Fuzzing: State of the Art

Hongliang Liang, *Member, IEEE*, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, *Member, IEEE*, and Jian Zhang, *Senior Member, IEEE*

***Abstract*—As one of the most popular software testing techniques, fuzzing can find a variety of weaknesses in a program, such as software bugs and vulnerabilities, by generating numerous test inputs. Due to its effectiveness, fuzzing is regarded as a valuable bug hunting method. In this paper, we present an overview of fuzzing that concentrates on its general process, as well as classifications, followed by detailed discussion of the key obstacles and some state-of-the-art technologies which aim to overcome or mitigate these obstacles. We further investigate and classify several widely used fuzzing tools. Our primary goal is to equip the stakeholder with a better understanding of fuzzing and the potential solutions for improving fuzzing methods in the spectrum of software testing and security. To inspire future research, we also predict some future directions with regard to fuzzing.**

***Index Terms*—Fuzzing, reliability, security, software testing, survey.**

I. Introduction

FUZZING (short for fuzz testing) is "an automatic testing technique that covers numerous boundary cases using invalid data (from files, network protocols, application programming interface (API) calls, and other targets) as application input to better ensure the absence of exploitable vulnerabilities" [1]. Fuzzing was first proposed by Miller *et al*. [2] in 1988, and since then, it has developed into an effective, fast, and practical method to find bugs in software [3]–[5]. The key idea behind fuzzing is to generate and feed the target program with plenty of test cases which are hopeful to trigger software errors. The most creative part of fuzzing is the way to generate suitable test cases, and current popular methods include coverage-guided strategies, genetic algorithm, symbolic execution, taint analysis, etc. Based on these methods, a modern fuzzing technique is very intelligent to reveal hidden bugs. Therefore, as the unique testing method for which test success can be quantified in meaningful software-quality terms, fuzzing has an important theoretical and experimental role. It serves as the standard of comparison by which other methods should be evaluated [6]. Furthermore, fuzzing has gradually evolved into a synthesis technique to synergistically combine static and dynamic information of a target program so that better test cases can be produced and more bugs are detected [7].

Fuzzing simulates attacks by constantly sending malformed or semivalid test cases to the target program. Thanks to these irregular inputs, fuzzers, also named as fuzzing tools, can often find out previously unknown vulnerabilities [8]–[11]. That is one of the key reasons why fuzzing plays an important role in software testing. However, the blindness during a process of test case generation which can lead to low code coverage is the main drawback that fuzzing has been trying to overcome. As mentioned above, several methods have been utilized to mitigate this problem and fuzzing has made an impressive progress. Nowadays, fuzzing has been widely applied to test various software, including compilers, applications, network protocols, kernels, etc., and multiple application areas, such as evaluation of syntax error recovery [12] and fault localization [13].

A. Motivation

There are two reasons that motivate us to write this overview.

1) Fuzzing is getting more and more attention in the area of software security and reliability because of its effective ability to find bugs. Many IT companies such as Google and Microsoft are studying fuzzing techniques and further developing fuzzing tools (e.g., SAGE [14], Syzkaller [15], SunDew [16], etc.) to find bugs in the target program.
2) There is no systematic review of fuzzing in the past few years. Although some papers present overviews of fuzzing, they are usually review of selected articles [1], [17] or surveys on a specific testing topic [18], [19]. Therefore, we think it is necessary to write a comprehensive review to summarize the latest methods and new research results in this area. Thus, with this paper, we hope that not only the beginners can get a general understanding of fuzzing but also the professionals can have a thorough review of fuzzing.

B. Outline

The rest of this paper is organized as follows: Section II presents the review methodology used in our survey as well as a brief summary and analysis of some selected papers. Section III describes the general process of fuzzing. Section IV introduces the classification of fuzzing methods. Section V

Manuscript received November 1, 2017; revised February 21, 2018 and April 25, 2018; accepted May 5, 2018. Date of publication April 6, 2018; date of current version August 30, 2018. This work was supported by the National Natural Science Foundation of China (NSFC) under Grant U1713212 and Grant 91418206 and the Key Research Program of Frontier Sciences, Chinese Academy of Sciences, under Grant QYZDJ-SSW-JSC036. Associate Editor: T. H. Tse. *(Corresponding author: Hongliang Liang.)*

H. Liang, X. Pei, and X. Jia are with the Beijing University of Posts and Telecommunications, Beijing 100876, China (e-mail: hliang@bupt.edu.cn; calvin@bupt.edu.cn; rose11130@bupt.edu.cn).

