IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received September 8, 2021, accepted September 16, 2021, date of publication September 20, 2021,
date of current version September 29, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3114202

Exploratory Review of Hybrid Fuzzing for Automated Vulnerability Detection

FAYOZBEK RUSTAMOV[1], JUHWAN KIM[2], JIHYEON YU[2], AND JOOBEOM YUN[2]

[1]Department of Computer and Information Security, Sejong University, Seoul, Gwangjin-gu 05006, South Korea

[2]Department of Computer and Information Security, and Convergence Engineering for Intelligent Drone, Sejong University, Seoul 05006, South Korea

Corresponding author: Joobeom Yun (jbyun@sejong.ac.kr)

This work was supported by the Ministry of Science and ICT (MSIT), South Korea, through the Information Technology Research Center (ITRC) Support Program supervised by the Institute for Information and Communications Technology Planning and Evaluation (IITP) under Grant IITP-2021-2018-0-01423 and Grant IITP-2021-2020-0-01602.

ABSTRACT Recently, software testing has become a significant component of information security. The most reliable technique for automated software testing is a fuzzing tool that feeds programs with random test-input and detects software vulnerabilities that are critical to security. Similarly, symbolic execution has gained the most attention as an efficient testing tool for producing smart test-inputs and discovering hard-to-reach bugs using search-based heuristics and compositional approaches. The combination of fuzzing and symbolic execution makes software testing more efficient by mitigating the limitations in each other. Although several studies have been conducted on hybrid fuzzing in recent years, a comprehensive and consistent review of hybrid fuzzing techniques has not been explored. To add coherence to the extensive literature on hybrid fuzzing and to make it reach a large audience, this study provides an overview of key concepts along with the taxonomy of existing hybrid fuzzing tools, problems, and solutions that have been developed in this sphere. It also includes evaluations of the proposed approaches and a number of suggestions for the development of hybrid fuzzing in the future.

INDEX TERMS Hybrid fuzzing, symbolic execution, concolic execution, vulnerability, software testing, survey.

I. INTRODUCTION

The introduction of fuzzing in the early 1990s [1] brought significant changes to the field of computer and information security and has become a more prevalent approach for identifying software vulnerabilities. Generally, fuzzing refers to the method of continuously executing a Program Under Test (PUT) with produced test-inputs that can trigger a program's abnormal behavior. Practically, attackers regularly employ fuzzing for pen testing and "Automatic Exploit Generation" [2], [3]. For instance, competitor teams utilized fuzzing tools in the "2016 DARPA Cyber Grand Challenge" (CGC) [4] for their cyber systems [5]–[7]. Meanwhile, security defenders also apply fuzzing tools to protect their systems from attackers and to identify system vulnerabilities. For instance, well-known vendors such as Google [8]–[10], Adobe [11], Microsoft [12], [13], and Cisco [14] utilize fuzzing to secure their production system.

Symbolic or concolic execution [15] and fuzzing [1], [16], [17] are considered as the prominent software testing techniques for effective test-input generation and for hunting software bugs. These techniques have been highly utilized and applied in recent years. Symbolic execution (SE) is a tool for analyzing a PUT to identify the effects of test-input on each executable program path. Executing the program symbolically involves exploring all paths where the constraints of the branches are gathered. With the help of constraint solvers, it can generate effective test-inputs for each path. SE comprises techniques such as bounded model checking (BMC) [18], whitebox fuzzing [19] approaches, and search-based heuristics. Concolic execution (CE) [20], [21] is a combination of concrete and SE. It enables the collection of the constraints of branches while executing a program. The function of whitebox fuzzing is similar to that of black-box fuzzing, which gathers some initial constraints of a program sequentially and provides new test cases by solving the constraints. The BMC approach is also used to address PUT constraints with certain optimizations.

The associate editor coordinating the review of this manuscript and approving it for publication was Tyson Brooks.

Both the SE and fuzzing approaches serve as beneficial tools for discovering vulnerabilities. Considering the time limits, a fuzzing tool is an efficient manner of deeply analyzing some of the paths in a PUT, whereas SE can help explore most branches in a PUT at low depths. For example, two steps are involved in the use of hybrid fuzzing. The first step is to execute the most easy-to-cover branches in a short time by fuzzing the PUT, and the second step is to symbolically explore the hard-to-reach branches that are not covered by fuzzing. Thus, hybrid fuzzing integrates fuzzing with an SE/CE execution method to address the shortcomings of both techniques. Because of this ability to combine multiple functions, hybrid fuzzing has attracted attention and made important contributions to software vulnerability detection. This study will focus on hybrid fuzzing techniques by considering their perspectives on the abovementioned technological aspects.

A. MOTIVATION

The two predominant motivations for conducting this survey are as follows:

1) Hybrid fuzzing has already gained recognition in the field of software security because of its relative reliability and contribution to easing the determination of bugs. Additionally, it attracts notable IT companies such as Google, Microsoft, and Cisco to advance fuzzing techniques to discover vulnerabilities in the PUT. The practical advantages of hybrid fuzzing have encouraged us to illustrate the most recent techniques and new findings that have a positive effect on this field.
2) While most studies provide an overview of software testing techniques, they tend to focus on one software testing tool, such as a systematic overview exclusively pertaining to fuzzing or SE only. Despite the fact that some hybrid fuzzing surveys have reviewed a minority of hybrid approaches, there is no comprehensive study that provides an extensive overview of the hybrid fuzzing techniques. Owing to the aforementioned reasons, it is essential to provide a detailed summary of the hybrid fuzzing techniques.

B. RELATED WORKS

As mentioned above, attention has seldom been afforded to the empirical analysis of hybrid fuzzing containing SE/CE and fuzzing before previous studies. In 1999, Edvardsson [22] conducted a survey on the use of SE for an automatic test-input generation. Moreover, several meta-studies and reviews [23]–[27] have focused on practical SE/CE, results of evaluations and overview optimizations in SE/CE. Fuzzing has been empirically reviewed as a vulnerability identification technique by Sutton *et al.* [17], and McNally *et al.* [28]. Furthermore, Van Sprundel [29] published a survey of studies in the area of fuzzing. Moreover, comprehensive review studies on fuzzing have recently been presented by Manès *et al.* [30] and Liang *et al.* [31]. Fuzzing and SE have also been regarded as effective methods for the detection of software flaws in some studies [32], [33].

In addition, several review papers on hybrid testing have been published [34]–[36]. Saahil Ognawala *et al.* [34] presented a survey on a hybrid testing tool that combines only the fuzzing and SE. This survey also provides new ideas for the implementation of hybrid test-input production methods. Yang CAO *et al.* [35] presented a survey on a test-input generation model of hybrid testing methods that combined fuzzing and dynamic symbolic execution. Furthermore, this paper empirically compared AFLFast and KLEE by testing real-world programs. Recently, Zhang *et al.* [36] presented a mini-survey of hybrid testing based on SE. It provides a brief overview of vulnerability mining technologies. Although these published hybrid testing surveys provide useful information, they do not cover a majority of hybrid fuzzing tools. For example, Saahil Ognawala, Yang CAO, Tao Zhang have provided a survey on 9, 12, and 8 hybrid testing tools, respectively.

Based on the factors mentioned above, we consider that integrating the considerable improvement of hybrid fuzzing into a mainstream study is long overdue. Therefore, this empirical study is intended to demonstrate the usefulness of hybrid techniques in this field and, also provides rich statistical data for explaining the comparisons between different hybrid fuzzers and covers a wide range of research perspectives such as the trend, technical features, and genealogy of hybrid fuzzing tools. Furthermore, the corresponding comprehensive review gives a clear picture of hybrid fuzzing for beginners and professionals to comprehend its functions.

C. OUTLINE

The following explains how this study is organized. Section II discusses the review technique employed in our study, providing a summary and analysis of the selected publications. Section III is devoted to the overall workflow of vulnerability detection methods. Section IV discusses the taxonomy of fuzzing techniques, and in Section V, hybrid fuzzing technologies are described. In Section VI, the most prevalent hybrid fuzzers are classified considering the application, areas, and problem domains. Section VII demonstrates the performance evaluation of hybrid fuzzers. According to the results of this study, we uncover some potential problems that require further studies in Section VIII. Finally, in Section IX, we present a summary of this study.

II. REVIEW METHODOLOGY

We adopted a systematic approach originally promoted by Keele *et al.* [37] and Kitchenham [38], which first led to performing a comprehensive analysis of the hybrid fuzzing studies. In the following sections, we will detail our review approach, research questions, data collection, and publication selection criteria.

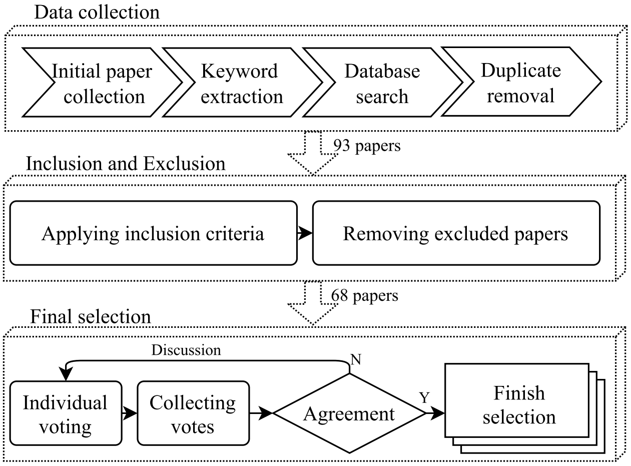


FIGURE 1. **Overview of publication selection criteria.**

A. RESEARCH QUESTIONS

To explain hybrid fuzzing techniques, we will address five research questions:

Q1 What is the trend of studies in the field of hybrid fuzzing?
Q2 Which studies have been introduced in hybrid fuzzing methods over the years?
Q3 What technical features of fuzzing and SE/CE have been applied in innovative solutions proposals?
Q4 Which datasets are utilized for evaluation, and what is the capacity of software vulnerability detection for the existing hybrid fuzzing tools?
Q5 What are the potential research prospects or directions in the future?

Q1 is discussed in Section II-C, and the aim of this question is to determine whether the field of hybrid testing techniques is increasing through the analysis of publication trends. Q2, which is addressed in Section V, provides a better insight into how these studies proposed solutions for hybrid techniques, including fuzzing and SE/CE. Considering Q3, we will explore technical aspects of both methods that have been used in hybrid fuzzing proposals discussed in Sections V and Section VI. Regarding Q4, we will demonstrate the performance and comparison of existing hybrid testing tools, presented in Section VII. Finally, depending on the answers to the last questions, considering Q5, we will provide our viewpoint in identifying unresolved issues and research possibilities that can be tackled in the future, which is provided in Section VIII.

B. PUBLICATION SELECTION CRITERIA

Several studies were considered to present a comprehensive research study that covers all publications associated with hybrid fuzzing. Particularly, to make the hybrid fuzzing survey more relevant, we collected more than 128 studies published from March 2010 to December 2020 from various sources. Thus, after the data screening process, we obtained 49 unique studies as part of our survey. An overview of our study's selection methodology is illustrated in Figure 1. Publications were selected in three steps.

1) DATA COLLECTION

Initially, the data collection began with searching for well-known bibliographical databases such as *Springer Online Library, Internet Society, IEEE Xplore, ACM Digital Library,* and *USENIX.* We gathered articles that included either one or more of the terms *"hybrid fuzzing," "hybrid testing," "SE/CE and fuzzing",* and *"hybrid fuzzer"* in their abstracts, titles, or keywords. Thereafter, we read the selected study, considering cases where the relevancy could not be determined by the abstract. Additionally, these collected materials were considered as main studies [38]. Figure 2 (a) illustrates the bibliographical distribution of the collected main literature. The subsequent step in the data collection was to delete the duplicated manuscripts. For example, these manuscripts can be (1) various variants of the same manuscript such as the expanded journal edition of a conference article or (2) the same manuscript in multiple bibliographic databases. We obtained only 93 unique publications.

2) INCLUSION AND EXCLUSION

To verify that the publications in the classification were in the software testing area and were related to hybrid fuzzing, for each article, we used these inclusion criteria [37].

- Is the study published in the English language?
- Is the publication linked to software analysis or vulnerability detection?
- Does the publication provide a contribution to SE, CE, or fuzzing?

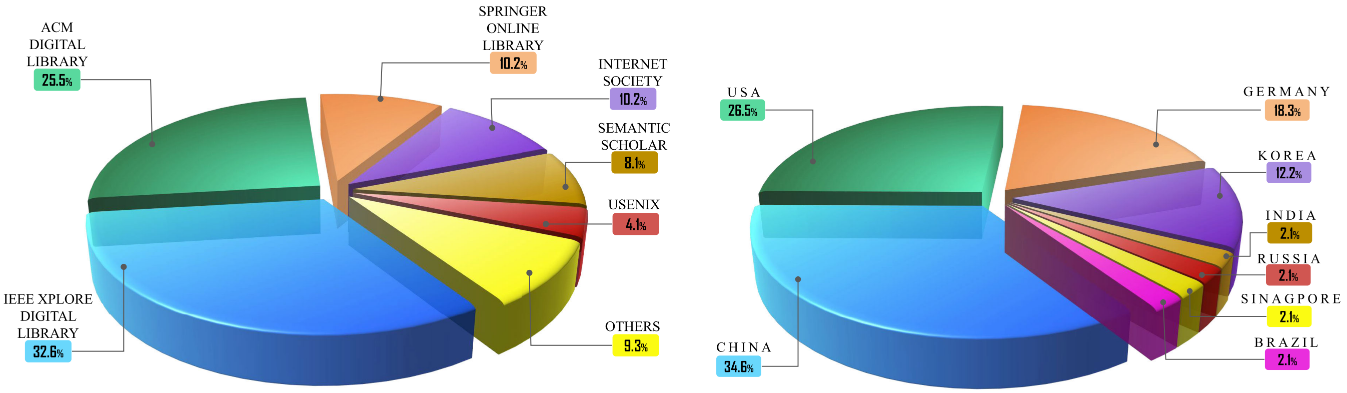
If we do not receive the answers to the above questions from the *"Abstract"* of a study, each author has to read the study's *"Introduction", "Implementation",* and *"Methodology"* sections. We exclusively selected publications that answered "YES" to all the questions above. Otherwise, we removed it from our database. Moreover, studies that poorly clarify the key objective of the suggested analysis were excluded. Finally, the studies that did not include clear results and reasonable comparisons were excluded. After the inclusion and exclusion step, we were left with 68 publications in our database.

3) FINAL SELECTION

Considering this step, all the authors read the articles and sorted them out based on the following criteria:

- A common area of contributions.
- Introduction of an innovative approach.
- An innovative solution to the weaknesses of fuzzing.
- A creative solution to the SE or CE issues.

The final article selection was determined by a vote of all authors. After the authors' voting process, 49 unique publications matched our selection criteria remaining. We are



(a) Bibliographic and regional distribution of the collected literature.

(b) Regional distribution of papers

FIGURE 2. Bibliographic and regional distribution of collected literature.

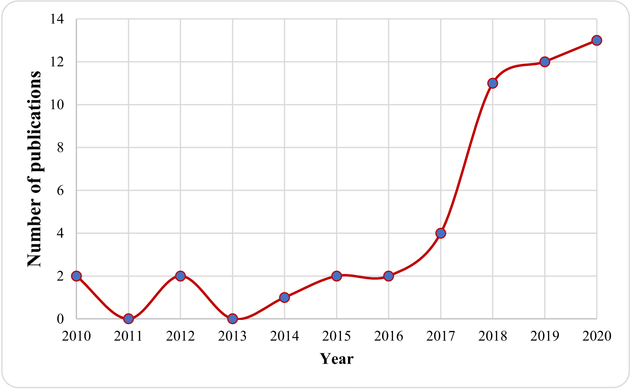


FIGURE 3. Number of published studies per year.

TABLE 1. Top venues on hybrid fuzzing.

Venue	Papers
IEEE Symposium on Security and Privacy **(S&P)**	8
International Conference on Software Engineering **(ICSE)**	6
Network and Distributed System Security Symposium **(NDSS)**	5
USENIX Security Symposium **(USENIX SEC.)**	3
ACM Conf. on Computer and Communications Security **(CCS)**	3

assured that the general patterns are correct in this study, and they give a decent description of hybrid fuzzing techniques.

C. SUMMARY OF FINDINGS

To answer Q1, we summarize our central reviews considering the publishing venues, publication trends, and geographical affiliations of researchers of hybrid fuzzing in this subsection.

1) TRENDS IN PUBLICATIONS

Figure 3 shows the number of publications on hybrid fuzzing from March 2010 to December 2020. As can be observed in the figure, since 2016, there has been a considerable growth in the number of publications on hybrid fuzzing techniques. After the hybrid testing approach won the DARPA CGC, the popularity of this technique has been rapidly expanded in the field of computer security, which led to the publication of more papers.

2) PUBLISHER VENUES

The expansion of this technique inspired scholars to publish 49 studies in 21 distinct venues, which increased the possibility of finding a broad literature on this subject. Furthermore, this technique is valued because of its practicality and reliability in multiple testing by the audience. Considering the different types of venues, the majority of publications were presented at conferences (65%), in journals (29%), workshops (6%), and technical reports. Table 1 illustrates the top venues in which at least three hybrid fuzzing-related studies were presented.

3) DISTRIBUTION OF PUBLICATIONS BY REGIONS

We linked the country of origin of each primary study to the first coauthor's affiliate region. Regarding the geographical distribution of the publications, 49 primary studies stemmed from eight different countries. China, the USA, Germany, and South Korea were the top countries to represent the studies as shown in Figure 2 (b). Considering the data, 28.6% of the publications belonged to American scholars, 18.3% belonged to Europeans, and 53.1% were authored by Asians.

