)*. IEEE/ACM, 765–777.
[171] Michael W. Whalen, Suzette Person, Neha Rungta, Matt Staats, and Daniela Grijincu. 2015. A flexible and non-intrusive approach for computing complex structural coverage metrics. In *Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE'15)*. IEEE/ACM, 506–516.
[172] Matt Windsor, Alastair F. Donaldson, and John Wickerson. 2021. C4: The C compiler concurrency checker. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'21)*. ACM, 670–673.
[173] Xie Xiaofei and Li Xiaohong. 2019. Hybrid testing method based on symbolic execution and fuzzing. *J. Softw.* 30, 10 (2019), 3071–3089.
[174] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. 2020. Krace: Data race fuzzing for kernel file systems. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'20)*. IEEE, 1643–1660.
[175] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. *ACM SIGPLAN Not.* 46, 6 (2011), 283–294.
[176] Yoav Alon and Netanel Ben-Simon 2018. Fuzzing Adobe Reader. Retrieved from https://research.checkpoint.com/2018/
[177] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM : A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the 27th USENIX Security Symposium*, 745–761.
[178] Jonas Zaddach, Luca Bruno, Davide Balzarotti, and Aurelien Francillon. 2014. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'14)*. 1–16.
[179] Gen Zhang, Xu Zhou, Yingqi Luo, Xugang Wu, and Erxue Min. 2018. PTfuzz: Guided fuzzing with processor trace feedback. *IEEE Access* 6 (2018), 37302–37313.
[180] Tao Zhang, Yu Jiang, Runsheng Guo, Xiaoran Zheng, and Hui Lu. 2020. A survey of hybrid fuzzing based on symbolic execution. In *Proceedings of the International Conference on Cyberspace Innovation of Advanced Technologies (CIAT'20)*. ACM, 192–196.
[181] Yan Zhang, Junwen Zhang, Dalin Zhang, and Yongmin Mu. 2018. Survey of directed fuzzy technology. In *Proceedings of the IEEE 9th International Conference on Software Engineering and Service Science (ICSESS'18)*. IEEE, 696–699.

[182] Zhuo Zhang, Wei You, Guanhong Tao, Yousra Aafer, Xuwei Liu, and Xiangyu Zhang. 2021. StochFuzz: Sound and cost-effective fuzzing of stripped binaries by incremental and stochastic rewriting. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'21)*. IEEE, 659–677.

[183] Bodong Zhao, Zheming Li, Shisong Qin, Zheyu Ma, Ming Yuan, Wenyu Zhu, Zhihong Tian, and Chao Zhang. 2022. StateFuzz: System call-based state-aware linux driver fuzzing. In *Proceedings of the 31st USENIX Security Symposium*, 3273–3289.

[184] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. 2020. SQUIRREL: Testing database management systems with language validity and coverage feedback. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS'20)*. ACM, 955–970.

[185] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. 2021. Automatic firmware emulation through invalidity-guided knowledge inference. In *Proceedings of the 30th USENIX Security Symposium*, 2007–2024.

[186] Xu Zhou, Pengfei Wang, Chenyifan Liu, Tai Yue, Yingying Liu, Congxi Song, Kai Lu, Qidi Yin, and Xu Han. 2023. UltraFuzz: Towards resource-saving in distributed fuzzing. *IEEE Trans. Softw. Eng.* 49, 4 (2023), 2394–2412.

[187] Shunkai Zhu, Jingyi Wang, Jun Sun, Jie Yang, Xingwei Lin, Liyi Zhang, and Peng Cheng. 2021. Better pay attention whilst fuzzing. *CoRR* abs/2112.07143 (2021). arXiv:2112.07143. https://arxiv.org/abs/2112.07143

[188] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: A survey for roadmap. *ACM Comput. Surv.* 54, 11 (2022), 1–36.

[189] Xiaochen Zou, Guoren Li, Weiteng Chen, Hang Zhang, and Zhiyun Qian. 2022. SyzScope: Revealing high-risk security impacts of fuzzer-exposed bugs in linux kernel. In *Proceedings of the 31st USENIX Security Symposium*, 3201–3217.

Received 24 August 2022; revised 11 August 2023; accepted 21 August 2023