W. Shen is with Western Michigan University, Kalamazoo, MI 49008 USA (e-mail: wuwei.shen@wmich.edu).

J. Zhang is with the Institute of Software, Chinese Academy of Sciences, Beijing 100190, China (e-mail: zj@ios.ac.cn).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TR.2018.2834476

describes the state of the art in fuzzing. Section VI provides several popular fuzzers classified by their application areas and problem domains. As a result of our survey, a number of research challenges are identified as avenues for future research in Section VII. Finally, Section VIII concludes this paper.

II. Review Method

To conduct a comprehensive survey on fuzzing, we followed a systematic and structured method inspired by the guidelines of Kitchenham [20] and Webster and Watson [21]. In the following sections, we will introduce our research methods, collected data, and analysis in detail.

A. Research Questions

This survey mainly aims to answer the following research questions about fuzzing.

1) RQ1: What are the key problems and the corresponding techniques in fuzzing research?
2) RQ2: What are the usable fuzzers and their known application domains?
3) RQ3: What are the future research opportunities or directions?

RQ1, which is answered in Section V, allows us to explore an in-depth view on fuzzing outlining the state-of-the-art advancement in this area since its original introduction. RQ2, which is discussed in Section VI, is proposed to give an insight into the scope of fuzzing and its applicability to different domains. Finally, based on the answer to the previous questions, we expect to identify unresolved problems and research opportunities in response to RQ3, which is answered in Section VII.

B. Inclusion and Exclusion Criteria

We scrutinized the existing literature in order to find papers related to all aspects of fuzzing, such as methods, tools, applications to specific testing problems, empirical evaluations, and surveys. Articles written by the same authors with similar content were intentionally classified and evaluated as separate contributions for a more rigorous analysis. Then, we grouped these articles with no major differences in the presentation of results. We excluded those papers based on the following criteria:

1) not related to the computer science field;
2) not written in English;
3) not published by a reputed publisher;
4) published by a reputed publisher but with less than six pages;
5) not accessible via the Web.

For instance, using the search interface of Wiley InterScience website with keywords, such as fuzzing/fuzz testing/fuzzer, we had 32 papers, seven of which are only related to computer science field according to their abstract.

C. Source Material and Search Strategy

In order to provide a complete survey covering all the publications relating to fuzzing, we constructed a fuzzing publication repository, which includes more than 350 papers from January 1990 to June 2017, via three steps. First, we searched some main online repositories such as IEEE XPlore, ACM Digital Library, Springer Online Library, Wiley InterScience, USENIX and Elsevier ScienceDirect Online Library, and collected papers with either "fuzz testing," "fuzzing," "fuzzer," "random testing," or "swarm testing" as keywords in their titles, abstracts, or keywords. Second, we used abstracts of the collected papers to exclude some of them based on our selection criteria. We did read through a paper if it cannot be decided by its abstract. This step was carried out by two different authors. The set of candidate papers was reduced to 171 publications within the scope of our survey. These papers are referred to as the primary studies [20]. Table I presents the number of primary studies retrieved from each source.

TABLE I
Publishers and Number of Primary Studies

Publisher	Primary studies
ACM digital library	33
Elsevier ScienceDirect	10
IEEEXplore digital library	87
Springer online library	17
Wiley InterScience	7
USENIX	11
Semantic scholar	6
Total	171

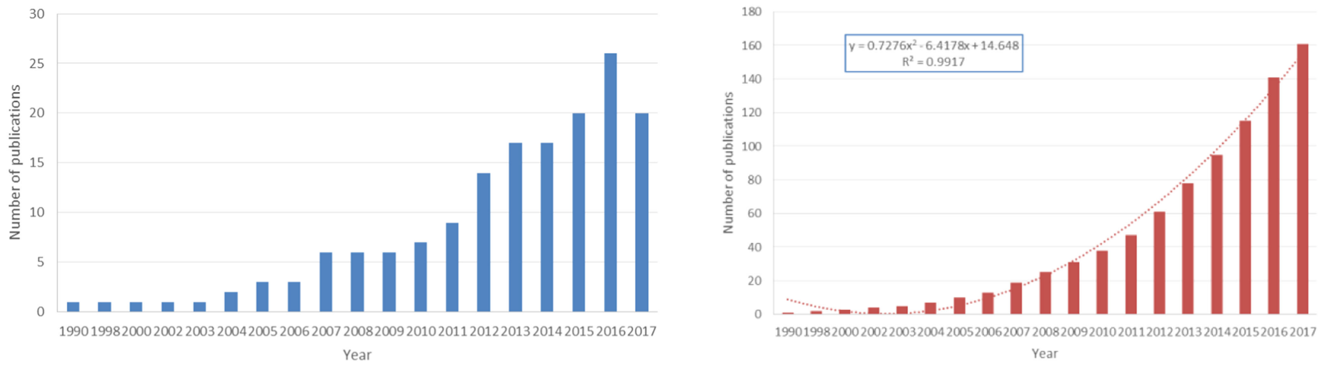
It is still possible for our search to not completely cover all the related papers since we focused on a subset of reputed publishers. However, we are confident that the overall trends in this paper are accurate and provide a fair picture of the state of the art on fuzzing.