III. VULNERABILITY DETECTION METHODS

We include a brief explanation of software vulnerability detection approaches, including static analysis, dynamic analysis, fuzzing, SE, CE, and hybrid fuzzing in this section. Subsequently, we explain the pros and cons of each strategy.

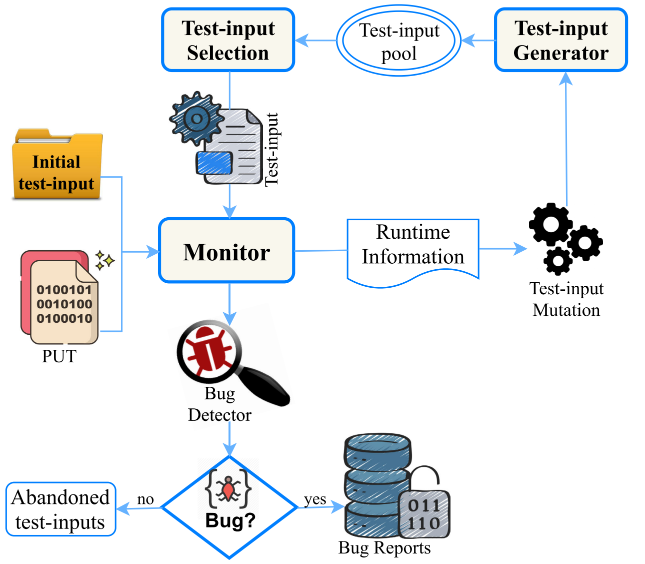


FIGURE 4. **Overall workflow of traditional fuzzing.**

A. STATIC ANALYSIS

Static analysis is a software testing method utilized for identifying possible software flaws in the PUT without running the source code. This type of analysis allows a user to promptly identify where the bug exists in the code based on the applied rules. The benefit of using static analysis is its speed and the decreased cost incurred to fix bugs. This provides advantages over dynamic analysis that fails to report vulnerabilities without substantial effort. Through parsing the code and creating an "Abstract Syntax Tree" (AST) of the software, static analysis is typically implemented. Moreover, this analysis is also useful for identifying coding issues, such as buffer overflows, and format string vulnerabilities [39].

Despite the benefits, there are a few limitations of static analysis. Depending on the employed rules, static analysis may only be workable or practical in specific situations. Another drawback of static analysis is that it can provide false positives and false negatives.

B. DYNAMIC ANALYSIS

To address the drawbacks of static analysis, dynamic analysis is introduced. Unlike static analysis, dynamic analysis detects software vulnerabilities while executing a program [40]. Specifically, dynamic analysis techniques can effectively find software vulnerabilities by analyzing the running states and evaluating the runtime information. The benefit of dynamic analysis is high precision because it determines bugs or defects with extreme precision. However, it has some disadvantages such as slow speed, high technological criteria for testers, and low stability. The following section will discuss each dynamic analysis method, including fuzzing, SE, CE, and hybrid fuzzing, in detail.

1) FUZZING

Barton Miller, a professor at the University of Wisconsin, first introduced the term fuzzing in 1988. Considering the world of cybersecurity, fuzzing is an automated testing technique that plays a vital role in the detection of program bugs by feeding test-inputs to the PUT and analyzing the execution states [41]. Takanen *et al.* [42] provided detailed information regarding fuzzing techniques in their book with rich statistical data and comprehensive case studies. Depending on whether test-inputs are produced from scratch or through the modification of existing ones, fuzzers can be generation- or mutation-based fuzzers. Furthermore, fuzzers can also be classified as whitebox, greybox, or black-box fuzzers. For instance, a whitebox fuzzer is proficient at monitoring the execution path and solving complex constraints by using heavyweight program analysis [43]–[46]. On the contrary, greybox fuzzing tools [47]–[50] utilize lightweight program analysis to increase the code coverage. For instance, the highly effective *Google ClusterFuzz* has detected more than 25000 vulnerabilities in Google and more than 22500 vulnerabilities in over 340 projects, as of September 2020 [8]. Black-box fuzzers only observe the behavior of input/output execution [51]–[53] and do not require the PUT's source code. Although the fuzzing technique is considered one of the efficient approaches to expose program bugs, fuzzers often struggle to obtain complex path conditions in the PUTs; hence, code coverage performance is usually low. Despite that, fuzzing techniques alone are insufficient for detecting complete vulnerability threats in a program.

The overall workflow of the traditional fuzzing technique is illustrated in Figure 4. The fuzzing method consists of four major phases: test-input generation, execution of the target program, monitoring, and vulnerability detection. Determining how to generate highly efficient test-inputs is a vital challenge of fuzzing techniques.

2) SYMBOLIC EXECUTION

Another productive software testing technique is SE, often referred to as whitebox fuzzing. This approach was proposed in the mid 1970's to check for violations in software programs [54], [55]. SE has gained wide attention amongst a heterogeneous audience since DARPA established a CGC competition in 2013. The challenge pertains to the detection of software bugs, exploitation, and patching [56].

Technically, SE is an effective way to test lightweight programs and cover all execution paths in a PUT. However, users of this technique often suffer from path explosion issues during the testing of heavyweight programs.

Overall test-input generation processes of SE are demonstrated in Algorithm 1. The SE takes the $\mathbb{PUT}$ and $\mathbb{I}_{\mathbb{TI}}$ initial test-input as the input. It outputs the new test-inputs by solving the constraints. First, SE loads the $\mathbb{PUT}$ and constructs the CFG, and extracts all the basic block addresses. Thereafter, the $\mathbb{PUT}$ is executed dynamically with the $\mathbb{I}_{\mathbb{TI}}$ initial test-input, and the covered basic block addresses are identified

Algorithm 1 Test-Input Generation of Symbolic Execution

```
Input: $\mathbb{PUT}$ (Program Under Test)
Input: $\mathbb{I}_{\mathbb{TI}}$ (Initial Test-Input)
$T_P \leftarrow Load(\mathbb{PUT})$
CFG $\leftarrow cfgConstruction(\mathbb{PUT})$
AllBBaddr $\leftarrow ExtractAllBBaddr$(CFG)
Path $\leftarrow$ DynExecution($\mathbb{PUT}$, $\mathbb{I}_{\mathbb{TI}}$)
CoveredBBaddr $\leftarrow$ PathMapping (Path, CFG)
foreach BB $\in$ AllBBaddr do
    if BB $\notin$ CoveredBBaddr do
        $CONS \leftarrow$ ConstraintCollection(BB)
        $\mathbb{N}_{\mathbb{TI}} \leftarrow$ Solver($CONS$)
    else
        Continue
    end if
end for
Output: $\mathbb{N}_{\mathbb{TI}}$ (New Test-Input)
```

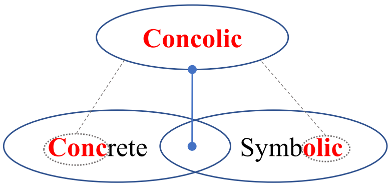


FIGURE 5. Concolic execution model.

by mapping the path to the CFG. If the *BB* basic block in the *AllBBaddr* does not match the *BB* in *CoveredBBaddr*, it will put it into *CONS* for further processing. Finally, a new effective test-input $\mathbb{N}_{\mathbb{TI}}$ is produced by solving the constraints collect in $\mathbb{N}_{\mathbb{TI}}$ using the Z3 solver.

3) CONCOLIC EXECUTION

Concolic execution [57]–[59] is a classical SE that considers program variables as symbolic variables and executes the PUT with concrete execution. It mainly focuses on revealing software vulnerabilities instead of proving the correctness of the application. It functionally forks the symbolic interpreter into true and false nodes when it arrives in a conditional node. The symbolic interpreter aims to generate a path formula for each node encountered throughout the program execution. A path formula can be satisfied when there is a concrete test-input that performs the intended path. The concrete test-inputs are generated by an SMT solver [60] to provide a path formula solution.

Concolic testing combines concrete and symbolic tests, wherein both the SE and the concrete execution are carried out simultaneously (Figure 5). Concrete execution can allow for a decrease in the ambiguity of the symbolic constraints. Studies on the function of CE are provided in [61]. However, as compared to black-box and greybox fuzzing approaches, the CE is slow because it involves an instrumentation process and examination of each branch of the program [59].

TABLE 2. Performance comparison of testing techniques.

Testing Approaches	Usage	Accuracy	Bug detection	Scalability
Static	Easy	Low	Moderate	Good
Dynamic	Hard	High	Easy	Good
SE/CE	Hard	High	Moderate	Bad
Fuzzing	Easy	Moderate	Easy	Good
Hybrid Fuzzing	Easy	Highest	Easy	Best

4) HYBRID FUZZING

Research in the field of hybrid fuzzing has gained popularity and provides a significant contribution to bug detection. To illustrate this, the Shellphish team [62] won the DARPA CGC in 2016 by utilizing a hybrid fuzzing technique. Hybrid fuzzing involves an extra CE or SE engine that reexamines the covered paths by a fuzzer, resolves the path constraints, and exposes the uncovered paths compared to plain fuzzing. The combination of fuzzing and SE/CE allows precise results and makes the software test process easier.

Hybrid fuzzers [6], [63]–[65] aims to take advantage of those techniques. First, it begins with carrying out fuzzing before switching to SE automatically to reveal an uncovered path. Second, the hybrid fuzzer returns to fuzzing. Fuzzing is capable of quickly performing shallow program paths, whereas SE provides the advantage of covering more complex program paths. However, the scalability of symbolic techniques is low. Thus, more recent tools like Driller [6] or QSYM [64] prefer concolic testing approach.

Figure 6 indicates the overall workflow of hybrid fuzzing. The entire system includes three key components: fuzzer, CE engine, and a coordinator. The coordinator component is a middleware that controls the fuzzing and CE techniques. It has three main tasks. First, it monitors the fuzzer to determine when to start the CE engine; second, it prepares the running environment for CE; third, it selects and filters test-inputs that run between the fuzzer and CE. Initially, the coordinator's test-input selection component determines the test-input file that queue of fuzzer should be sent to the CE engine first. Before performing CE, the coordinator needs to sort out the test-inputs in the fuzzer's queue according to their efficiency.

C. COMPARING THE PERFORMANCE OF VULNERABILITY DETECTION METHODS

In this section, we compare the performance of each software testing technique. Figure 7 depicts the code coverage process of each dynamic analysis technique in CFG. The SE/CE can cover and analyze all basic blocks in the target program. However, it is not technically scalable because of the tremendous number of paths in the PUT. The path's explosion problem is a significant challenge for the users of the SE/CE. Although these approaches have a smart solution for complex constraints (SMT solver), they are not highly effective when testing heavyweight applications. For instance, there are more

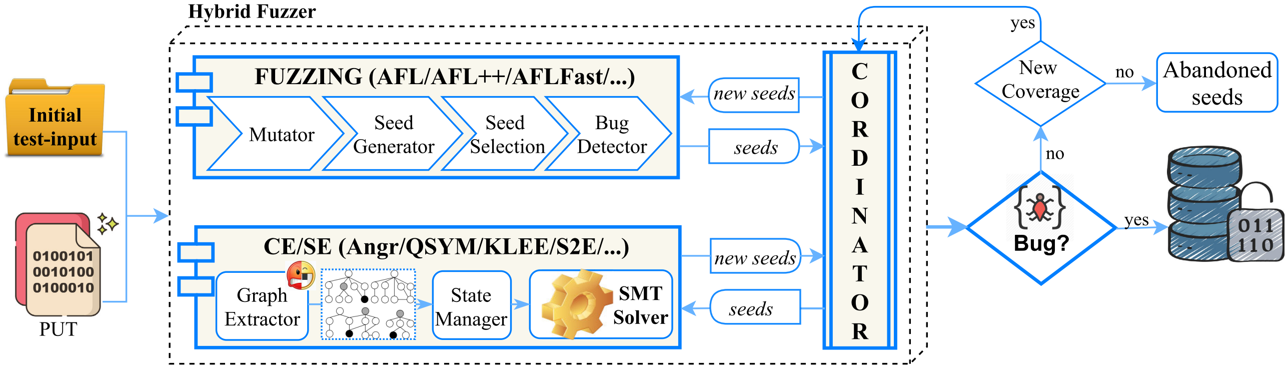


FIGURE 6. Hybrid fuzzing technique's high-level architecture.

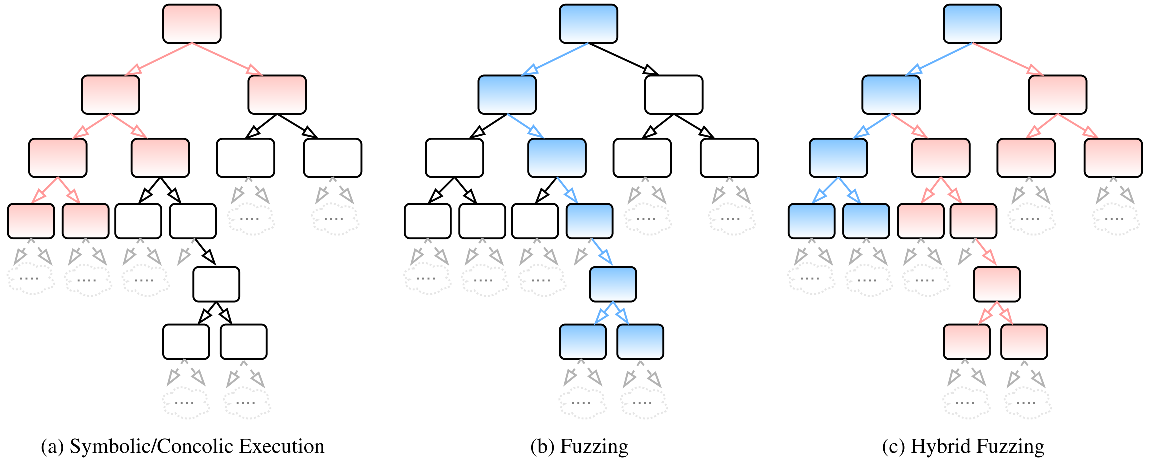


FIGURE 7. Code coverage comparison of dynamic analysis techniques in CFG.

than 15 K LOC in a *grep* program which has 8,000 basic blocks in the execution path. Thus, it is not feasible to cover all the branches within a sufficient time [66]. Fuzzing tools are quicker than SE/CE, allowing them to explore deeply hidden PUT branches more easily. However, the fuzzer is not smart and often produces useless test-inputs that cannot explore new paths. Consequently, this leads the fuzzer to be more time-consuming, and it decreases its effectiveness. On the contrary, as shown in Figure 7c, the hybrid fuzzer can cover all branches quickly. Generally, the cardinal idea behind the hybrid fuzzer is to combine both techniques to eliminate each other's shortcomings and achieve high code coverage results.

Table 2 shows the performance comparison of different software testing approaches. The table shows that the hybrid fuzzing methods have a higher performance than other methods. In addition, because many software testing competitions are currently mainly dominated by hybrid fuzzing tools, it is observed that the use of this technique is highly effective.

Figure 8 demonstrates comparisons of the effectiveness of each method's execution rate and code coverage. Although SE/CE has a high code-coverage, one of its main drawbacks is its slow execution rate. Moreover, fuzzing and static analyzing methods are fast in bug hunting; nevertheless, they have low code coverage results. The low code coverage result makes it inefficient in detecting all program bugs. Although the hybrid fuzzer does not have high code-coverage compared to the SE/CE, this method is the most effective owing to its speed and ability to test heavyweight programs.

IV. TAXONOMY OF HYBRID FUZZER

In this section, we provide the taxonomy of hybrid fuzzers based on execution and test-input generation. Figure 9 illustrates the taxonomy of the hybrid fuzzer. We have divided the hybrid fuzzing approaches into two groups: test-input generation and execution monitoring.

A. TEST-INPUT GENERATION

Generating meaningful test-inputs is vital for automated program testing. Considering the fuzzing approach, test-inputs are produced through a generation- or mutation-based approach [29], [50]. Regarding the mutation-based method, the generation of test-inputs is based on the mutation of test-input files, whereas the generation-based method requires

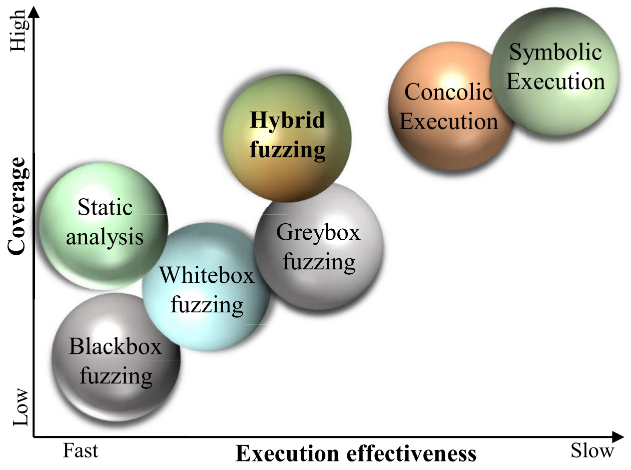


FIGURE 8. Comparison of testing approaches in the execution process and code coverage.

that test-inputs are produced depending on the model of the file format. The main challenge is the production of efficient test-inputs that explore the hard-to-reach program branches.

1) GENERATION-BASED

Considering the generation-based method, the fuzzer requires knowledge of the test-input. Based on the configuration file test-input files are generated. The given file format information enables generation-based fuzzers to generate test-inputs that provide the capability to test the validity of programs more quickly and exploring the deeper basic blocks of PUT.

a: SMART-INPUT BASED HYBRID FUZZER

The recognition of highly efficient test-inputs that can solve complex constraints and cover more paths and vulnerabilities are critical challenges in hybrid fuzzing. Therefore, incorporating machine learning technology into hybrid fuzzing techniques provides a possibility to improve the performance of the software testing process. The advancement of machine learning technology adds a significant contribution to the test-input generation [67]–[70]. Godefroid *et al.* [67] first used Machine Learning (ML) approaches (such as the Neural Network) to learn test-inputs' grammar and produced format-fulfilled test-input files using the learned grammar [71]. The usage of the ML engine models (including feature extraction, training, and prediction) produce powerful files as test-inputs. Specifically, a smart-input-based hybrid fuzzer (SIHF) combines fuzzing, CE, and a smart-input selection paradigm that functions in accordance with machine learning. Thereafter, fuzzing begins to run a program with predefined or empty test-inputs. Subsequently, the ML engine extracts PUT's features and test-inputs to model the coverage gains. These features allow the prediction of the code coverage that the unknown test-inputs can produce. Thereafter, the CE engine retrieves the possible practical test-inputs from the prior stage and generates test-inputs. Subsequently, the fuzzer utilizes these effective test-inputs and their produced mutants to continuously explore the PUT. Because the fuzzing persists, the prediction model is enhanced to provide more accurate predictions.

2) MUTATION-BASED

Several hybrid fuzzing tools use a mutation approach because of its ability to generate large test-inputs quickly and easily. Considering this process, test-inputs are produced by altering parts of the test-inputs, and the bytes of the test-inputs are changed through random or many unique values, which appears ineffective. Another critical obstacle is determining the position adjustments and special value utilized in the modification. Based on the techniques for exploring the PUT, mutation-based hybrid fuzzers can be categorized into coverage-guided and directed hybrid fuzzing.

a: DIRECTED HYBRID FUZZER

A directed hybrid fuzzer (DHF) is intended to generate test-inputs that explore the specified suspicious code and paths of the PUT. Typically, DHFs primarily accomplish the task of reaching the specified locations by integrating both dynamic and static analyses. Static analysis involves determining the path in the basic block located in the function call chain via the control flow graph analysis and function call relationship. Considering the DHF, the dynamic analysis process involves the interconnection between the fuzzer and SE/CE. Each testing tool exchanges test-inputs with different preferences and append test-inputs into the queue.

b: COVERAGE–GUIDED HYBRID FUZZER

Nowadays, coverage–guided hybrid fuzzers (CGHF) are practically the most widespread hybrid fuzzers. This type of fuzzer aims to produce productive test-inputs, which involves a more comprehensive test to detect as many software vulnerabilities as possible. This indicates that this technique aims to analyze the whole target program's code, making the mutated test-inputs reachable to uncovered code branches. Technically, CGHFs mostly reach high code coverage by uniting both the coverage-guided fuzzing method and the CE engine. Regarding this method, coverage-guided fuzzing exchanges already-covered branch information with the CE engine. For instance, AFL intercepts transitions between branches and retains this data in a Bitmap. Thereafter, the CE engine receives a test-input in the queue to reveal further uncovered paths. The CE only forks previously uncovered branches. According to this hybrid testing method, the CE engine detects hard-to-reach bugs and generates a new test-input for further fuzzing.

B. EXECUTION MONITORING

1) WHITEBOX-BASED HYBRID FUZZER

Whitebox based hybrid fuzzer (WBHF) associates fuzzing with CE to enhance software testing performance by detecting deeply-located bugs in binaries, reaching high code

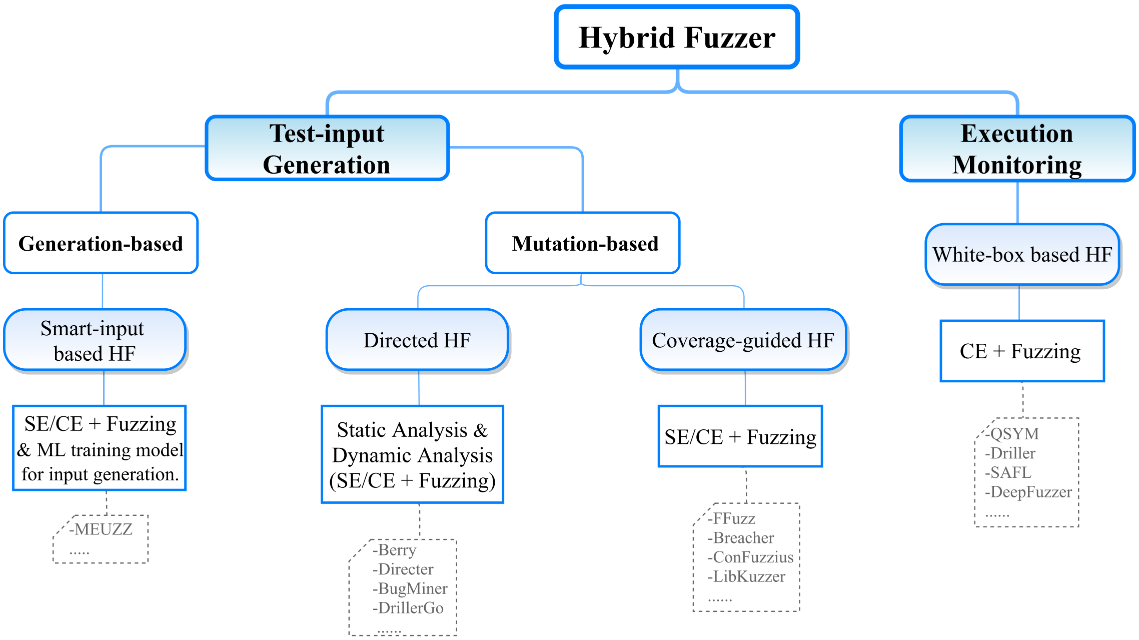


FIGURE 9. Taxonomy of hybrid fuzzers based on test-input production and PUT exploration methods.

coverage. Most of the WBHFs utilize a common off-the-shelf fuzzer (AFL) [47] and a top binary analyzer tool (angr) [56]. The testing process begins with the generation of test-inputs by the fuzzer. Thereafter, coverage information that informs the user of the number of times each basic block is covered during the testing is gathered. The fuzzer shares the test-inputs with CE to discover uncovered paths. To simplify the path constraints, the CE engine simultaneously explores programs on both the symbolic and concrete values. The CE produces effective test-inputs for paths that have not been covered by solving path constraints and returns the already produced test-inputs to the fuzzer. By getting the new test-inputs from the CE, the fuzzer can dive deeply into the PUT and increase the code coverage.

C. SELECTION OF A SUITABLE HYBRID FUZZER

Depending on how bugs are detected or triggered, they can be "hidden" or "shallow" bugs. Considering the early execution stage, the bugs that crash the PUT are termed "shallow" bugs. In contrast, the bugs located deep in the PUT and challenging to explore are termed "hidden" vulnerabilities. Despite their existence, there is no exact method to detect these bugs; therefore, the fuzzer is widely utilized to discover bugs in the software. Software testers can opt to use the fuzzer based on the following aspects: 1) the requirements for the testing process (e.g., time efficiency) and 2) the type of PUT. For example, if the target program's test-input format is complex or unique, selecting a SIHF is suggested. Considering other situations, test requirements should be considered by software testers. If the cardinal purpose of the test is efficiency rather than high-quality results or accuracy, and if the testers want to detect the specified vulnerable code and patch it, DHFs are the best option. The DHS, for example, is preferred if a target program has not been previously examined and testers aim to detect or eliminate "shallow" vulnerabilities rapidly. On the contrary, if testers are interested in the accuracy of the results and aim to reach high code coverage, it is typically more appropriate to use CGHFs or WBHFs. Despite the possibilities of these techniques, they are very expensive, considering the time and consumption of resources.

V. STATE-OF-THE-ART IN HYBRID FUZZING

To answer Q2, we present the genealogy of hybrid fuzzers in Figure 10, and answer Q3 in this section by outlining the key contributions of the suggested hybrid fuzzing approaches. We discuss studies on HF in Section V-A based on instrumentation and test-input selection, test-input generation in Section V-B, and execution monitoring in Section V-C.

A detailed description of the hybrid fuzzing approaches is presented in Table 3 (page 11). Each technique is summarized based on the implementation process of the hybrid fuzzer. The requirement (*REQ*) column shows whether a hybrid fuzzer requires an open-source program or binary. The *Preprocess* column shows whether a hybrid tool utilizes instrumentation and test-input selection methods. The *Key techniques* column indicates the bug-hunting methods (including static analyzer, SE engine, CE engine, fuzzing tool, and taint analyzer) are leveraged by the hybrid fuzzer. The *Test-input generation* column indicates whether a hybrid fuzzer uses mutation-, generation-, or constraint-based approach to produce test-inputs. The *Schedule* column shows whether a hybrid fuzzer uses scheduling algorithms. The *Scalability* column indicates whether a hybrid fuzzer tests only real-world programs●, experimental programs○, or both ◑. The *Platform* column indicates the operating system, Linux (L) or Windows (W), necessary for running the hybrid fuzzer. The final column shows whether a hybrid fuzzer is open-sourced or not.

Figure 10 (page 15) demonstrates the genealogy of the introduced hybrid fuzzers over the last decade. Each node in each row indicates a hybrid fuzzing technique that is introduced in the same year. The colored shapes inside the node indicate the fuzzing tools in the figure above utilized in the hybrid fuzzers. A solid arrow shows the relationship between the hybrid fuzzing approaches.

A. PREPROCESSING

Before the first fuzz iteration, some hybrid fuzzers alter the initial collection of fuzzing configurations. Preprocessing is widely used for the instrumentation of the PUT and allows testers to eradicate possible redundant configurations. Preprocessing methods are mainly utilized in DHF to explore the target code as outlined in Section V-B2

1) INSTRUMENTATION

Considering fuzzing, the goal of instrumenting the program [48], [49] is to fuzz the contents of the memory at runtime and collect execution feedbacks. The amount of the execution data obtained contributes to determining a hybrid fuzzer's status. While there are several methods for obtaining information regarding the PUT internals such as "processor traces or system call usage" [72], [73], instrumenting methods are mainly considered as the best ways to obtain the most valuable feedback. Instrumentation of the PUT can be either dynamic or static.

The static instrumentation process occurs before the target program executes, whereas the dynamic instrumentation occurs during the target program execution. We will discuss static and dynamic instrumentations in this subsection. Static instrumentation is mainly executed on either intermediate or source code.

Contrary to dynamic instrumentation, static instrumentation requires less time because it occurs before execution. If the target program is dependent on several libraries, separate instrumentation of libraries is required. This typically indicates recompiling of these libraries with the same instrumentation. Apart from the instrumentation based on the source code, binary-level static instrumentation, such as binary rewriting tools [74], [75], has been implemented by researchers.

Dynamic instrumentation can instrument libraries without difficulties while the instrumentation is carried out at runtime, regardless of higher overhead, compared to the static instrumentation. There are many highly prominent dynamic instrumentation techniques, including DynamoRIO [76], Valgrind [77], QEMU [78] Pin [31], and DynInst [79]. Instrumentation is applied at compile-time for open-source programs, whereas it is applied at runtime through an adapted QEMU [78] for target binaries. If the source code of the PUT is available, hybrid fuzzers that are based on the AFL employ static instrumentation with a clang compiler. Otherwise, they employ dynamic instrumentation with the assistance of QEMU [78]. For instance, QSYM [64], a fast CE engine, utilizes AFL to instrument the target program and fuzz. Furthermore, QSYM relies on the PIN to explore the target binary and to choose the basic blocks for the SE engine. There are hybrid fuzzers that approve LLVM [80] instrumentation, which has been designed to provide the user with quick and easy access to the code that is sanitized, analyzed, and injected. For example, Wildfire [81] introduced a novel open-source compositional fuzzing technique that detects bugs in a C program by fuzzing separated functions and evaluating the exploitation feasibility of these bugs by utilizing a targeted SE engine. Wildfire supported LLVM 3.4-6.0, and it was developed as an enhancement of Macke [82], an analysis technique for C programs. Wildfire employs AFL to fuzz isolated functions, and KLEE22 [83] is used to identify the possibility of vulnerabilities. This study saves approximately 10% more than other common hybrid fuzzers.

KLUZZER [84] suggests a novel whitebox fuzzer that applies the "whole-program-llvm" [85] to generate LLVM IR rather than the LLVM clang compiler. LLVM 6.0 version is a recommended option for use when the user analyzes target programs via KLUZZER. Another instrumentation-based hybrid fuzzer, LibKluzzer [86], [87], combines the coverage-guided fuzzing techniques known as LibFuzzer [49] and whitebox fuzzer KLUZZER. It uses the LLVM compiler interface which compiles the PUT to build the LLVM IR by employing *Clang*. The compilation, among other advantages, provides instrumentation for code coverage and connects with the LibFuzzer. Lastly, to execute the whitebox fuzzing, the LLVM bitcode is sent to the KLUZZER.

Another hybrid fuzzing tool is Map2check [88] which consists of fuzzing and SE. This tool also utilizes LLVM v6.0 [89] compiler techniques to examine C language-based programs. Similar to the abovementioned techniques, to first simplify the code, this tool converts the C-code using the LLVM compiler and feeds the testing approaches with instrumented binary. KleeFL [90] is another hybrid fuzzing tool that combines KLEE and AFL, where SE generates additional test-inputs to direct AFL's exploration. First, KleeFL gives the task of code exploration to the SE engine. By enacting simple and fast changes on the resulting test-inputs, the action leads to high-performance fuzzers subsequently. KleeFL exhibits limitations in exploring heavyweight programs with complex libraries owing to the well-known SE restriction.

2) TEST-INPUT SELECTION

Hybrid fuzzers obtain a variety of fuzz settings to monitor the fuzzing algorithm's behavior. Unfortunately, certain fuzz configuration options, including test-inputs for mutation-based hybrid fuzzers, possess broad value domains. Considering an MP3 player as an example, if the tester analyzes the MP3 player, the MP3 files represent the test-input. A number of valid MP3 files are available, leaving the tester confused about which test-input to select for fuzzing. The decline in the size of the initial test-input queue is classified as the test-input selection issue [91]. There are many methods to tackle the problem associated with test-input selection. One of the effective ways is to identify a minimum number of test-inputs that expand a code coverage parameter, such as edge and node coverages. For instance, the present collection of C configurations comprises two test-inputs T_{I1} and T_{I2} that include the following addresses of the PUT: ($T_{I1} = (10, 20)$, $T_{I2} = (20, 30)$). If there is a third test-input $T_{I3} = (10, 20, 30)$ that runs the same speed as T_{I1} and T_{I2}, it is preferable to fuzz the PUT with T_{I3} test-input rather than T_{I1} and T_{I2} because it intuitively checks more codes. The AFL-based hybrid fuzzers encourage this *minset* that the test-input selection algorithm is dependent on node coverage with a logarithmic counter. The cardinal idea of this choice is to encourage node counts to be considered distinct only if they vary in terms of magnitude orders.

B. TEST-INPUT GENERATION

Generating efficient test-inputs to hunt the bug is very important in software testing. Test-input file quality can significantly affect the outcome of the fuzzing. Therefore, a significant problem is determining how to produce sufficient test-inputs. As mentioned in Section IV, there are two approaches in the test-input generation for fuzzers: a mutation-based approach which produces test-inputs according to the random mutation of the test-input files, or the use of predefined mutation strategies and generation-based approach, which produces test-input files from a specification.

1) GENERATION-BASED HYBRID FUZZERS

As stated above, the advancement of ML technology greatly contributes to the test-input generation of hybrid fuzzer. Test-input selection and generation based on ML techniques offer an opportunity to select test-input rules without designing manually, testing, and deciding the impracticality when the number of data to be evaluated is tremendous [96].

For instance, MEUZZ [96] is the first hybrid fuzzer that was constructed on machine learning techniques. It analyzes the test-inputs previously encountered, and it identifies which test-inputs can effectively analyze uncovered branches in the PUT being tested based on the training. Additionally, MEUZZ employs UBSAN [147] to instrument the target program by first beginning to fuzz the PUT with predefined and empty seeds. Thereafter, it extracts features from the PUT and test-inputs to model the coverage gains. These features enable forecasting of the coverage that the uncertain test-inputs can provide. Subsequently, CE attains the theoretically robust test-inputs from the previous step and generates mutated test-inputs. Considering the next stage, MEUZZ directs the fuzzer to employ these smart test-inputs and their produced mutants via the genetic algorithm to continuously test the PUT. Because fuzzing persists, the model becomes reinforced, leading to a more accurate prediction.

Another hybrid fuzzer suggested by Pak [39] employs SE to collect as many specific constraints as possible in the user-defined resource limit and solves collected constraints to produce the "random" test-inputs for the fuzzer. The generation module of the test-input in Pak's hybrid fuzzer is based on the "Parma Polyhedra Library" (PPL) [148], and advanced the test-input generation module that assists in producing a pre-configured number of test-inputs. Although this technique provides sufficient diversity within the test-inputs rather than the manual entry, it highly depends on the fuzzer to thoroughly evaluate all the paths. The Eclipser [137], a hybrid fuzzer that has been recently presented, proposes a new path-based test-input generation algorithm. To produce high coverage test-inputs, it employs a lightweight instrumentation tool. The Eclipser incorporates fuzzing and CE techniques such as QSYM or Driller. Contrary to the QSYM or Driller, the Eclipser utilizes greybox CE rather than the traditional CE. This indicates that greybox CE is considered a lightweight version of CE and allows solving relatively simple branch conditions.