D. Summary of Results

The following sections summarize our primary studies in terms of publication trends, venues, authors, and geographical distribution on fuzzing.

1) Publication Trends: Fig. 1(a) illustrates the number of publications about fuzzing between January, 1990 and June 30, 2017. The graph shows that the number of papers on this topic had a constant increase since 2004, especially after 2009. The cumulative number of publications is illustrated in Fig. 1(b). We calculated a close fit to a quadratic function with a high determination coefficient ($R^2 = 0.992$), indicating a strong polynomial growth, a sign of continued health and interest on this subject. If the trend continues, there will be more than 200 fuzzing papers published by reputed publishers by the end of 2018, three decades after this technique was first introduced.

2) Publication Venues: The 171 primary studies were published in 78 distinct venues. It means the fields covered by fuzzing literature are very wide. It is probably because this technique is very practical and has been applied to multiple testing, reliability, and security domains. Regarding the type of venues, most papers were presented at conferences and symposia (73%), followed by journals (15%), workshops (9%), and



(a) (b)

Fig. 1. Fuzzing papers published between January 1, 1990 and June 30, 2017. (a) Number of publications per year. (b) Cumulative number of publications per year.

TABLE II
TOP VENUES ON FUZZING

Venue	Papers
Int. Conf. on Programming Language Design and Implementation (PLDI)	12
Int. Symposium on Software Testing and Analysis (ISSTA)	9
Int. Conf. on Software Engineering (ICSE)	8
Int. Conf. on Automated Software Engineering (ASE)	7
Int. Conf. on Software Testing, Verification & Validation (STVV)	6
USENIX Security Symposium (USENIX SEC.)	5
Workshop on Offensive Technologies (WOOT)	5
IEEE Transaction on Reliability (TR-IEEE)	4
Network and Distributed System Security Symposium (NDSS)	4
ACM Conf. on Computer and Communications Security (CCS)	4
IEEE Symposium on Security and Privacy (S&P)	3
Communication of ACM (CACM)	3
ESEC/ACM Foundations of Software Engineering (FSE)	3
Security and Communication Networks (SCN)	3

TABLE III
GEOGRAPHICAL DISTRIBUTION OF PUBLICATIONS

Country	Papers
United States	71
China	17
Germany	12
United Kingdom	9
France	9
Austria	9
Switzerland	6
Singapore	5
Netherland	5
Italy	4
Korea	4

TABLE IV
TOP 10 COAUTHORS ON FUZZING

Author	Institution	Papers
P. Godefroid	Microsoft Research	9
S. Rawat	Vrije Universiteit Amsterdam	6
T. Y. Chen	Swinburne University of Technology	6
A. Arcuri	Simula Research Laboratory	4
C. Cadar	Stanford University	4
A. Groce	Oregon State University	4
B. P.Miller	University of Wisconsin	4
Y. Chen	University of Utah	3
V.T. Pham	National University of Singapore	3
K. Sen	University of California	3

technical reports (3%). Table II lists the venues where at least three fuzzing papers have been presented.

3) Geographical Distribution of Publications: We related the geographical origin of each primary study to the affiliation country of its first coauthor. Interestingly, we found that all 171 primary studies originated from 22 different countries with USA, China, and Germany being the top three, as presented in Table III (only the countries with over four papers). By continents, 43% of the papers are originated from America, 32% from Europe, 20% from Asia, and 5% from Oceania. This suggests that the fuzzing community is formed by a modest number of countries but fairly distributed around the world.

4) Researchers and Organizations: We identified 125 distinct coauthors in the 171 primary studies under review. Table IV presents the top authors on fuzzing and their most recent affiliation.

III. GENERAL PROCESS OF FUZZING

Fuzzing is a software testing technique which can automatically generate test cases. Thus, we run these test cases on a target program, and then observe the corresponding program behavior to determine whether there is any bug or vulnerability in the target program. The general process of fuzzing is shown in Fig. 2.