2) MUTATION-BASED HYBRID FUZZERS

Mutation-based hybrid fuzzers can be categorized as Directed Hybrid Fuzzing (DHF) or Coverage-Guided Hybrid Fuzzing (CGHF) based on the strategies of the programs' exploration.

a: DIRECTED HYBRID FUZZER

Considering the DHF, static analysis approaches provide ample support to detect vulnerability in the target code. Several techniques of static analysis may also be applied to collect control flow data. For example, the depth of the direction can be utilized as another guide in directed testing [50]. Berry [98] suggests a "Sequence Directed Hybrid Fuzzing" (SDHF) methodology that guides both the fuzzer and CE engine using the program's improved statement sequences of the target. SDHF consists of static and dynamic analysis. Considering the first stage, SDHF takes the target program and sequence as inputs before initially mapping the target sequence statements to each branch, and it measures the "Enhanced Target Sequence" (ETS). Thereafter, the SDHF instruments the target program to obtain data at runtime. Regarding the second stage, the fuzzer and CE engine interconnect through test-input synchronization. The fuzzer gets the ETS and the PUT, whereas the CE explores the target program. Therefore, the fuzzer and CE exchange test-inputs to produce test-inputs that explore the specified target. At the final stage, test-inputs that crash the program are stored in the bug report database.

Another DHF is known as DrillerGo [99] retrieves the vulnerable functions from the CVE [149] description, and it suggests a novel technique known as "Backward Pathfinding" that searches suspected call functions from the CFG. First, the basic block address of a specified function is set as a target location, and until reaching the target destination,

TABLE 3. Overview of the hybrid fuzzing techniques.

Hybrid Fuzzer	REQ		Pre-process		Key techniques					Test-input generation			Schedule	Sca-lability	Platform	
	Source Code	Binary	Instrumentation	Test-input Selection	Static Analysis	Symbolic Execution	Concolic Execution	Fuzzing	Taint Analysis	Mutation based	Generation based	Constraint based	Seed Schedule	Scalability in fuzzing	Linux / Windows	Open-Sourced
Wildfire [92]	✓		✓			✓		✓				✓		●	L	[81]
KLUZZER [84]	✓		✓			✓		✓				✓		●	L	[93]
Map2Check [88]	✓		✓		✓	✓		✓				✓		○	L	[94]
LibKluzzer [86]	✓		✓			✓		✓		✓		✓		○	L	[87]
KleeFL [90]	✓		✓			✓		✓				✓		◑	L	[95]
DeepDiver [65]	✓		✓	✓			✓	✓		✓		✓	✓	◑	L	✗
MEUZZ [96]	✓		✓	✓			✓	✓			✓	✓	✓	●	L	[97]
Berry [98]	✓		✓	✓	✓		✓	✓		✓		✓	✓	◑	L	✗
DrillerGO [99]		✓			✓		✓	✓		✓		✓		○	L	✗
1dVul [100]		✓			✓	✓		✓		✓		✓		○	L	✗
FFuzz [101]		✓	✓		✓	✓		✓		✓		✓	✓	◑	L	[102]
Breacher [103]		✓		✓	✓		✓	✓		✓		✓	✓	◑	L	✗
Driller [6]		✓	✓				✓	✓		✓		✓	✓	○	L	[104]
QSYM [64]	✓	✓	✓	✓			✓	✓		✓		✓	✓	◑	L	[105]
SAFL [106]	✓		✓	✓		✓		✓		✓		✓		●	L	✗
DeepFuzzer [107]	✓		✓	✓		✓		✓		✓		✓	✓	◑	L	[108]
T-Fuzz [109]		✓				✓		✓		✓		✓	✓	◑	L	[110]
Taintscope [44]		✓	✓	✓		✓		✓	✓	✓			✓	●	L/W	✗
RedQueen [111]		✓				✓		✓		✓			✓	◑	L	[112]
Saad et al. [113]	✓				✓			✓	✓	✓		✓		○	L	✗
Zhang et al. [114]		✓	✓			✓		✓		✓		✓		○	L/W	✗
SAVIOR [63]	✓		✓	✓	✓		✓	✓		✓		✓	✓	◑	L	[115]
Gerasimov et al. [116]		✓	✓		✓		✓	✓		✓		✓		○	L	✗
Sword [117]		✓				✓		✓	✓	✓		✓	✓	-	-	✗
Musliner et al. [118]	✓		✓				✓	✓		✓		✓		-	L	✗
DigFuzz [119]		✓			✓		✓	✓		✓		✓	✓	◑	L	✗
Angora [120]	✓		✓					✓	✓	✓				◑	L	[121]
Badger [122]	✓		✓				✓	✓		✓		✓		◑	L	✗
HyDiff [123]	✓		✓	✓	✓	✓		✓		✓		✓		◑	L	[124]
Fangquan et al. [125]		✓	✓		✓	✓		✓		✓		✓		○	L	✗
FAS [126]		✓	✓	✓		✓		✓		✓		✓		○	L	✗
SynFuzz [127]	✓		✓	✓			✓	✓	✓		✓	✓	✓	◑	L	✗
Cyberdyne [72]	✓		✓			✓		✓		✓		✓	✓	◑	L	✗
Pak et al. [39]	✓		✓			✓		✓			✓	✓		●	L	✗
Colossus [128]	✓					✓		✓		✓		✓		●	L	✗
VUzzer [50]		✓		✓	✓			✓	✓	✓			✓	◑	L	[129]
SymFuzz [130]		✓	✓	✓		✓		✓	✓	✓				●	L	[131]
Tinker [132]		✓	✓		✓	✓		✓		✓		✓		○	L	✗
Munch [133]		✓		✓		✓		✓		✓		✓		●	L	[134]
DeepFuzz [135]		✓		✓			✓	✓			✓	✓		●	L	✗
S2F [136]		✓				✓		✓		✓		✓		◑	L	✗
Eclipser [137]		✓	✓	✓			✓	✓			✓	✓	✓	◑	L	[138]
Pangolin [139]		✓					✓	✓		✓		✓		◑	L	✗
Intriguer [140]		✓	✓			✓		✓	✓	✓		✓		◑	L	[141]
AutoDES [142]		✓			✓		✓	✓		✓		✓	✓	○	L	✗
SHFuzz [143]		✓	✓	✓	✓		✓	✓	✓	✓		✓	✓	◑	L	✗
HFL [144]	✓		✓		✓	✓		✓		✓		✓		◑	L	✗
FIoT [145]		✓			✓	✓		✓		✓		✓		○	L	✗
BugMiner [146]	✓		✓		✓			✓		✓		✓		◑	L	✗

it explores the PUT using dynamic analysis that includes the AFL fuzzer and directed CE known as angr [150].

A new approach to boost the efficacy of a directed test-input generation was proposed by 1dVul [100]. It utilizes "a distance-based directed fuzzing", and "dominator-based directed SE" systems [100]. Using binary patches, 1dVul detects one-day software vulnerabilities. To find specified basic blocks in the patched binary, the authors of 1dVul implemented heuristic rules based on BinDiff [151]. The subsequent phase involves estimating the distance between the generated test-input and specified basic block. It prioritizes the test-input nearer to the specified basic block. Each test-input produced by fuzzing is fed to the SE. The SE subsequently generates novel test-inputs that can arrive closest to the destination point and sends them to the fuzzer. If the generated test-input crashes the patched program, 1dVul will explore the original program with that test-input to validate the crashed test-input as true or false. Another innovative DHF named BugMiner [146] boosted fuzzing performance with a target-oriented CE. The authors proposed a novel *BugReportAnalyzer* method that identified and retrieved unsafe functions from CVE using the ML-based NLP [152] technology. Subsequently, the collected data is utilized in static analysis to measure the distance between the PUT's "*main ()*" and unsafe function's addresses. To tackle the path explosion problem, the authors proposed a *Branch-Pruner* that collects all the basic block locations unrelated to the specified function and avoids exploring those basic blocks. Consequently, BugMiner hunts the bug in a short time without path explosion.

b: COVERAGE GUIDED HYBRID FUZZERS

The target programs are executed repetitively in the fuzzing loop. The new path discovery and guidance of the fuzzing process are the main challenges of coverage-guided hybrid fuzzers (CGHF). To detect a deeply hidden bug more quickly, the hybrid fuzzer requires the PUT's path execution information. Considering CGHF, instrumentation tools are applied to collect the PUT's path execution and evaluate the code coverage. KLUZZER [84] is a CGHF that constructed on the KLEE symbolic execution tool, whereas the coverage-based grey-box fuzzer known as LibFuzzer [49] is utilized. This process makes substantial upgrades that help prepare KLUZZER for its usage in hybrid fuzzing. The authors of KLUZZER implemented its own *main()* function instead of connecting LibFuzzer, and before calling the fuzzing target, it allowed the possibility of marking byte array as symbolic, and it performed a standard execution to call back the coverage.

Another CGHF is known as LibKluzzer [86], [87], which achieved good performance in Test-Comp 2020 [153], had two key components, LibFuzzer and KLUZZER were employed as coverage-guided and whitebox fuzzers, respectively. Considering the LibKluzzer, KLUZZER addresses a majority of the KLEE infrastructure such as SMT solver STP [154] and instruments the target program with LLVM. Similar to the LibKluzzer, another CGHF known as Map2Check also uses LibFuzzer and KLEE. To easily explore "shallow" vulnerabilities, Map2Check [88] generates random data as a test-input for C-language programs, and KLEE examines the properties of safety in a new way. Furthermore, Map2Check leverages MetaSMTs such as Yices [155] and Boolector [156] as the SMT solver. The SVCOMP'20 [157] findings demonstrate that Map2Check exhibits high performance in reaching the vulnerable location and in pointing the safety-related properties.

Automated hybrid bug detector tools including FFuzz [101], BREACHER [103], and S2F [136]) are constructed on AFL and a profoundly precise SE known as the S2E. The FFuzz aims to explore the full system, including all kernels and user-spaces. The use of these tools can assist the FFuzz to dive deeper into the binary to trigger the bugs and increase code coverage. The S2E starts to perform the CE engine with an initial test-input file. The engine examines coverage of any symbolic node that has not yet been covered, generating a novel test-input for the new node using the constraint solver. If this process does not work, the symbolic forking on this node is interrupted to prevent an overhead. After exploring all the uncovered nodes by the S2E with that test-input, the newly produced test-inputs are stored in the queue. The BREACHER [103] comprises two key techniques including *"Searcher"* and *"Symbolic PathFinder"* (SPF). To provide support for the fuzzer to dive into deeply hidden code locations that are complicated to cover, the SPF technique is employed. The SPF also contributes considerably by eliminating the path explosion issue. The Searcher provides mutation strategy to the test-input files that are obtained from the distance-based selection approach. Consequently, this process would make the fuzzer reachable to more code locations that were previously unexplored in a given time budget. The S2F [136] proposed a new approach known as the *"Semi Symbolic Fuzz Testing"* to explore deeply hidden bugs. The authors of the S2F implemented the *"Low-Frequency Seed Detector"* approach, which helped to measure the frequency of the test-input files and to split the fuzzer's test-input files into high- and low-frequency test-input groups. Thereafter, it leverages the S2E to resolve only the uncovered basic blocks for eliminating the path-explosion issue. Considering each low-frequency test-input, the S2F is advantageous because it identifies critical basic blocks and analyzes each critical basic block's symbolic state, providing potential solutions. Subsequently, it fixes the test-inputs generated by the mutator. Angora [120] reduces the cost of software testing by generating test-inputs that are executed with lightweight instrumentation. Moreover, it strengthens bug hunting and the concept of "hot bytes" by leveraging taint analysis. To resolve path constraints effectively, Angora employs "gradient-descent-based search algorithm" and taint-tracking approaches. Another CGHF known as Pangolin [139] implemented a *"polyhedral path abstraction"* approach, retained a state of exploration in the CE step and enabled more powerful constraint solutions and mutations. Considering the fuzzing stage, the unexplored blocks are first detected and sent to the CE to continue the testing process. Contrary to the CE of a traditional hybrid fuzzer that invokes an SMT solver to achieve an efficient solution directly, Pangolin produces a description of these unexplored basic blocks that denote the abstraction of the polyhedral path.

Saad and Stajano [113] present a novel hybrid fuzzer that employs static and taint analyses alongside the fuzzing tool. Considering the initial step, it analyzes a program utilizing the static analysis component that helps to rank attack points according to their seriousness and sends them to the taint analysis. Thereafter, it requires the user to supply the taint analyzer with a test-input file which makes bytes usable at each attack point. Subsequently, the taint analyzer generates a set of attack points utilized at the particular test-input files and a set of tainted bytes utilized at the control flows. The first list represents the fuzzer's test-input, and a constraint solver will utilize a tainted file with the subsequent list. Gerasimov *et al.* [116] incorporate static analysis, Anxiety [158], as a CE engine, and an AFL fuzzer to detect software vulnerabilities. During the fuzzing process, indirect function calls are involved and passed to the static analyzer. Considering a target program's CFG, this strategy contributes to the improvement of static path identification. The CE engine uses discovered paths to build test-inputs which aids in covering unexplored paths in the execution process. The fuzzer leverages these test-inputs to achieve high code coverage. This method is introduced because it is related to classic hybrid fuzzing that aims to expand code coverage.

A depth-wise coverage pattern of fuzzing and SE was introduced by Fangquan *et al.* [125]. Considering the proposed hybrid fuzzer, the SE engine starts to explore the PUT when the fuzzer fails to expand the code coverage. The first fuzzer runs the PUT with mutated test-inputs as the front end, and the coverage analyzer measures the coverage ratio as a link between fuzzing and the SE, utilizing the static analysis. In addition, it stores the explored nodes, using a dynamic instrumentation tool. If the fuzzer is unable to improve the code coverage, the SE engine begins to explore the PUT as a back end. Fangquan *et al.* employed the ''Generation Search'' (GS) [19] algorithm that negated any complicated path constraints.

VUzzer [50] is defined as a program-aware fuzzing approach that does not demand any prior knowledge of the program or test-input format. It utilizes taint analysis to identify the location of ''magic bytes'' in test-inputs and sets these ''magic bytes'' to fixed test-input locations. The utilization of control and data-flow heuristics rely on static analyzing methods such as *''constant string extraction''* and *''node weight measurement''*, enabling the enhancement of code coverage to explore the basic blocks located in the deeper portions of the program. Tinker [132] introduced the ''Growth Rate of Path Coverage'' (GRPC) algorithm to assess the effectiveness of the fuzzer. Tinker is similar to Driller, considering the working process. It first instruments the target program and constructs the CFG. Whereas the fuzzing process is pending, the GRPC algorithm is utilized to compute the fuzzing's current state. When the fuzzing becomes stalled or stops to cover new paths, Tinker runs the SE engine to produce a novel test input that can provide assistance for the fuzzer to explore the novel paths.

C. EXECUTION MONITORING

1) WHITEBOX-BASED HYBRID FUZZERS

Driller [6] is regarded as a vulnerability excavation hybrid system to expose deep bugs in binaries by incorporating the AFL fuzzer with a selective CE known as angr. Statistics of published articles showed that the rapidly increasing development of hybrid fuzzing tools began with the success of the Driller in the DARPA CGC in 2016. Driller first starts to explore a target program with an AFL fuzzer. The AFL examines the PUT with a specific test-input until it arrives at the first check that is complicated to explore. At this stage, the AFL gets ''stuck'' and fails to produce test-inputs that can cover new paths; hence, Driller leverages angr, the selective CE tool, to solve the constraints. To accomplish this, Angr uses the Z3 SMT solver [159], [160] and produces efficient test-inputs that force the fuzzer to dive deeper inside the uncovered branches. Once CE generates the test-inputs, it sends them back to the AFL. Driller proceeds to cycle between both techniques until the test-inputs trigger the bug. Another hybrid fuzzer, DigFuzz [119], proposed a new approach known as ''Monte Carlo based Probabilistic Path Prioritization'' (MCP3). The MCP3 is seen as a smart algorithm that enables the computation of basic block complexities. It measures the complexity of the path using based on a random test-input that can cover its path. The authors leveraged the Monte Carlo approach [161] to estimate this probability. Considering fuzzing as a random selection process, the MCP3 model constructs an execution tree between various basic blocks. To construct the execution tree, DigFuzz utilizes code coverage information from the fuzzer, and regarding the CE engine, it identifies the test-inputs that include specified basic blocks and run the CE engine on those test-inputs.