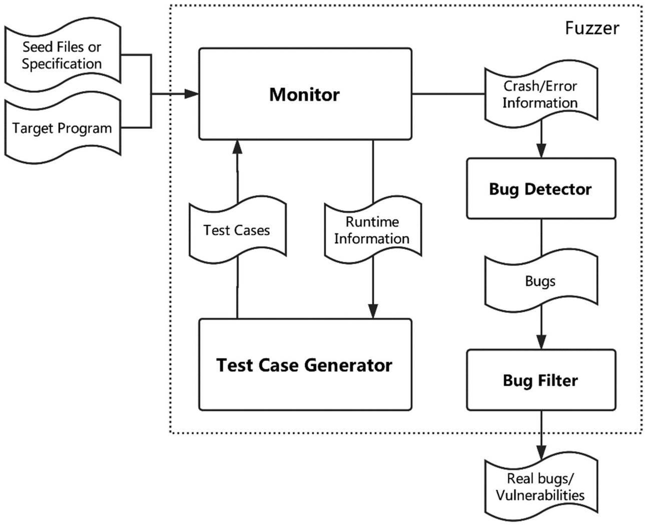


Fig. 2. General process of fuzzing.

Target program: A target program is the program under test. It could be either binary or source code. However, the source code of real-world software usually cannot be accessed easily, so fuzzers often target binary code.

Monitor: This component is generally built into a white-box or gray-box fuzzer. A monitor leverages the techniques, such as code instrumentation, taint analysis, etc., to acquire code coverage, taint data flow or other useful runtime information of the target program. A monitor is not necessary in a black-box fuzzer.

Test case generator: There are two main methods for fuzzers to generate test cases: mutation-based and grammar-based [1] methods. The first method generates test inputs by mutating well-formed seed files randomly or using predefined mutation strategies which can be adjusted based on target-program-oriented information gathered during runtime. On the contrary, the second method does not need any seed file. It generates inputs from a specification (e.g., grammar). In many cases, the test cases in fuzzing usually are semivalid inputs which are valid enough to pass the early parsing stage and invalid enough to trigger bugs in the deep logic of the target program.

Bug detector: To help users find potential bugs in a target program, a bug detector module is designed and implemented in a fuzzer. When the target program crashes or reports some errors, the bug detector module collects and analyzes related information (e.g., stack traces [22]) to decide whether a bug exists. Sometimes, a debugger can be used manually to record exception information [23]–[25] as an alternative of this module.

Bug filter: Testers usually focus on correctness or security related bugs. Therefore, filtering exploitable bugs (i.e., vulnerabilities) from all the reported bugs is an important task and usually performed manually [23], which is not only time consuming but hard to tackle as well. Currently, some research works [26] have proposed various approaches to mitigate this problem. For example, by sorting the fuzzer's outputs (bug-inducing test cases), the diverse, interesting test cases are prioritized, and testers do not need to manually search for wanted bugs, which is a process like looking for a needle in a haystack.

To explain the process of fuzzing more clearly, as an example, we use American fuzzy lop (AFL) [27], a mutation-based coverage-guided fuzzer, to test png2swf, a file converter. In this example, we provide the png2swf executable file for AFL as its target program. First, we provide a few seed files (the ideal seeds should be well formed and of small size) for AFL since it adopts the mutation-based technique. Second, we run AFL by a simple command (e.g., "afl-fuzz -i [input_directory] -o [output_directory] -Q – [target_directory] (@@)", if the target program gets input from a file, then "@@" is necessary). During the testing process, the "Monitor" component of AFL collects specific runtime information (path-coverage information in this case) by binary instrumentation, then passes these information to the "Test Case Generator" component to help guide its test case generation process. The general strategy is to save those test cases, which are able to cover new program paths for the next round of mutation, and discard those otherwise. Third, the newly generated test cases are passed back to the "Monitor" as the inputs of the target program. This process continues until we stop the AFL instance or a given time limit is reached. Besides, AFL also prints some useful information on the screen during the runtime, such as the execution time, the number of unique crashes, the execution speed, etc. So, we may get a set of test cases which can crash the target program. Finally, we analyze the test cases and identify those bugs that crash the target program manually or with the help of other tools. The bugs that AFL finds are mainly relative to memory operations, like buffer overflow, access violation, stack smash, etc., which usually cause the program crash or may be exploited by crackers.

IV. Black, White, or Gray?

Fuzzing techniques can be divided into three kinds: black box, white box, and gray box depending on how much information they require from the target program at runtime [28]. The information can be code coverage, data-flow coverage, program's memory usage, CPU utilization, or any other information to guide test case generation.

A. Black-Box Fuzzing

Traditional black-box fuzzing is also known as "black-box random testing." Instead of requiring any information from the target program or input format, black-box random testing uses some predefined rules to randomly mutate the given well-formed seed file(s) to create malformed inputs. The mutation rules could be bit flips, byte copies, or byte removals [28], etc. Recent black-box fuzzing also utilizes grammar [29] or input-specific knowledge [30] to generate semivalid inputs.