DeepFuzz [135], and Munch [133] are hybrid fuzzers that are nearly identical to the Driller approach. DeepFuzz allocates probabilities for path execution to overcome the path explosion issue and uses a novel search heuristic that effectively postpones the explosion of the path into the PUT's deep layers. Essentially, it leverages the probabilistic CE engine to allocate the target program's paths with probabilities and applies these probabilities for the guidance of the path exploration in fuzzing. Munch is an adaptive tool that works in two operating modes: *FS* (Fuzzing+SE) and *SF* (SE+Fuzzing). Considering the *FS* mode, the PUT is explored by the AFL for a set period of time. Thereafter, the PUT is symbolically explored with a sonar-search method to target the uncovered branches by the AFL. The *SF* mode is initiated with the KLEE

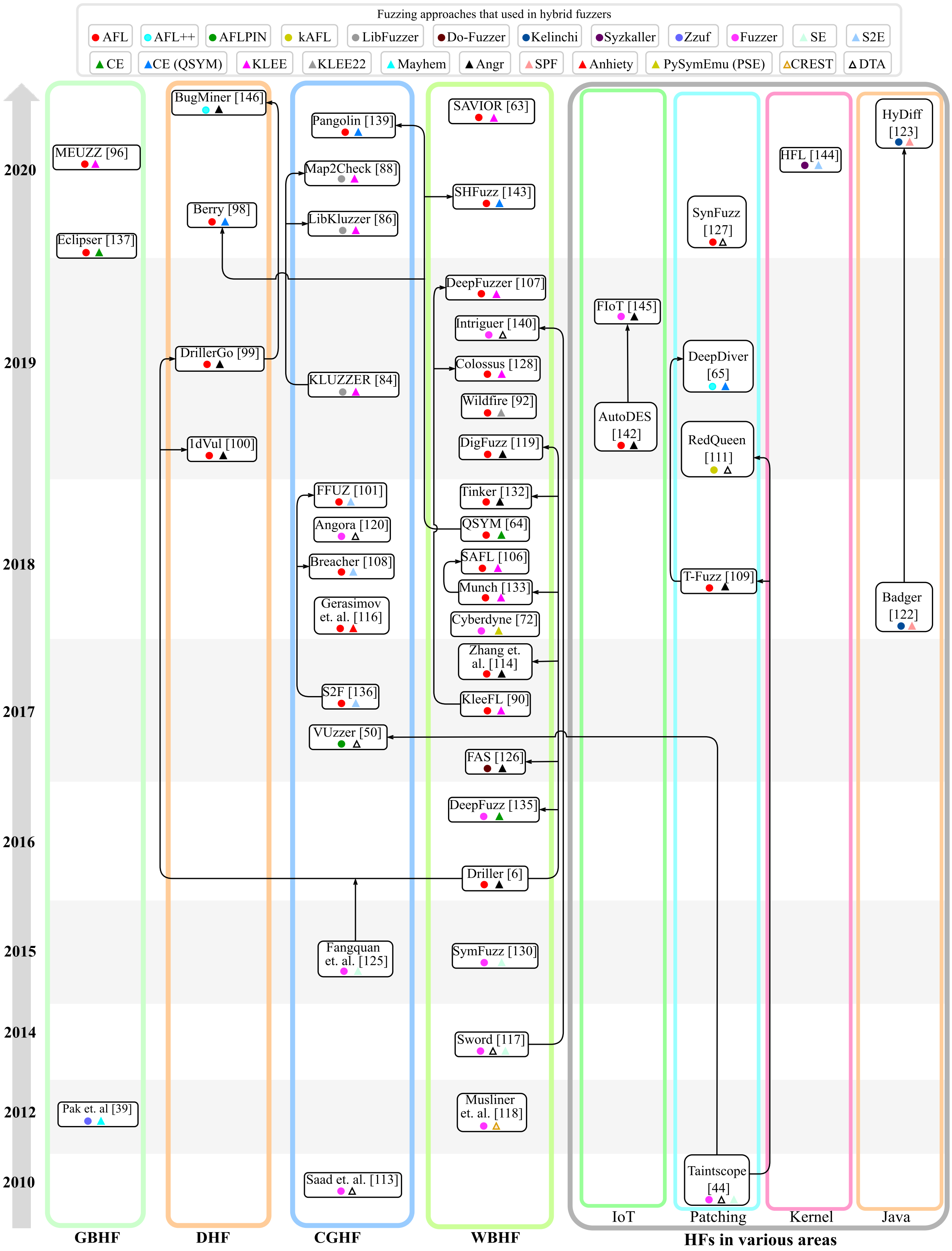


FIGURE 10. Genealogy of hybrid fuzzers. (GBHF: Generation-based hybrid fuzzer; DHF: Directed hybrid fuzzer; CGHF: Coverage-guided hybrid fuzzer; WBHF: Whitebox-based hybrid Fuzzer).

before fuzzing the PUT using the test-inputs generated by the SE.

An approach similar to Munch's latest version has also been implemented by SAFL [106]. The method of SAFL is improved with an efficient coverage-guided mutation strategy and qualified test-input generation. Considering a lightweight method, the SAFL leverages KLEE for an initial test-input generation that can obtain a reasonable path for the AFL fuzzer. It categorizes the test-inputs and mutates them in various forms and weights, depending on the path coverage. The authors emphasized that the algorithm of SAFL can speedily analyze as many deep paths as possible. However, most of the hybrid-fuzzing techniques are highly limited by the slow performance of the SE/CE engine. To tackle this problem, QSYM [64] proposed the fast CE that leveraged ''Dynamic Binary Translation'' (DBT) to integrate the symbolic emulation and the native execution. First, in the QSYM, the PUT is instrumented and executed using the DBT with an initial test-input file obtained from the AFL fuzzer. Considering the native execution, the basic blocks are generated by the DBT and are pruned for the SE engine, enabling a quick swapping between both execution-modes. Thereafter, the system selectively emulates only the instructions that require constraint production. This leads to a reduction in the number of symbolic emulations, attaining a faster execution speed.

Another hybrid fuzzer, SHFuzz [143], presented a novel ''Selective Hybrid Fuzzing'' (SHF) method along with an algorithm for selecting a critical basic block which measures each basic block's critical score and an algorithm for calculating the priority score, providing an effective test-input selection. SHFuzz's employs AFL and QSYM CE, and similar to the QSYM, SHFuzz first explores the PUT with the AFL slave and AFL master, and the QSYM CE obtains candidates from the AFL slave's test-input queue before executing the CE engine. DeepFuzzer [107] integrates KLEE appropriately for the AFL fuzzing. First, DeepFuzzer applies the SE engine to produce initial test-inputs. This indicates that the complicated nested checks can be covered at the beginning of the testing process. Second, it uses a balanced test-input selection strategy to provide sufficient time for each test-input mutation. It prevents over investment of time spent analyzing frequent parts. Third, it leverages a suggested novel technique known as the ''hybrid mutation'' approach which incorporates the ''restricted mutation'' and ''random mutation'' approaches. Consequently, it covers the complicated conditions faster and constantly analysis more deeper paths.

Zhang and Thing [114] begin the PUT testing by executing a resource-bounded SE engine. The SE engine will execute until reaching a predefined threshold. The subsequent process involves the collection of complicated constraints in the analyzed paths and offering solutions for the test-inputs that feed the fuzzing tool. Thereafter, the fuzzer can quickly analyze certain unique basic blocks that are immensely difficult to reach. If the fuzzer cannot explore any novel path, the CE engine starts to trace each analyzed basic block and collects the complicated constraints to solve them. Subsequently, the newly generated test-inputs are sent to the fuzzer queue. The hybrid bug-driven fuzzing approach SAVIOR [115] comprises a compiling tool-chain, KLEE, AFL, and a coordinator that links the fuzzer and CE. The SAVIOR's completion tool-chain is intended to identify bug labels, analyze control flow reachability, and form various target components. SAVIOR proposed the use of UBSan sanitizer [147] to improve the chance of calculating the number of bug labels. The function of CE is to replay the coordinator's scheduled test-inputs and select to resolve constraints according to code-coverage information. Furthermore, SAVIOR provides a bug-guided validation module to validate all detected bug destinations in the execution paths.

Musliner *et al.* [118] utilize an open-source CE technique known as CREST [162] for C language-based programs. It is employed for acquiring complicated constraints. The CREST applies ''C Intermediate Language'' (CIL) [163] that instruments a PUT to run concrete and symbolic execution concurrently. Considering the subsequent stage, vulnerabilities are explored utilizing an *off-the-shelf fuzzer*. Sword [117] hybrid fuzzer incorporates fuzzing, SE, and taint analysis. The SE engine discovers paths and valid variable assignments in the PUT. Taint analysis is performed on these paths to identify taint information. Moreover, the fuzzing tool generates test-inputs based on the path and path-related taint information. Despite these apparent advantages, it possesses some potential problems. For example, the implementation of the test-input generator that maintains these path constraints can be a possible complication.

Similar to Driller and DigFuzz, FAS [126] uses angr [150] as the CE and DO-Fuzzer [164]. Moreover, FAS authors implemented a ''Test-Input-Control System'' that provides the possibility to prioritize SE and the fuzzer based on the novel methodologies of FAS such as the ''Large Distance-First'' and ''Deep-Oriented'' strategies [165]. The goal of these strategies is to choose test-inputs that are located at a long distance from each other such that the directions of their neighbor are less random to avoid excessive testing processes. Consequently, gradual code-coverage can be achieved with maximum ease using FAS. The code-coverage results are improved by the next hybrid fuzzer, Cyberdyne [72]]. It includes four key components such as the GRR [166] which is a high-throughput fuzzer, PySymEmu [167] custom CE tool. Cyberdyne's fuzzer consists of GRR and a scheduler. The scheduler manages the GRR and specifies the way of fuzzing. The GRR provides and mutates test-inputs to the PUT and determines when a test-input triggers a bug by instrumenting the PUT. PySymEmu employs the GRR program snapshots to perform CE for generating test-inputs. Another WBHF SymFuzz [131] proposes a creative approach for maximizing the test-inputs in the fuzzer by tainting the input vectors, and these test-input vectors correspond to program branch circumstances. Using PUT information enhances the test-input mutation technique. SymFuzz includes two major stages to produce test-inputs, and each stage employs various fuzzing techniques such as black-box fuzzer and SE engine to summarize an effective mutation ratio for each test-input.

To overcome the path divergence and irreproducibility in the SE engine, Colossus [128] introduced the ‘‘Deferred Concretization’’ algorithm that utilizes *symcrete* values. To consider complicated constraints, a fuzz-based solver for complicated constraints was also proposed by the authors to leverage the *off-the-shelf fuzzer*. A *predictor* first scans a KLEE query, and if the query is satisfied, the query is sent to the constraints compiler which converts the query into the C language program to check the reachability performance. Thereafter, the AFL explores the produced program to detect the vulnerabilities. To enhance hybrid fuzzing efficiency based on deep research, Intriguer [140] proposes a set of innovative strategies such as ‘‘field-level constraints solving’’ and ‘‘trace reduction’’. The ‘‘field-level constraint solving’’ is the core concept of the Intriguer, which enhances the SE engine performance. The Intriguer receives a PUT and an initial test-input, and seeks to identify interesting values and offsets to produce a novel test-input that analyzes uncovered paths. It conducts taint tracking with a test-input generated by the fuzzing tool on the PUT execution and saves these trace executions. Subsequently, the Intriguer decreases the trace executions for the tainted instruction, which allows it to access various input bytes. These improvements allow the Intriguer to enhance its performance on the SE for relevant instructions and to solve nested constraints.

2) TEST-INPUT SCHEDULING

Scheduling refers to the selection of a fuzzing setup for further iteration. The key idea in scheduling is to evaluate fuzz configuration details and to select one that can achieve the best results. For example, detecting a number of unknown software vulnerabilities or increasing the code-coverage achieved by the produced test-inputs is a way of achieving the desired results. SHFuzz [143] researched the hybrid fuzzer’s basic block scheduling issue using the synchronizing strategy. There were three potential reasons for this issue, and SHFuzz overcame these issues by suggesting a new approach. SHFuzz scheduling contains three modules: code-coverage measurement, basic block, and test-input selection. First, the code coverage is calculated using the instrumentation method. Thereafter, the basic block selection identifies the set of uncovered basic blocks through the use of code coverage details. Finally, the test-input selection module chooses the generated test-input files based on the selected basic blocks and sends them to the CE tool. These three crucial modules ensure that each basic block can be beneficial to maximize code-coverage in lieu of repeatedly re-analyzing paths that have already been explored.

VI. TOOLS IN DIFFERENT APPLICATION AREAS

There are numerous hybrid fuzzers actively employed in various computing systems. By analyzing some of the most common and robust hybrid fuzzers that are categorized according to their application areas, we provide additional answers to the Q3 in this section.

A. PATCHING-BASED HYBRID FUZZERS

The program structure is complicated, particularly if it comprises several nested checksums and magic bytes. This kind of checksum proves difficult for the vulnerability hunting tool. Checksums [168], [169] are standards for evaluating the probity of data commonly utilized in real-world programs, file formats, and different networks. Considering bug hunting tools, generating test-inputs that pass the complicated magic bytes and checksums is too complex and there is a huge possibility that the produced test-inputs will be denied in the initial phases of PUT execution. To tackle this problem, several software security studies have promoted hybrid fuzzing approaches based on patching.

1) TaintScope [44] introduced the first checksum-aware fuzzing approach, which helped detect nested roadblock checks using taint analysis. Specifically, when it runs the program, TaintScope employs taint propagation information to identify nested checksums and generate test-inputs that trigger vulnerabilities, and patch the target program to pass validation of the checksum.

2) T-Fuzz [109] also promoted this perspective and expanded this idea through AFL fuzzing to cover all the complex roadblock checks reliably. Initially, without changing the logic of a program, it constructs ‘‘Non-Critical Checks (NCC)’’ sets which are transformed basic blocks. When the number of new paths coverage does not increase by the AFL, it selects the NCC to patch and resumes fuzzing on the patched program. Finally, if a bug is detected in a patched binary, T-Fuzz attempts to rebuild it on the initial binary through the SE engine. Nevertheless, the T-Fuzz crash analyzer does not function well on all programs. It suffers from path explosion issues if the actual vulnerability is deeply located in the PUT [110].

3) DeepDiver [65] is also intended to patch nested checksums in the PUT and to helps the fuzzer analyze uncovered paths. Using the DeepDiver as a fuzzing tool, AFL++ [170], [171] was utilized because it was combined with a large number of perfect bug hunting tool’s features [172]–[177]. The main goal of the DeepDiver is to dive deep into the target program to excavate software vulnerabilities as well as tackle the shortcomings of T-Fuzz [109] by leveraging the CE. First, it identifies the ‘‘Roadblock Checksums’’ and negates them to explore the uncovered paths; thereafter, the hybrid fuzzing process, AFL++, and the CE engine, analyzes these new directions. Considering the initial step of the CE, it takes a test-input from the AFL++ and transformed program as inputs. Thereafter, it begins to produce novel, efficient test-inputs that can explore the abysmal depth of the PUT. Lastly, it confirms the crash test-inputs as true or false through the bug validation module.

4) RedQueen [111] proposed an efficient alternative taint tracking method and SE to enhance software testing performance. RedQueen relies on lightweight branch-tracing, and it

is capable of solving "magic bytes" and complicated checksums. First, it detects magic bytes or nested complex checksums in the PUT. Thereafter, it patches the operation and later tries to fix the test-inputs. Comparatively, T-Fuzz and DeepDiver do not eliminate false positives in the fuzzing process. Therefore, several fuzzing instances operating at dead endpoints can rise to be practically boundless. RedQueen consistently maintains a queue of valid test-inputs to prevent these scalability problems. Hence, RedQueen neither generates false positives nor does it expend effort on them.

5) SynFuzz [127] introduced an innovative approach to execute formula-solving-based test-input generation through dynamic taint analysis, program synthesis, and a branch flipper. Dynamic taint analysis performs a lightweight data-flow analysis. Program synthesis employs the taint analysis result to synthesize the symbolic basic block and utilizes the synthesis outcomes to generate a new concrete test-input. The SynFuzz obtains two binaries after instrumenting the PUT using "Data Flow Sanitizer" (DFSAN) [178]. The first binary is for regular testing execution that comprises the fuzzing process and gathering of input-output pairs, whereas the second is for the taint analyzing process. SynFuzz automatically obtains new test-inputs applied to the AFL fuzzer, and attempts to synthesize branches to generate new test-inputs. Then, SynFuzz tests whether the test-inputs cover new paths. If the test-inputs can explore uncovered paths, SynFuzz stores them in its own queue which is then synced with AFL's queue.

B. HYBRID FUZZERS FOR OS KERNELS

It is difficult to fuzz Kernel components in OS because feedback systems cannot be easily employed. Specifically, there are some challenges [179], [180] such as *interruptions* and *kernel threads*. Whenever a system explores its own kernel, a crash happens in the kernel, which impacts the fuzzer's efficiency owing to the OS reboot.

The introduction of the first kernel hybrid fuzzer by HFL [144] proved a significant contribution to kernel bug detection. It was constructed on top of the existing fuzzing tool known as named Syzkaller [181] and the SE engine, S2E. Additionally, it solves kernel-specific fuzzer issues through a set of different heuristics: 1) The HFL changes indirect control flows into direct ones by interpreting the original kernel at the compilation time.; 2) It rebuilds system states by inferencing the right calling sequence. More precisely, to decrease the scope of the symbolic variables, HFL executes static points to evaluate in advance making so that it can selectively symbolize data variables within the system states. 3) The HFL extracts nested syscall arguments at runtime by using the domain information on how the kernel handles arguments.

C. HYBRID FUZZERS FOR IOT DEVICE FIRMWARE

The increasing use of IoT devices is encouraging the development of systems that assess their firmware efficiency [182]–[184].

TABLE 4. LAVA-M dataset statistics.

Program	Edges	BB	Ins..	T/Bugs	Options
base64	1308	822	-	44	-d @@
uniq	1407	890	5285	28	@@
md5sum	1560	1013	7397	57	-c @@
who	3332	1831	84648	2136	@@

1) AutoDES [142] is a hybrid fuzzer that seeks to enhance the performance of bug detection and exploitation in IoT applications. It contains three key steps: *AutoD*-automatic bug discovery, *AutoE*-automatic bug exploitation, and *AutoS*-efficient scheduling strategy. Considering the AutoD stage, the authors introduced an innovative approach known as Anti-Driller. Compared to the Driller, it first leverages a CE to identify a particular path and produces a certain test-input. Thereafter, it utilizes a fuzzing tool to determine the IoT application defects during the avoidance of disabled mutations. Three attack approaches were introduced in the AutoE module, including "AutoJS," "AutoROP," and "IPOV fuzzers," which can generate a shell based on the discovered bugs. Regarding the AutoS stage, the authors introduced a "Genetic Algorithm" (GA) that generated a scheduling solution by improving a particular fitness feature.