Black-box fuzzers, such as fuzz [31] and Trinity [32], are popular in software industry due to its effectiveness in finding bugs and simplicity for use. For instance, Trinity aims to fuzz system call interfaces of the Linux kernel. Testers should describe the type of input by using the provided templates first.

```
1 void test(char input[4]) {
2  int count = 0;
3  if (input[0] == 'o' ) count++;
4  if (input[1] == 'o' ) count++;
5  if (input[2] == 'p' ) count++;
6  if (input[3] == 's' ) count++;
7  if (count == 4) abort(); //error
8 }
```

Fig. 3. Example function.

Then, Trinity can generate more valid inputs which can reach higher coverage. This fuzzer has found plenty of bugs.[1]

However, the drawback of this technique is also obvious. Consider the function shown in Fig. 3. The *abort()* function at Line 7 has only 1/2^{32} chance to be reached when the *input* parameter at line 1 is assigned randomly. This example intuitively explains why it is difficult for black-box fuzzing to generate test cases that cover large number of paths in a target program. Due to this blindness nature, black-box fuzzing often provides low code coverage in practice [33]. That is why recent fuzzer developers have mainly focused on reverse engineering [34], code instrumentation [35], taint analysis [23], [36], [37], and other techniques to make the fuzzer "smarter" in order to mitigate this problem, which is why white-box and gray-box fuzzing have received more attention recently.

B. White-Box Fuzzing

White-box fuzzing is based on the knowledge of internal logic of the target program [17]. It uses a method which in theory can explore all execution paths in the target program. It was first proposed by Godefroid *et al.* [38]. In order to overcome the blindness of black-box fuzzing, they dedicated to exploring an alternative method and called it white-box fuzzing [14]. By using dynamic symbolic execution (also known as concolic execution [39]) and coverage-maximizing heuristic search algorithm, white-box fuzzing can search the target program thoroughly and quickly.

Unlike black-box fuzzing, white-box fuzzing requires information from the target program and uses the required information to guide test case generation. Specifically, starting execution with a given concrete input, a white-box fuzzer first gathers symbolic constraints at all conditional statements along the execution path under the input. Therefore, after one execution, the white-box fuzzer combines all symbolic constraints together using logic AND to form a path constraint (PC for short). Then, the white-box fuzzer systematically negates one of the constraints and solves the new PC. The new test case leads the program to run a different execution path. Using a coverage-maximizing heuristic search algorithm, white-box fuzzers can find bugs in the target program as fast as possible [38].

In theory, white-box fuzzing can generate test cases which cover all the program paths. In practice, however, due to many problems such as the numerous execution paths in a real software system and the imprecision of solving a constraint during symbolic execution (see Section V-B for details), the code coverage of white-box fuzzing cannot achieve 100%. One of the most famous white-box fuzzers is SAGE [14]. SAGE targets large-scale Windows applications and uses several optimizations to deal with the huge number of execution traces. It can automatically find software bugs and has achieved impressive results.

C. Gray-Box Fuzzing

Gray-box fuzzing stands in the middle of black-box fuzzing and white-box fuzzing to effectively reveal software errors with partial knowledge of the target program. The commonly used method in gray-box fuzzing is code instrumentation [40], [41]. By this means, a gray-box fuzzer can obtain code coverage of the target program at runtime; then, it utilizes this information to adjust (e.g., by using a genetic algorithm [28], [42]) its mutation strategies to create test cases which may cover more execution paths or find bugs faster. Another method used in gray-box fuzzing is taint analysis [23], [43]–[45], which extends code instrumentation for tracing taint data flow. Therefore, a fuzzer can focus on mutating specific fields of input which can influence the potential attack points in the target program.

Gray-box and white-box fuzzing are quite similar in that both methods make use of information of the target program to guide test case generation. But there is also an obvious difference between them: gray-box fuzzing only utilizes some runtime information (e.g., code coverage, taint data flow, etc.) of the target program to decide which paths have been explored [28]. Also, gray-box fuzzing only uses the acquired information to guide test case generation, but it cannot guarantee that using this piece of information will surely generate better test cases to cover new paths or trigger specific bugs. By contrast, white-box fuzzing utilizes source codes or binary codes of the target program to systemically explore all the execution paths. By using concolic execution and a constraint solver, white-box fuzzing can make sure that the generated test cases will lead the target program to explore new execution paths. Thus, white-box fuzzing helps reducing the blindness during fuzzing process more thoroughly. In summary, both methods make use of information of the target program to mitigate blindness of black-box fuzzing, but to different degrees.

BuzzFuzz [46] is a good example of showing how gray-box fuzzer works, although the developer of BuzzFuzz called it a white-box fuzzer. We regard this tool as a gray-box fuzzer, because it only acquires partial knowledge (taint data flow) of the target program. BuzzFuzz works as follows. First, it instruments the target program to trace tainted well-formed input data. Then, based on the collected taint propagation information, it calculates which part of the input data may influence the predefined attack points (BuzzFuzz regards lib calls as potential attack

[1] http://codemonkey.org.uk/projects/trinity/bugs-found.php

points) in the target program. Next, it mutates sensitive parts of the input data to create new test cases. Finally, it executes the new test cases and observes whether the target program crashes or not. By this means, BuzzFuzz can find bugs in a more target-oriented and effective manner. More importantly, attack points can be defined as specific lib calls, or vulnerability patterns, etc., depending on a developer's concern.

D. How to Choose?

According to the possibilities of being triggered or found, bugs can be classified into two categories: "shallow" bugs and "hidden" bugs. The bugs that cause the target program crash in the early stage of execution are regarded as "shallow" bugs, e.g., a potential divide-by-zero operation without any precedent conditional branch. On the contrary, the bugs which exist deeply in the program logic and are hard to trigger are regarded as "hidden" bugs, such as bugs existing in complex conditional branches. There is no standard way to identify "shallow" and "hidden" bugs; therefore, the commonly used evaluation criteria of a fuzzer are the code coverage it achieves, the number and exploitability of bugs it finds. In general, a traditional black-box fuzzer, which only uses random mutation method to generate test cases, cannot reach high code coverage and thus usually finds shallow bugs; however, it is lightweight, fast, and easy to use. By comparison, white-box or gray-box fuzzers can achieve higher code coverage and usually find more hidden bugs than black-box fuzzers, but these fuzzers are more complicated to build and the fuzzing processes are more time consuming than black-box fuzzers.

Fuzzing techniques that only use simple mutation methods are usually regarded as "dumb" fuzzing, such as the famous "5-lines of Python" method used by Charlie Miller,[2] which only randomly mutates some bytes of an input file without knowing its format. On the contrary, those techniques utilizing the input's specification or other knowledge or adopting runtime information (e.g., path coverage) to guide the test case generation are usually considered as "smart" fuzzing. In general, "dumb" and "smart" fuzzing methods provide different cost/precision tradeoffs and are suitable for different situations. For testers, which kind of fuzzer to choose mainly depends on two factors: 1) the type of the target program and 2) the test requirements (time/cost, etc.). In case that the input format of a target program (e.g., complier, system call, network protocol, etc.) is specific or strict, it will be more effective to choose a grammar-based fuzzer (many of this kind of fuzzers are black-box, such as Trinity [32] but developers recently built gray-box fuzzers targeting this sort of software, like Syzkaller [15]). In other cases, testers should consider more about the test requirements. If the primary goal of testing is efficiency rather than precision or high-quality output, black-box fuzzing will be a good choice. For instance, if a software system is not tested before and testers want to find and eliminate "shallow" bugs as quickly as possible, black-box fuzzing is a good start. On the contrary, if testers focus more on the quality of output (i.e., the variety, exploitability, and number of the discovered bugs) and want to achieve a higher code coverage, gray-box (sometimes white-box) fuzzing is usually more suitable [14]. Compared with gray-box fuzzing, white-box fuzzing is not very practical in industry (although SAGE is a famous white-box fuzzer in the market) because this technique is very expensive (time/resource-consuming) and faces many challenges (e.g., path explosion, memory modeling, constraint solving, etc.). However, white-box fuzzing is a popular research direction with lots of potential.

V. State of the Art in Fuzzing

According to the general process of fuzzing described in Section III, the following questions should be considered at the time of building a fuzzer:

1) how to generate or select seeds and other test cases;
2) how to validate those inputs against the specification of the target program;
3) how to deal with those crash-inducing test cases;
4) how to leverage runtime information;
5) how to improve the scalability of fuzzing.

In this section, we address RQ1 by summarizing the main contributions to above five issues of fuzzing in the literature. We review the papers related to seed generation and selection in Section V-A, input validation and coverage in Section V-B, runtime information and effectiveness in Section V-C, crash-inducing test cases handling in Section V-D, and scalability in fuzzing in Section V-E.

A. Seeds Generation and Selection

Given a target program to fuzz, the tester first needs to find the input interface (e.g., stdin file) so that the target program could read from a file, then determine the file formats which the target program can accept, and then choose a subset of collected seed files to fuzz the target program. The quality of seed files may highly influence the fuzzing result. Therefore, how to generate or select suitable seed files in order to discover more bugs is an important issue.