2) FIoT [145] introduced a new hybrid fuzzer to detect memory corruption bugs in lightweight firmware images of IoT devices. The rationale behind the concept is to explore the PUT code snippets dynamically via the SE engine and a fuzzer. Mainly, it traverses the CFG in a backward manner to produce code snippets. Considering a better performance, the authors enhanced the CFG recovery and backward slice methods. Furthermore, to mitigate the binary firmware impacts, FIoT utilizes the "Loading Address Determination" and "Library Function Identification" analyses.

D. HYBRID FUZZERS FOR JAVA PROGRAMS

1) Badger [122] is a hybrid fuzzer that explores Java language-based programs. It utilizes Kelinci [185], [186], an interface for running an AFL fuzzer on Java programs, and "Symbolic PathFinder" [187], [188] which is an SE engine for Java bytecode. An innovation of this method is controlling user-dependent expenses, which are converted to symbolic expenses on the SE and are managed by the execution of a symbolic maximization process to produce the productive test-inputs. The fuzzer produces and transfers test-inputs that are observed as improving either code-coverage or expenses, and the SE engine imports these test-inputs. The SE updates these test-inputs until it achieves a new code-coverage or discovers a path at a reasonable analytical cost, considering computational resources. Thereafter, test-inputs, which contribute to interesting novel behavior, are transferred to the AFL. Despite this impressive function, Badger lacks the ability to analyze extensive Java-based web programs, and it is

TABLE 5. Number of found bugs in testing on LAVA-M dataset.

Hybrid Fuzzer	T	Fuzzing results			
		base64	uniq	who	md5sum
T-Fuzz	5h	43 97.7%	26 92.7%	63 2.94%	49 85.9%
DeepFuzz	5h	29 64.9 %	16 57.1 %	21 0.98%	30 52.6%
DeepDiver	5h	44 100%	28 100%	101 4.72%	57 100%
QSYM	5h	44 100%	28 100%	1238 57.9%	57 100%
Breacher	5h	37 84.1%	28 100%	203 9.51%	29 50.8%
RedQueen	5h	44 100%	28 100%	2134 99.9%	57 100%
Agnora	5h	48 100%	29 100%	1541 72.1%	57 100%
SAVIOR	24h	48 100%	29 100%	2213 100%	59 100%
DigFuzz	5h	48 100%	29 100%	167 100%	59 100%
Driller	5h	48 100%	25 89.2%	142 6.67%	59 100%
SynFuzz	5h	48 100%	29 100%	2218 100%	61 100%
VUzzer	5h	17 38.6%	27 96.4 %	50 2.34%	- 0%
S2F	5h	10 22.7 %	5 17.8%	17 0.79%	3 5.26%
Eclipser	5h	46 100%	29 100%	1135 53.1%	55 96.4%
Pangolin	24h	48 100 %	30 100%	2021 100%	60 100%
Intriguer	5h	44 100 %	28 100%	2136 100%	57 100%
Listed Bugs	-	**44**	**28**	**2136**	**57**

TABLE 6. Number of crashed binaries in testing on CGC dataset.

Hybrid Fuzzer	T	Total number of binaries	Number of selected binaries	Number of crashed binaries
T-Fuzz	24h	296	296	166
RedQuen	6h	131	54	40
VUzzer	6h	131	63	29
DigFuzz	12h	131	64	12
FAS	6h	131	131	9
Driller	24h	131	126	77
Tinker	2h	131	126	116
QSYM	-	131	126	104
DrillerGo	24h	131	126	14
1dVul	8h	131	126	96
Gerasimov	-	131	18	18

TABLE 7. Google FTS dataset statistics.

Programs	Version	Size (kb)	Type
boringssl	2016-02-12	2241	server program
c-ares	2016-5180	14	asynchronous DNS
guetzli	2017-03-30	3505	image processing
lcms	2017-03-21	1235	management systems
libarchive	2017-01-04	1435	compression library
libssh	2017-1272	1128	server library
libxml2	v2.9.2	4771	image processing
pcre2	10.00	808	regular expression
proj4	2017-08-14	3395	cartographic projections
re2	2014-12-19	4551	regular expression

incapable of doing further exploitability tests in complicated web programs.

2) HyDiff [124] is a novel hybrid differential fuzzing method for Java programs that incorporate the AFL and SE engine "Symbolic Path Finder" (SPF) [188]. The differential fuzzing component of HyDiff consists of a "heuristic-driven greybox fuzzer" with a "divergence-based feedback channel". It leverages an "Inter-procedural Control Flow Graph" (ICFG) [189] for calculating distance metrics for the guidance of exploration in the fuzzer. The differential SE component of HyDiff is the same as Badger's component.

VII. EVALUATION OF HYBRID FUZZING TOOLS

The proposed hybrid fuzzing techniques leverage a number of experimental [190]–[193] and real-world programs [194]–[199] to evaluate the performance of their systems. To answer Q4, we provide widely used datasets by productive hybrid fuzzers in this section. In addition, we provide rich evaluation statistics for explaining the comparisons between different hybrid fuzzers, which can help assess the level of bug hunting tool capabilities. Note that provided evaluation results in tables 5, 6, 8 are obtained from the corresponding studies. The experimental setup of many hybrid fuzzers is almost the same. For example, mainly all tools used at least an Intel i7 processor and Ubuntu 16.04 LTS with 8GB-32GB of RAM. Also, there are hybrid tools that utilize more than 256 RAM and Intel Xeon E5/E7 64 cores [64], [115], [120], [127].

A. LAVA-M DATASET

The LAVA [191] dataset includes numerous buggy programs and software vulnerabilities that are manually inserted to evaluate software testing tools. The LAVA-M [196] dataset is a LAVA dataset's component, and it is one of the popular evaluation datasets that are utilized by virtually all hybrid fuzzers. It includes four programs: the *md5sum, uniq, who,* and *base64* buggy programs. Usually, each hybrid fuzzer tests one of the LAVA-M programs for five hours and tests each program five or eight times to summarize the reasonable performance.

Table 4 illustrates the statistics of the LAVA-M datasets comprising the total listed bugs, basic blocks, edges, instructions, and test options [65]. Table 5 shows the evaluation results of hybrid fuzzers against the LAVA-M dataset. The results show that SAVIOR [63] and Pangolin [139] detected

TABLE 8. Number of covered paths, branches, and crashes in testing on Google FTS dataset.

Programs	Hybrid fuzzing techniques											
	SAFL (24h)			DeepFuzzer (24h)			SHFuzz (12h)			QSYM (24h)		
	Paths	Branches	Crashes	Paths	Branches	Crashes	Paths	Branches	Crashes	Paths	Branches	Crashes
boringssl	988	10231	0	766	11320	0	-	1116	0	716	9228	0
c-ares	37	105	4	36	277	7	-	42	7	34	277	3
guetzli	1329	8982	0	1638	32521	2	-	4300	18	663	5155	0
lcms	335	6587	10	296	7720	7	-	-	-	293	6911	29
libarchive	2093	16296	4	2738	2791	31	-	5836	0	1079	10635	0
libssh	28	1592	7	20	1595	0	-	-	-	18	1595	0
libxml2	4276	40933	5	5827	57048	19	-	5833	6	1870	26212	3
pcre2	18883	75800	103	12568	78552	1734	-	25663	570	11479	71208	699
proj4	850	6281	121	245	2574	0	-	1336	0	76	594	0
re2	3169	30593	1	2213	33025	6	-	5456	0	2413	32665	0
Average	3198	19740	25	2634	22742	180	-	4958	60	1864	16448	73

100% of the bugs on the dataset in 24h, whereas DigFuzz [119], SynFuzz [127], Intriguer [140] achieved 100% results in 5h.

B. DARPA CGC DATASET

The DARPA CGC [4], [190] dataset includes numerous manually realized vulnerable binaries for the CGC competition hosted by DARPA. It includes 296 DECREE binary programs with 248 challenges [109]. However, only 131 binary programs have been implemented to evaluate the capabilities of automated software testing tools in the CGC event [64]. Considering Table 6, we illustrate the evaluation results of hybrid fuzzers that test the CGC dataset programs. The results show that Tinker has the highest performance when testing a CGC database compared to other tools.

C. GOOGLE FUZZER TEST SUITE

The Google fuzzer test suite (FTS) dataset [199] contains popular real-world benchmarks that are utilized to assess the capability of software testing approaches. Table 7 illustrates these statistics of the Google FTS dataset [143]. Table 8 shows the number of covered paths, branches, and crashes by DeepFuzzer, QSYM, SAFL, and SHfuzz in testing on the Google FTS dataset. Considering the test results, DeepFuzzer achieved the highest results in covering program branches and triggering crashes (22742 and 180, respectively). The SAFL achieved the highest result in covering program paths (3198 paths).

VIII. FUTURE DIRECTIONS

To answer Q5, we discuss the potential future directions of hybrid fuzzing tools in this section. Although it is difficult to provide accurate further direction of the hybrid fuzzers, we can summarize some trends mentioned in the reviewed studies, which may serve as guidelines for future studies.

1) TEST-INPUT GENERATION AND SELECTION

The result of software testing is associated with the test-input's quality. Thus, the selection of proper test-input files in the testing process is an essential issue. In addition, a more efficient testing approach must be used to create an opportunity to cover more paths and branches to increase the program's reliability.

The intersections can help enhance the constraint-solving approaches, avoiding the explosion problems, and the use of machine learning-based techniques can help improve the generation of efficient test input files. Furthermore, incorporating ML technologies (including feature extraction, training, and prediction) with hybrid fuzzing approaches enhances the software testing performance. For example, MEUZZ [96] examines the previously encountered test-inputs based on the training and determines which test-inputs can successfully explore uncovered parts of the PUT.

Overall, the ML-based test-input generation and selection approaches prevent the necessity to manually design and test the test-input selection rules because this can be overwhelming when analyzing a large amount of data. Implementing effective and accurate test-input generation and algorithms of proper input selection is an open research issue.

2) PATH EXPLOSION AND COMPLEX CONSTRAINT SOLVING

Integrating SE, CE, and DTA with fuzzers can significantly contribute to improving software testing performance. Although these approaches have several advantages, they also raise several issues: imprecise SE in concolic analysis, path explosion, over-tainting in DTA, and under-tainting. The implementation of a more robust hybrid architecture consumes less time on the analysis in the constraint solver. Moreover, although the current hybrid testing methods have attempted to overcome path explosion in SE quite effectively, the complex constraint solver still remains challenging.

The intersections that optimize constraint-solver methods with static analysis, test-input mutation, and branch pruning can tackle this problem. Another technique described in [39] can be applied as SE threshold restrictions. As a result, the SMT solvers are not blocked in several paths. However, studies on hybrid approaches also have a missing link, which may inspire further studies such as SMT query thresholding and security property assertion statements. We believe this issue is worth investigating more thoroughly and needs further research.

3) TEST-INPUTS FOR INDIVIDUAL COMPONENTS

Considering large-scale programs, it is true that more effective hybrid fuzzers identify more vulnerable parts of the software. Considerable contributions are made in specified vulnerability exploration using SE, CE [82], [200] and fuzzing [201]. However, the key obstacle in integrating fuzzing with compositional SE is the complexity in producing test-inputs for individual components. Compositional SE can be beneficial in generating test-inputs for components, whereas directed or compositional fuzzers [49], [201] can provide fast branch coverage within components.

Overall, the system's internal view allows the fuzzing tool to instrument target program's basic blocks. This assists the SE in directing a modified path search algorithm. It is also possible to employ instrumentation and approaches such as targeted-oriented SE [202], [203] to concentrate on unexplored basic blocks. Considering the fuzzer's perspective, SE and CE make test-input generation more effective, increasing the performance of diverse path exploration in the fuzzer.

4) COMPLICATED NESTED ROADBLOCK CHECKS

The program structure is complex; thus, testing heavyweight real-world programs is not straightforward, especially when it contains many nested roadblock checksums and magic bytes. Furthermore, nested roadblock checksums make the vulnerability mining process complex and cause path explosion problems if bugs are located in the depth of the PUT. To tackle this issue, T-Fuzz and DeepDiver presented a patching-based hybrid fuzzer that transforms the conditional jump instruction of roadblock check if the fuzzers get stuck.

The patching complex nested checksum simplifies the testing process, and the software testing engine can easily explore the hard-to-reach bugs. Our research has shown that patching-based hybrid fuzzing has not yet been sufficiently studied, and we consider it is one of the most trending topics.

5) HYBRID FUZZERS FOR IOT DEVICE FIRMWARE

Security risks are increasing in tandem with the growing usage of IoT devices, and remote attackers can easily target Internet of Things (IoT) devices. The automated vulnerability mining process in IoT firmware is more complicated than testing traditional programs. There are several reasons to make complex firmware testing, for example, multiple micro-architectures, non-standard implementations, hardware interactions, no library or OS-level abstractions, and instrumentation of firmware.

Although fuzzers or SE/CE engines are implemented for hunting bugs in IoT firmware, their code coverage performance is low and they do not yield the expected results. Therefore, it is necessary to develop firmware emulation methods and robust hybrid testing tools that detect vulnerabilities in IoT firmware. We believe that the improvement of the existing hybrid approaches can be further studied.

IX. CONCLUSION

This study aims to review modern hybrid fuzzing studies. We conducted a survey on hybrid fuzzers involving 49 studies published between March 2010 and December 2020. The study's findings indicate that hybrid testing techniques have attracted increasing attention from software testers. Existing hybrid fuzzing techniques have been widely used in numerous industrial products such as compilers, kernels, applications, and systems ranging from binaries to source codes and have detected a tremendous number of exploitable software vulnerabilities. We reviewed several potential issues related to hybrid fuzzing. This study inspires further practices and studies that seek to overcome the challenges associated with a wider adaption of hybrid fuzzing in steady software systems' integration. We believe this study effectively provides a clear insight into modern hybrid techniques, and we hope it serves as a motivation for further studies in more efficient software testing approaches.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their insightful comments.

REFERENCES

[1] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990.

[2] A. Technica, "Pwn2own: The perfect antidote to fanboys who say their platform is safe," San Francisco, CA, USA, 2014. [Online]. Available: https://arstechnica.com

[3] W. E. Howden, "Methodology for the generation of program test data," *IEEE Trans. Comput.*, vol. C-24, no. 5, pp. 554–560, May 1975.

[4] DARPA. *Cyber Grand Challenge*. Accessed: Jun. 12, 2020. [Online]. Available: http://archive.darpa.mil/cybergrandchallenge/

[5] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as Markov chain," *IEEE Trans. Softw. Eng.*, vol. 45, no. 5, pp. 489–506, May 2019.

[6] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *Proc. NDSS*, vol. 16, 2016, pp. 1–16.

[7] GrammaTech. *Grammatech Blogs: The Cyber Grand Challenge*. Accessed: Jul. 10, 2020. [Online]. Available: http://blogs.grammatech.com/the-cyber-grand-challenge

[8] C. S. Team. *Clusterfuzz*. Accessed: Jul. 11, 2020. [Online]. Available: https://code.google.com/p/clusterfuzz/

[9] *Google Chromium Security*. Accessed: Jul. 11, 2020. [Online]. Available: https://www.chromium.org/Home/chromium-security/bugs

[10] M. Aizatsky, K. Serebryany, O. Chang, A. Arya, and M. Whittaker. *Announcing OSS-Fuzz: Continuous Fuzzing for Open Source Software*. Accessed: Jul. 7, 2021. [Online]. Available: https://testing.googleblog.com/2016/12/announcingoss-fuzz-continuous-fuzzing.html

[11] *Binspector: Evolving a Security Tool*. Accessed: Jul. 11, 2020. [Online]. Available: https://blogs.adobe.com/security/2015/05/binspector-evolving-a-security%-tool.html

[12] *Microsoft Security Development Lifecycle, Verification Phase*. Accessed: Jul. 11, 2020. [Online]. Available: https://www.microsoft.com/en-us/sdl/process/verification.aspx

[13] E. Bounimova, P. Godefroid, and D. Molnar, "Billions and billions of constraints: Whitebox fuzz testing in production," in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, May 2013, pp. 122–131.

[14] *Cisco Secure Development Lifecycle*. Accessed: Jul. 11, 2020. [Online]. Available: https://www.cisco.com/c/en/us/about/trust-center/technology-built-in-se%curity.html

[15] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[16] P. Oehlert, "Violating assumptions with fuzzing," *IEEE Secur. Privacy*, vol. 3, no. 2, pp. 58–62, Mar./Apr. 2005.

[17] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. London, U.K.: Pearson, 2007.

[18] E. Clarke, D. Kroening, and K. Yorav, "Behavioral consistency of C and verilog programs using bounded model checking," in *Proc. 40th Conf. Design Autom. (DAC)*, Jun. 2003, pp. 368–371.

[19] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Proc. NDSS*, vol. 8, 2008, pp. 151–166.

[20] C. Cadar and D. Engler, "Execution generated test cases: How to make systems code crash itself," in *Proc. Int. SPIN Workshop Model Checking Softw.* Berlin, Germany: Springer, 2005, pp. 2–23.

[21] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for C," *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 263–272, 2005.

[22] J. Edvardsson, "A survey on automatic test data generation," in *Proc. 2nd Conf. Comput. Sci. Eng.*, 1999, pp. 21–28.

[23] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[24] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: Preliminary assessment," in *Proc. 33rd Int. Conf. Softw. Eng.*, May 2011, pp. 1066–1071.

[25] C. S. Păsăreanu and W. Visser, "A survey of new trends in symbolic execution for software testing and analysis," *Int. J. Softw. Tools Technol. Transf.*, vol. 11, no. 4, p. 339, 2009.

[26] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 1–39, 2018.