Some research works have been carried out to address this problem. Rebert *et al.* [47] tested six kinds of selection algorithms:

1) the set cover algorithm from Peach;
2) a random seed selection algorithm;
3) a minimal set cover (same as minset, which can be calculated by the greedy algorithm);
4) a minimal set cover weighted by size;
5) a minimal set cover weighted by execution time;
6) a hotset algorithm (which fuzz tests each seed file for *t* seconds, ranks them by the number of unique found bugs, and returns the top several seed files in the list).

By spending 650 CPU days on Amazon Elastic Compute Cloud for fuzzing ten applications, they drew the following conclusions.

1) The algorithms employing heuristics perform better than fully random sampling.
2) The unweighted minset algorithm performs the best among these six algorithms.

[2] https://fuzzinginfo.files.wordpress.com/2012/05/cmiller-csw-2010.pdf.

3) A reduced set of seed files performs more efficiently than the original set in practice.
4) Reduced set of seeds can be applied to different applications parsing the same file type.

Kargén and Shahmehri [48] claimed that by performing mutations on the generating program's machine code instead of directly on a well-formed input, the resulting test inputs are closer to the format expected by the program under test, and thus yield better code coverage. To test complex software (e.g., PDF readers) that usually take various inputs embedded with multiple objects (e.g., fonts, pictures), Liang *et al.* [49] leveraged the structure information of the font files to select seed files among the heterogeneous fonts. Skyfire [50] took as inputs a corpus and a grammar, and used the knowledge in the vast number of existing samples to generate well-distributed seed inputs for fuzzing programs that process highly structured inputs. An algorithm was presented in [51] to maximize the number of bugs found for black-box mutational fuzzing given a program and a seed input. The main idea behind this is to leverage white-box symbolic analysis on an execution trace for a given program-seed pair to detect dependences among the bit positions of an input, and then use this dependence relation to compute a probabilistically optimal mutation ratio for this program-seed pair. Furthermore, considering fuzzing as an instance of the coupon collector's problem in probability analysis, Arcuri *et al.* [52] proposed and proved nontrivial, optimal lower bounds for the expected number of test cases sampled by random testing to cover predefined targets, although how to achieve the bounds goal in practice is not given.

For random generation of unit tests of object-oriented programs, Pacheco *et al.* [53] proposed to use feedback obtained from executing the sequence as it is being constructed, in order to guide the search toward sequences that yield new and legal object states. Therefore, inputs that create redundant or illegal states are never extended. However, Yatoh *et al.* [54] argued that feedback guidance may overdirect the generation and limit the diversity of generated tests, and proposed an algorithm named feedback-controlled random testing, which controls the amount of feedback adaptively.

How can we get original seed files in the first place? For some open-source projects, the applications are published with a vast of input data for testing, which can be freely obtained as quality seeds for fuzzing. For example, FFmpeg automated testing environment (FATE) [55] provides various kinds of test cases which can be hard to collect on testers' own strength. Sometimes, the testing data are not publicly available, but the developers are willing to exchange it with people who will report program bugs in return. Some other open-source projects also provide format converters. Therefore, with a variety of file sets in certain format, the tester can obtain decent seeds for fuzzing by using format converter. For instance, cwebp [56] can convert TIFF/JPEG/PNG to WEBP images. Furthermore, reverse engineering is helpful to provide seed input for fuzzing. Prospex [57] can extract network protocol specifications including protocol state machines and use them to automatically generate input for a stateful fuzzer. Adaptive random testing (ART) [58] modifies random testing by sampling the space of tests and only executing those most distant, as determined by a distance metric over inputs, from all previously executed tests. ART has not always been shown to be effective for complex real-world programs [59], and has mostly been applied to numeric input programs.

Compared to the approaches mentioned above, gathering seed files by crawling the Internet is more general. Based on specific characters (e.g., file extension, magic bytes, etc.), testers can download required seed files. It is not a severe problem if the gathered corpus is huge, since storage is cheap and the corpus can be compacted to a smaller size while reaching equivalent code coverage [60]. To reduce the number of fault-inserted files and maintain the maximum test case coverage, Kim *et al.* [61] proposed to analyze fields of binary files by tracking and analyzing stack frames, assembly codes, and registers as the target software parses the files.

B. Input Validation and Coverage

The ability of automatically generating numerous test cases to trigger unexpected behaviors of the target program is a significant advantage of fuzzing. However, if the target program has input validation mechanism, these test cases are quite likely to be rejected in the early stage of execution. Therefore, how to overcome this obstacle is a necessary consideration when testers start to fuzz a program with such a mechanism.