[27] L. Y. Araki and L. M. Peres, "A systematic review of concolic testing with aplication of test criteria," in *Proc. 20th Int. Conf. Enterprise Inf. Syst.*, 2018, pp. 121–132.

[28] R. McNally, K. Yiu, D. Grove, and D. Gerhardy, "Fuzzing: The state of the art," Defence Sci. Technol. Organisation Edinburgh, Fairbairn, ACT, Australia, Tech. Rep. ADA558209, 2012.

[29] I. Van Sprundel, "Fuzzing: Breaking software in an automated fashion," in *Proc. 22nd Chaos Commun. Congr., Private Invest.* Berlin, Germany: Berliner Congr. Center, 2005.

[30] V. J. M. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Trans. Softw. Eng.*, early access, Oct. 10, 2019, doi: 10.1109/TSE.2019.2946563.

[31] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Trans. Rel.*, vol. 67, no. 3, pp. 1199–1218, Sep. 2018.

[32] Z. Li, J.-X. Zhang, X. Liao, and J. Ma, "Survey of software vulnerability detection techniques," *Chinses J. Comput.*, vol. 38, no. 4, pp. 717–732, 2015.

[33] Y. Zhang, F. Zhejun, W. Kai, W. Zhiqiang, Y. Hongzhou, L. Qixu, H. Yuan, L. Xiaoqi, and Y. Gang, "Survey of Android vulnerability detection," *J. Comput. Res. Develop.*, vol. 52, no. 10, pp. 2167–2177, 2015.

[34] S. Ognawala, A. Petrovska, and K. Beckers, "An exploratory survey of hybrid testing techniques involving symbolic execution and fuzzing," 2017, *arXiv:1712.06843*. [Online]. Available: http://arxiv.org/abs/1712.06843

[35] Y. Cao, Y. Jiang, C. Xu, J. Ma, and X. Ma, "Perspectives on search strategies in automated test input generation," *Frontiers Comput. Sci.*, vol. 14, no. 3, pp. 1–4, Jun. 2020.

[36] T. Zhang, Y. Jiang, R. Guo, X. Zheng, and H. Lu, "A survey of hybrid fuzzing based on symbolic execution," in *Proc. Int. Conf. Cyberspace Innov. Adv. Technol.*, Dec. 2020, pp. 192–196.

[37] S. Keele, "Guidelines for performing systematic literature reviews in software engineering," EBSE, Goyang-si, South Korea, Tech. Rep. Ver. 2.3, 2007.

[38] B. Kitchenham, *Procedures for Performing Systematic Reviews*, vol. 33. Keele, U.K.: Keele Univ., 2004, pp. 1–26.

[39] B. S. Pak, "Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution," School Comput. Sci. Carnegie Mellon Univ., Pittsburgh, PA, USA, 2012.

[40] *Dynamic Program Analysis*. Accessed: Aug. 11, 2020. [Online]. Available: https://en.wikipedia.org/wiki/Dynamic_program_analysis

[41] J. Fell, "A review of fuzzing tools and methods," PenTest Mag., Tech. Rep., Mar. 2017. [Online]. Available: https://dl.packetstormsecurity.net/papers/general/a~

[42] A. Takanen, J. D. Demott, C. Miller, and A. Kettunen, *Fuzzing for Software Security Testing and Quality Assurance*. Norwood, MA, USA: Artech House, 2018.

[43] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: Whitebox fuzzing for security testing," *Queue*, vol. 10, no. 1, pp. 20–27, Jan. 2012.

[44] T. Wang, T. Wei, G. Gu, and W. Zou, "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *Proc. IEEE Symp. Secur. Privacy*, May 2010, pp. 497–512.

[45] C. Cadar, D. Dunbar, D. R. Engler, and others, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. 8th USENIX Symp. Oper. Syst. Design Implement.*, vol. 8, Dec. 2008, pp. 209–224.

[46] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for *in-vivo* multi-path analysis of software systems," *ACM Sigplan Notices*, vol. 46, no. 3, pp. 265–278, 2011.

[47] M. Zalewski. *American Fuzzy Lop*. Accessed: Jun. 28, 2020. [Online]. Available: https://lcamtuf.coredump.cx/afl/

[48] R. Swiecki. *Honggfuzz*. Accessed: Jun. 28, 2020. [Online]. Available: https://honggfuzz.dev/

[49] K. Serebryany, "Continuous fuzzing with libFuzzer and AddressSanitizer," in *Proc. IEEE Cybersecur. Develop. (SecDev)*, Nov. 2016, p. 157.

[50] S. Rawat, V. Jain, A. Kumar, L. Cojocar, and C. Giuffrida, "VUzzer: Application-aware evolutionary fuzzing," in *Proc. NDSS*, vol. 17, Feb./Mar. 2017, pp. 1–14.

[51] M. Security. *Dharma: A Generation-Based, Context-Free Grammar Fuzzer*. Accessed: Jun. 29, 2020. [Online]. Available: https://github.com/MozillaSecurity/dharma

[52] M. Eddington. *Peach Fuzzing Platform*. Accessed: Jun. 29, 2020. [Online]. Available: https://www.peach.tech/products/peach-fuzzer/

[53] M. Vuagnoux. *Autodafe, an Act of Software Torture*. Accessed: Jun. 29, 2020. [Online]. Available: http://autodafe.sourceforge.net/

[54] W. E. Howden, "Symbolic testing and the DISSECT symbolic evaluation system," *IEEE Trans. Softw. Eng.*, vol. SE-3, no. 4, pp. 266–278, Jul. 1977.

[55] J. C. King, "A new approach to program testing," *ACM SIGPLAN Notices, Int. Conf. Reliable Softw.*, vol. 10, no. 6, pp. 228–233, 1975.

[56] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SOK: (State of) the art of war: Offensive techniques in binary analysis," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2016, pp. 138–157.

[57] R. Majumdar and K. Sen, "Hybrid concolic testing," in *Proc. 29th Int. Conf. Softw. Eng. (ICSE)*, May 2007, pp. 416–426.

[58] K. Sen and G. Agha, "Cute and jCUTE: Concolic unit testing and explicit path model-checking tools," in *Proc. Int. Conf. Comput. Aided Verification*. Berlin, Germany: Springer, 2006, pp. 419–423.

[59] R. Fayozbek, M. Choi, and J. Yun, "Search-based concolic execution for SW vulnerability discovery," *IEICE Trans. Inf. Syst.*, vol. E101.D, no. 10, pp. 2526–2529, 2018.

[60] L. De Moura and N. Bjørner, "Satisfiability modulo theories: Introduction and applications," *Commun. ACM*, vol. 54, no. 9, pp. 69–77, Sep. 2011.

[61] A. Saswat, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, A. Bertolino, and J. J. Li, "An orchestrated survey of methodologies for automated software test case generation," *J. Syst. Softw.*, vol. 86, no. 8, pp. 1978–2001, 2013.

[62] G. Vigna. *Shellphish Team*. Accessed: Jun. 12, 2020. [Online]. Available: https://shellphish.net/

[63] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu, "SAVIOR: Towards bug-driven hybrid testing," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2020, pp. 1580–1596.

[64] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A practical concolic execution engine tailored for hybrid fuzzing," in *Proc. 27th USENIX Secur. Symp. (USENIX Secur.)*, 2018, pp. 745–761.

[65] F. Rustamov, J. Kim, and J. Yun, "DeepDiver: Diving into abysmal depth of the binary for hunting deeply hidden software vulnerabilities," *Future Internet*, vol. 12, no. 4, p. 74, Apr. 2020.

[66] H. Seo and S. Kim, "How we get there: A context-guided search strategy in concolic testing," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Nov. 2014, pp. 413–424.

[67] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," in *Proc. 32nd IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Oct. 2017, pp. 50–59.

[68] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier, "Toward large-scale vulnerability discovery using machine learning," in *Proc. 6th ACM Conf. Data Appl. Secur. Privacy*, Mar. 2016, pp. 85–96.

[69] F. Wu, J. Wang, J. Liu, and W. Wang, "Vulnerability detection with deep learning," in *Proc. 3rd IEEE Int. Conf. Comput. Commun. (ICCC)*, Dec. 2017, pp. 1298–1302.

[70] B. Chernis and R. Verma, "Machine learning methods for software vulnerability detection," in *Proc. 4th ACM Int. Workshop Secur. Privacy Anal.*, Mar. 2018, pp. 31–39.

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

[72] P. Goodman and A. Dinaburg, "The past, present, and future of cyberdyne," *IEEE Secur. Privacy*, vol. 16, no. 2, pp. 61–69, Mar. 2018.

[73] F. G. R. Swiecki. *Honggfuzz*. Accessed: Oct. 11, 2020. [Online]. Available: https://github.com/google/honggfuzz

[74] W. Xu, S. Kashyap, C. Min, and T. Kim, "Designing new operating primitives to improve fuzzing performance," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 2313–2328.

[75] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, and T. Marz, "Comparing operating systems using robustness benchmarks," in *Proc. SRDS 16th IEEE Symp. Reliable Distrib. Syst.*, Oct. 1997, pp. 72–79.

[76] R. S. Boyer, B. Elspas, and K. N. Levitt, "Select—A formal system for testing and debugging programs by symbolic execution," *ACM SigPlan Notices*, vol. 10, no. 6, pp. 234–245, 1975.

[77] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 89–100, 2007.

[78] O. Bastani, R. Sharma, A. Aiken, and P. Liang, "Synthesizing program input grammars," *ACM SIGPLAN Notices*, vol. 52, no. 6, pp. 95–110, 2017.

[79] S. Pailoor, A. Aday, and S. Jana, "Moonshine: Optimizing OS fuzzer seed selection with trace distillation," in *Proc. 27th USENIX Secur. Symp. (USENIX Secur.)*, 2018, pp. 729–743.

[80] A. Arcuri, M. Z. Iqbal, and L. Briand, "Random testing: Theoretical results and practical implications," *IEEE Trans. Softw. Eng.*, vol. 38, no. 2, pp. 258–277, Mar. 2012.

[81] S. Ognavala. *Wildfire*. Accessed: Jun. 10, 2020. [Online]. Available: https://github.com/tum-i22/macke

[82] S. Ognawala, M. Ochoa, A. Pretschner, and T. Limmer, "MACKE: Compositional analysis of low-level vulnerabilities with symbolic execution," in *Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng.*, Aug. 2016, pp. 780–785.

[83] S. Ognawala, A. Pretschner, T. Hutzelmann, E. Psallida, and R. N. Amato, "Reviewing KLEE's sonar-search strategy in context of greybox fuzzing," 2018, *arXiv:1803.04881*. [Online]. Available: http://arxiv.org/abs/1803.04881

[84] H. M. Le, "KLUZZER: Whitebox fuzzing on top of LLVM," in *Proc. Int. Symp. Automated Technol. Verification Anal.* Taipei, Taiwan: Springer, 2019, pp. 246–252.

[85] I. A. Mason. *Whole Program LLVM*. Accessed: Oct. 10, 2020. [Online]. Available: https://github.com/travitch/whole-program-llvm

[86] H. M. Le, "LLVM-based hybrid fuzzing with LibKluzzer (competition contribution)," in *Proc. FASE*, 2020, pp. 535–539.

[87] H. M. Le. *LibKluzzer*. Accessed: Jun. 17, 2020. [Online]. Available: https://gitlab.com/sosy-lab/test-comp/archives-2020/blob/testcom%p20/2020/libkluzzer.zip

[88] H. Rocha, R. Menezes, L. C. Cordeiro, and R. Barreto, "Map2check: Using symbolic execution and fuzzing," in *Proc. Int. Conf. Tools Algorithms Construct. Anal. Syst.* Dublin, Ireland: Springer, 2020, pp. 403–407.

[89] R. Menezes, H. Rocha, L. Cordeiro, and R. Barreto, "Map2check using LLVM and Klee," in *Proc. Int. Conf. Tools Algorithms Construct. Anal. Syst.* Dublin, Ireland: Springer, 2018, pp. 437–441.

[90] J. Fietkau, B. Shastry, and J. Seifert. *Kleefl-Seeding Fuzzers With Symbolic Execution*. Accessed: May 5, 2021. [Online]. Available: https://github.com/julieeen/kleefl/blob/master/USENIX2017poster.pdf

[91] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *Proc. 23rd USENIX Security Symp. (USENIX Secur.)*, 2014, pp. 861–875.

[92] S. Ognawala, F. Kilger, and A. Pretschner, "Compositional fuzzing aided by targeted symbolic execution," 2019, *arXiv:1903.02981*. [Online]. Available: http://arxiv.org/abs/1903.02981

[93] H. M. Le. *Kluzzer*. Accessed: Jun. 15, 2020. [Online]. Available: http://unihb.eu/kluzzer

[94] H. Rocha. *Map2check*. Accessed: Jun. 15, 2020. [Online]. Available: https://github.com/hbgit/Map2Check

[95] J. Fietkau. *Kleefl*. Accessed: Jun. 15, 2020. [Online]. Available: https://github.com/julieeen/kleefl

[96] Y. Chen, M. Ahmadi, R. M. Farkhani, B. Wang, and L. Lu, "MEUZZ: Smart seed scheduling for hybrid fuzzing," 2020, *arXiv:2002.08568*. [Online]. Available: http://arxiv.org/abs/2002.08568

[97] RiS3-Lab. *Meuzz: Smart Seed Scheduling for Hybrid Fuzzing*. Accessed: Apr. 5, 2021. [Online]. Available: https://github.com/RiS3-Lab/muse

[98] H. Liang, L. Jiang, L. Ai, and J. Wei, "Sequence directed hybrid fuzzing," in *Proc. IEEE 27th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Feb. 2020, pp. 127–137.

[99] J. Kim and J. Yun, "Poster: Directed hybrid fuzzing on binary code," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2019, pp. 2637–2639.

[100] J. Peng, F. Li, B. Liu, L. Xu, B. Liu, K. Chen, and W. Huo, "1dVul: Discovering 1-Day vulnerabilities through binary patches," in *Proc. 49th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2019, pp. 605–616.

[101] B. Zhang, J. Ye, X. Bi, C. Feng, and C. Tang, "Ffuzz: Towards full system high coverage fuzz testing on binary executables," *PLoS ONE*, vol. 13, no. 5, May 2018, Art. no. e0196733.

[102] B. Zhang. *Ffuzz*. Accessed: Jun. 17, 2020. [Online]. Available: https://github.com/Epeius/FFuzz

[103] B. Zhang, C. Feng, A. Herrera, V. Chipounov, G. Candea, and C. Tang, "Discover deeper bugs with dynamic symbolic execution and coverage-based fuzz testing," *IET Softw.*, vol. 12, no. 6, pp. 507–519, Dec. 2018.

[104] N. Stephens. *Driller*. Accessed: Jun. 20, 2020. [Online]. Available: https://github.com/shellphish/driller

[105] I. Yun. *Qsym*. Accessed: Jun. 17, 2020. [Online]. Available: https://github.com/sslab-gatech/qsym

[106] M. Wang, J. Liang, Y. Chen, Y. Jiang, X. Jiao, H. Liu, X. Zhao, and J. Sun, "SAFL: Increasing and accelerating testing coverage with symbolic execution and guided fuzzing," in *Proc. 40th Int. Conf. Softw. Eng., Companion*, May 2018, pp. 61–64.

[107] J. Liang, Y. Jiang, M. Wang, X. Jiao, Y. Chen, H. Song, and K.-K.-R. Choo, "DeepFuzzer: Accelerated deep greybox fuzzing," *IEEE Trans. Depend. Sec. Comput.*, early access, Dec. 20, 2020, doi: 10.1109/TDSC.2019.2961339.

[108] J. Liang. *Deepfuzzer*. Accessed: Jun. 18, 2020. [Online]. Available: https://github.com/Ljiee/deepfuzz

[109] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: Fuzzing by program transformation," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2018, pp. 697–710.

[110] H. Peng. *T-Fuzz*. Accessed: Jun. 18, 2020. [Online]. Available: https://github.com/HexHive/T-Fuzz

[111] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: Fuzzing with input-to-state correspondence," in *Proc. NDSS*, vol. 19, 2019, pp. 1–15.

[112] C. Aschermann. *Redqueen*. Accessed: Jun. 19, 2020. [Online]. Available: https://github.com/RUB-SysSec/redqueen

[113] S. Aloteibi and F. Stajano, "On the value of hybrid security testing," in *Proc. Cambridge Int. Workshop Secur. Protocols*. Berlin, Germany: Springer, 2010, pp. 207–213.

[114] L. Zhang and V. L. L. Thing, "A hybrid symbolic execution assisted fuzzing method," in *Proc. TENCON IEEE Region Conf.*, Nov. 2017, pp. 822–825.

[115] Y. Chen. *Savior*. Accessed: Jun. 21, 2020. [Online]. Available: https://github.com/evanmak/savior-source

[116] A. Y. Gerasimov, S. S. Sargsyan, S. S. Kurmangaleev, J. A. Hakobyan, S. A. Asryan, and M. K. Ermakov, "Combining dynamic symbolic execution, code static analysis and fuzzing," *Proc. Inst. Syst. Program. RAS*, vol. 30, no. 6, pp. 25–38, 2018.

[117] J. Cai, S. Yang, J. Men, and J. He, "Automatic software vulnerability detection based on guided deep fuzzing," in *Proc. IEEE 5th Int. Conf. Softw. Eng. Service Sci.*, Jun. 2014, pp. 231–234.

[118] D. J. Musliner, J. M. Rye, and T. Marble, "Using concolic testing to refine vulnerability profiles in FUZZBUSTER," in *Proc. IEEE 6th Int. Conf. Self-Adaptive Self-Organizing Syst. Workshops*, Sep. 2012, pp. 9–14.

[119] L. Zhao, Y. Duan, H. Yin, and J. Xuan, "Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing," in *Proc. NDSS*, 2019, pp. 1–15.

[120] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2018, pp. 711–725.

[121] P. Chen. *Angora*. Accessed: Jun. 21, 2020. [Online]. Available: https://github.com/AngoraFuzzer/Angora

[122] Y. Noller, R. Kersten, and C. S. Păsăreanu, "Badger: Complexity analysis with fuzzing and symbolic execution," in *Proc. 27th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Jul. 2018, pp. 322–332.

[123] Y. Noller, C. S. Păsăreanu, M. Böhme, Y. Sun, H. L. Nguyen, and L. Grunske, "HyDiff: Hybrid differential software analysis," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.*, Jun. 2020, pp. 1273–1285.

[124] Y. Noller. *Hydiff*. Accessed: Jun. 25, 2020. [Online]. Available: https://github.com/yannicnoller/hydiff

[125] D. Fangquan, D. Chaoqun, Z. Yao, and L. Teng, "Binary-oriented hybrid fuzz testing," in *Proc. 6th IEEE Int. Conf. Softw. Eng. Service Sci. (ICSESS)*, Sep. 2015, pp. 345–348.

[126] J. Ye, B. Zhang, Z. Ye, C. Feng, and C. Tang, "Improving the cooperation of fuzzing and symbolic execution by test-cases prioritizing," in *Proc. 13th Int. Conf. Comput. Intell. Secur. (CIS)*, Dec. 2017, pp. 543–547.

[127] W. Han, M. Lutfor Rahman, Y. Chen, C. Song, B. Lee, and I. Shin, "SynFuzz: Efficient concolic execution via branch condition synthesis," 2019, *arXiv:1905.09532*. [Online]. Available: http://arxiv.org/abs/1905.09532

[128] A. Pandey, P. R. G. Kotcharlakota, and S. Roy, "Deferred concretization in symbolic execution via fuzzing," in *Proc. 28th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Jul. 2019, pp. 228–238.

[129] S. Rawat. *Vuzzer*. Accessed: Jun. 28, 2020. [Online]. Available: https://github.com/vusec/vuzzer

[130] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 725–741.

[131] S. K. Cha. *Symfuzz*. Accessed: Jun. 30, 2020. [Online]. Available: http://security.ece.cmu.edu/symfuzz/

[132] L. Xu, L. Yin, W. Dong, W. Jia, and Y. Li, "Expediting binary fuzzing with symbolic analysis," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 28, nos. 11–12, pp. 1701–1718, Nov. 2018.

[133] S. Ognawala, T. Hutzelmann, E. Psallida, and A. Pretschner, "Improving function coverage with munch: A hybrid fuzzing and directed symbolic execution approach," in *Proc. 33rd Annu. ACM Symp. Appl. Comput.*, Apr. 2018, pp. 1475–1482.

[134] S. Ognawala. *Munch*. Accessed: Jul. 4, 2020. [Online]. Available: https://github.com/tum-i22/munch

[135] K. Böttinger and C. Eckert, "DeepFuzz: Triggering vulnerabilities deeply hidden in binaries," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*. San Sebastián, Spain: Springer, 2016, pp. 25–34.

[136] B. Zhang, J. Ye, C. Feng, and C. Tang, "S2f: Discover hard-to-reach vulnerabilities by semi-symbolic fuzz testing," in *Proc. 13th Int. Conf. Comput. Intell. Secur. (CIS)*, Dec. 2017, pp. 548–552.

[137] J. Choi, J. Jang, C. Han, and S. K. Cha, "Grey-box concolic testing on binary code," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, May 2019, pp. 736–747.

[138] J. Choi. *Eclipser*. Accessed: Jul. 5, 2020. [Online]. Available: https://github.com/SoftSec-KAIST/Eclipser

[139] H. Huang, P. Yao, R. Wu, Q. Shi, and C. Zhang, "Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2020, pp. 1613–1627.

[140] M. Cho, S. Kim, and T. Kwon, "Intriguer: Field-level constraint solving for hybrid fuzzing," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2019, pp. 515–530.

[141] M. Cho. *Intriguer*. Accessed: Jul. 8, 2020. [Online]. Available: https://github.com/seclab-yonsei/intriguer

[142] Z. Wang, Y. Zhang, Z. Tian, Q. Ruan, T. Liu, H. Wang, Z. Liu, J. Lin, B. Fang, and W. Shi, "Automated vulnerability discovery and exploitation in the Internet of Things," *Sensors*, vol. 19, no. 15, p. 3362, Jul. 2019.

[143] X. Mi, B. Wang, Y. Tang, P. Wang, and B. Yu, "SHFuzz: Selective hybrid fuzzing with branch scheduling based on binary instrumentation," *Appl. Sci.*, vol. 10, no. 16, p. 5449, Aug. 2020.

[144] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "HFL: Hybrid fuzzing on the Linux kernel," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2020, pp. 1–17.

[145] L. Zhu, X. Fu, Y. Yao, Y. Zhang, and H. Wang, "FIoT: Detecting the memory corruption in lightweight IoT device firmware," in *Proc. 18th IEEE Int. Conf. Trust, Secur. Privacy Comput. Commun./13th IEEE Int. Conf. Big Data Sci. Eng. (TrustCom/BigDataSE)*, Aug. 2019, pp. 248–255.

[146] F. Rustamov, J. Kim, J. Yu, H. Kim, and J. Yun, "BugMiner: Mining the hard-to-reach software vulnerabilities through the target-oriented hybrid fuzzer," *Electronics*, vol. 10, no. 1, p. 62, Dec. 2020.

[147] C. Team. *Undefined Behavior Sanitizer—Clang 9 Documentation*. Accessed: Oct. 12, 2020. [Online]. Available: https://clang.llvm.org/docs/

[148] R. Bagnara, M. P. Hill, and E. Zaffanella, "The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems," *Sci. Comput. Program.*, vol. 72, nos. 1–2, pp. 3–21, 2008.

[149] T. Armerding. *What is CVE, its Definition and Purpose*. Accessed: Dec. 12, 2020. [Online]. Available: https://www.csoonline.com/article/3204884/what-is-cve-its-defini%tion-and-purpose.html

[150] Y. Shoshitaishvili. *Angr*. Accessed: Jun. 10, 2020. [Online]. Available: http://fuse.sourceforge.net

[151] Zynamics. *Bindiff—Comparison Tool for Binary Files*. Accessed: Oct. 12, 2020. [Online]. Available: https://www.zynamics.com/bindiff.html

[152] N. Hardeniya, J. Perkins, D. Chopra, N. Joshi, and I. Mathur, *Natural Language Processing: Python and NLTK*. Birmingham, U.K.: Packt Publishing Ltd, 2016.

[153] Test-Comp-2020. *2nd Competition on Software Testing*. Accessed: Oct. 12, 2020. [Online]. Available: https://test-comp.sosy-lab.org/2020/

[154] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *Proc. Int. Conf. Comput. Aided Verification*. Berlin, Germany: Springer, 2007, pp. 519–531.

[155] B. Dutertre, "Yices 2.2," in *Proc. Int. Conf. Comput. Aided Verification*. Vienna, Austria: Springer, 2014, pp. 737–744.

[156] R. Brummayer and A. Biere, "Boolector: An efficient SMT solver for bit-vectors and arrays," in *Proc. Int. Conf. Tools Algorithms Construct. Anal. Syst.* Berlin, Germany: Springer, 2009, pp. 174–177.

[157] SVCOMP20. *9th Competition on Software Verification*. Accessed: Dec. 12, 2020. [Online]. Available: https://sv-comp.sosy-lab.org/2020/results/results-verified/

[158] A. Gerasimov, S. Vartanov, M. Ermakov, L. Kruglov, D. Kutz, A. Novikov, and S. Asryan, "Anxiety: A dynamic symbolic execution framework," in *Proc. Ivannikov ISPRAS Open Conf. (ISPRAS)*, Nov. 2017, pp. 16–21.

[159] L. D. Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.* Berlin, Germany: Springer, 2008, pp. 337–340.

[160] M. Research. *Z3*. Accessed: Dec. 12, 2020. [Online]. Available: https://github.com/Z3Prover/z3

[161] C. P. Robert, V. Elvira, N. Tawn, and C. Wu, "Accelerating MCMC algorithms," *WIREs Comput. Statist.*, vol. 10, no. 5, Sep. 2018, Art. no. e1435.

[162] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *Proc. 23rd IEEE/ACM Int. Conf. Automated Softw. Eng.*, Sep. 2008, pp. 443–446.

[163] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: Intermediate language and tools for analysis and transformation of C programs," in *Proc. Int. Conf. Compiler Construct.* Berlin, Germany: Springer, 2002, pp. 213–228.

[164] J. Ye, C. Feng, and C. Tang, "A fuzzer based on a fine-grained deeper strategy," in *Proc. 4th Int. Conf. Inf. Sci. Control Eng. (ICISCE)*, Jul. 2017, pp. 24–28.

[165] C. Zhang, A. Groce, and M. A. Alipour, "Using test case reduction and prioritization to improve symbolic execution," in *Proc. Int. Symp. Softw. Test. Anal. (ISSTA)*, 2014, pp. 160–170.

[166] P. Goodman. *GRR Binary Translator*. Accessed: Oct. 28, 2020. [Online]. Available: https://github.com/lifting-bits/grr

[167] Goodman. *Pysymemu—Custom Symbolic Execution Engine*. Accessed: Oct. 28, 2020. [Online]. Available: https://github.com/trailofbits/manticore

[168] Wikipedia. *Checksum*. Accessed: Oct. 5, 2020. [Online]. Available: https://en.wikipedia.org/wiki/Checksum

[169] L. Fitzgibbons. *Checksum*. Accessed: Oct. 5, 2020. [Online]. Available: https://searchsecurity.techtarget.com/definition/checksum

[170] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *Proc. 14th USENIX Workshop Offensive Technol. (WOOT)*, 2020, pp. 1–12.

[171] M. Hauster. *American Fuzzy Lop Plus Plus (AFL++)*. Accessed: Dec. 5, 2020. [Online]. Available: https://github.com/AFLplusplus/AFLplusplus

[172] M. Bohme. *Aflfast*. Accessed: Dec. 5, 2020. [Online]. Available: https://github.com/mboehme/aflfast

[173] C. Lyu. *Mopt-AFL*. Accessed: Dec. 5, 2020. [Online]. Available: https://github.com/puppet-meteor/MOpt-AFL

[174] A. Herrera. *AFL N-Gram Branch Coverage*. Accessed: Dec. 5, 2020. [Online]. Available: https://github.com/adrianherrera/afl-ngram-pass

[175] QuarkslaB. *A Dynamic Binary Instrumentation Framework Based on LLVM*. Accessed: Dec. 5, 2020. [Online]. Available: https://github.com/QBDI/QBDI

[176] A. Helin. *Radamsa*. Accessed: Dec. 5, 2020. [Online]. Available: https://gitlab.com/akihe/radamsa

[177] Google. *Hongfuzz*. Accessed: Dec. 5, 2020. [Online]. Available: https://github.com/google/honggfuzz

[178] ClangTeam. *Dataflowsanitizer Design Document*. Accessed: Dec. 5, 2020. [Online]. Available: https://clang.llvm.org/docs/DataFlowSanitizerDesign.html

[179] A. Prakash, E. Venkataramani, H. Yin, and Z. Lin, "Manipulating semantic values in kernel data structures: Attack assessments and implications," in *Proc. 43rd Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2013, pp. 1–12.

[180] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-assisted feedback fuzzing for OS kernels," in *Proc. 26th USENIX Secur. Symp. (USENIX Secur.)*, 2017, pp. 167–182.

[181] D. Vyukov. *Syzkaller—Kernel Fuzzer*. Accessed: Jan. 5, 2021. [Online]. Available: https://github.com/google/syzkaller

[182] J. Kim, J. Yu, H. Kim, F. Rustamov, and J. Yun, "FIRM-COV: high-coverage greybox fuzzing for IoT firmware via optimized process emulation," *IEEE Access*, vol. 9, pp. 101627–101642, 2021.

[183] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "FIRM-AFL: High-throughput greybox fuzzing of IoT firmware via augmented process emulation," in *Proc. 28th USENIX Secur. Symp. (USENIX Secur.)*, 2019, pp. 1099–1114.

[184] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "Avatar: A framework to support dynamic security analysis of embedded Systems' firmwares," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2014, pp. 1–16.

[185] R. Kersten, K. Luckow, and C. S. Păsăreanu, "POSTER: AFL-based fuzzing for Java with Kelinci," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 2511–2513.

[186] ISSTAC. *Afl-Based Fuzzing for Java With Kelinci*. Accessed: Apr. 1, 2021. [Online]. Available: https://github.com/isstac/kelinci

[187] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta, "Symbolic PathFinder: Integrating symbolic execution with model checking for Java bytecode analysis," *Automated Softw. Eng.*, vol. 20, no. 3, pp. 391–425, Sep. 2013.

[188] C. Pasareanu. *Symbolic (Java) Pathfinder*. Accessed: Apr. 1, 2021. [Online]. Available: https://github.com/SymbolicPathFinder/jpf-symbc

[189] F. Nielson and H. R. Nielson, "Interprocedural control flow analysis," in *Proc. Eur. Symp. Program.* Berlin, Germany: Springer, 1999, pp. 20–39.

[190] DARPA. *Cyber Grand Challenge Dataset*. Accessed: Jun. 12, 2020. [Online]. Available: https://github.com/CyberGrandChallenge/samples/tree/master/cqe-challeng%es

[191] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "LAVA: Large-scale automated vulnerability addition," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2016, pp. 110–121.

[192] D. Gavitt. *Lava Dataset Source Code*. Accessed: Apr. 21, 2021. [Online]. Available: https://github.com/panda-re/lava

[193] Dolan. *Lava-M Dataset Source Code*. Accessed: Jun. 12, 2020. [Online]. Available: https://sites.google.com/site/steelix2017/home/lava

[194] GNU. *Binutils Source Code*. Accessed: Apr. 21, 2021. [Online]. Available: https://ftp.gnu.org/gnu/binutils/

[195] G. E. Schalnat. *Libpng—A Library for Processing Png Files*. Accessed: Apr. 21, 2021. [Online]. Available: http://www.libpng.org/pub/png/libpng.html

[196] Google. *Openssl-Cryptography and SSL/TLS Toolkit*. Accessed: Apr. 21, 2021. [Online]. Available: https://ftp.openssl.org/source/old/1.0.1/

[197] M. Matuska. *Libarchive: Multi-Format Archive and Compression Library*. Accessed: Apr. 21, 2021. [Online]. Available: https://libarchive.org/downloads/

[198] Poppler. *PDF Rendering Library*. Accessed: Apr. 21, 2021. [Online]. Available: https://poppler.freedesktop.org/releases.html

[199] Google. *Google Fuzzer-Test-Suite*. Accessed: Oct. 12, 2020. [Online]. Available: https://github.com/google/fuzzer-test-suite

[200] M. Christakis and P. Godefroid, "Proving memory safety of the ani windows image parser using compositional exhaustive testing," in *Proc. Int. Workshop Verification, Model Checking, Abstract Interpretation*. Naples, Italy: Springer, 2015, pp. 373–392.

[201] C. Shortt and J. Weber, "Hermes: A targeted fuzz testing framework," in *Proc. Int. Conf. Intell. Softw. Methodolog., Tools, Techn.* Berlin, Germany: Springer, 2015, pp. 453–468.

[202] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks, "Directed symbolic execution," in *Proc. Int. Static Anal. Symp.* Venice, Italy: Springer, 2011, pp. 95–111.

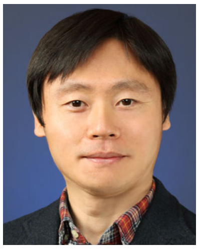
[203] M. Christakis and P. Godefroid, "IC-Cut: A compositional search strategy for dynamic test generation," in *Proc. Int. SPIN Workshop Model Checking Softw.* Stellenbosch, South Africa: Springer, 2015, pp. 300–318.

FAYOZBEK RUSTAMOV was born in Andijan region, Uzbekistan, in 1990. He received the B.S. degree in economics and management in the sphere of ICT and the M.S. degree in information security, cryptography, and cryptanalysis from Tashkent University of Information Technologies (TUIT), in 2013 and 2015, respectively, and the Ph.D. degree in computer and information security from Sejong University, Seoul, South Korea, in 2021.

His research interests include software security and vulnerability detection. He was an award recipient named "Beruniy" by the Ministry of Higher Education of the Republic of Uzbekistan, in 2013, and a recipient of "Young Scientist" Award by the Republic of Uzbekistan ICT Ministry, in 2014.

JUHWAN KIM received the M.S. degree in computer and information security from Sejong University, Seoul, South Korea, in 2019, where he is currently pursuing the Ph.D. degree in computer and information security and convergence engineering for intelligent drone. His research interests include fuzzing, vulnerability detection, the IoT security, and software security.

JIHYEON YU is currently pursuing the M.S. degree in computer and information security, and convergence engineering for intelligent drone with Sejong University, Seoul, South Korea. His research interests include fuzzing, vulnerability detection, the IoT security, artificial intelligence (AI) security, and network security.



JOOBEOM YUN received the B.S. degree in computer science and engineering from Korea University, Seoul, South Korea, in 1999, the M.S. degree in computer engineering from Seoul National University, Seoul, in 2001, and the Ph.D. degree in computer science from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, South Korea, in 2012. He is currently an Associate Professor with the Department of Computer and Information Security, Sejong University, Seoul. His research interests include software security, artificial intelligence (AI) security, and network security.