1) Integrity Validation: During the transmission and storage, errors may be introduced into the original data. In order to detect these "distorted" data, the checksum mechanism is often used in some file formats (e.g., PNG) and network protocols (e.g., TCP/IP) to verify the integrity of their input data. Using the checksum algorithm (e.g., hash function), the original data are attached with a unique checksum value. For the data receiver, the integrity of the received data can be verified by recalculating checksum value with the same algorithm and comparing it to the attached one. In order to fuzz this kind of system, additional logic should be added into the fuzzer, so that the correct checksum values of newly created test cases can be calculated. Otherwise, the developer must utilize other methods to remove this obstacle. Wang *et al.* [36], [62] proposed a novel method to solve this problem and developed a fuzzer named TaintScope. TaintScope first uses dynamic taint analysis and predefined rules to detect potential checksum points and hot input bytes which can contaminate sensitive application programming interfaces (APIs) in the target program. Then, it mutates the hot bytes to create new test cases and changes the checksum points to let all created test cases pass the integrity validation. Finally, it fixes the checksum value of those test cases which can make the target program crash by using symbolic execution and constraint solving. By this means, it can create test cases which can both pass the integrity validation and cause the program to crash.

Given a set of sample inputs, Höschele and Zeller [63] used dynamic tainting to trace the data flow of each input character, and aggregated those input fragments into lexical and syntactical entities. The result is context-free grammar that reflects valid input structure, which is helpful for the later fuzzing process. To mitigate coverage-based fuzzers' limitation to exercise the paths protected by magic bytes comparisons, Steelix [64] leveraged lightweight static analysis and binary instrumentation to provide not only coverage information but comparison progress information to a fuzzer as well. Such program state information informs a fuzzer about where the magic bytes are located in the test input and how to perform mutations to match the magic bytes efficiently. There are some other efforts [65], [66], which have made progress in mitigating this problem.

2) Format Validation: Network protocols, compilers, interpreters, etc., have strict requirements on input formats. Inputs that do not meet format requirements will be rejected at the beginning of program execution. Therefore, fuzzing this kind of target systems needs additional techniques to generate test cases which can pass the format validation. Most solutions of this problem are to utilize input specific knowledge or grammar. Ruiter and Poll [30] evaluated nine commonly used transport layer security (TLS) protocol implementations by using black-box fuzzing in combination with state machine learning. They provided a list of abstract messages (also known as input alphabet) which can be translated by test harness into concrete messages sent to the system under test. Dewey *et al.* [67], [68] proposed a novel method to generate well-typed programs which use complicated type system by means of constraint logic programing (CLP) and applied their method to generate Rust or JavaScript programs. Cao *et al.* [69] first investigated the input validation situation of Android system services and for Android devices they built an input validation vulnerability scanner. This scanner can create semivalid arguments which can pass the preliminary check implemented by the method of target system service. There are some other works, such as [24], [25], [70], and [71], that focus on solving this problem.

3) Environment Validation: Only under certain environment (i.e., particular configurations, a certain runtime status/condition, etc.), will many software vulnerabilities reveal themselves. Typical fuzzing cannot guarantee the syntactic and semantic validity of the input, or the percentage of the explored input space. To mitigate these problems, Dai *et al.* [72] proposed the configuration fuzzing technique whereby the configuration of the running application is mutated at certain execution points, in order to check for vulnerabilities that only arise in certain conditions. FuzzDroid [73] can also automatically generate an Android execution environment where an app exposes its malicious behavior. FuzzDroid combines an extensible set of static and dynamic analyses through a search-based algorithm that steers the app toward a configurable target location.

4) Input Coverage: Tsankov *et al.* [74] defined semivalid input coverage (SVCov) that is the first coverage criterion for fuzz testing. The criterion is applicable whenever the valid inputs can be defined by a finite set of constraints. By increasing coverage under SVCov, they discovered a previously unknown vulnerability in a mature Internet Key Exchange (IKE) implementation. To address shortcomings of existing grammar inference algorithms, which are severely slow and overgeneralized, Bastani *et al.* [75] presented an algorithm for synthesizing a context-free grammar encoding the language of valid program inputs from a set of input examples and black-box access to the program. Unlike many methods which take the crash of the target program to determine whether an input is effective, ArtFuzz [76] aims at catching the noncrash buffer overflow vulnerabilities. It leverages type information and dynamically discovers likely memory layouts to help the fuzzing process. If a buffer border identified from the memory layout is exceeded, an error will be reported.

Considering increased diversity leads to improved coverage and fault detection, Groce *et al.* [77] proposed a low-cost and effective approach, called swarm testing, to increase the diversity of (randomly generated) test cases, which uses a diverse swarm of test configurations, each of which deliberately omits certain API calls or input features. Furthermore, to mitigate the inability to focus on part of a system under test, directed swarm testing [78] leveraged swarm testing and recorded statistical data on past testing results to generate new random tests that target any given source code element. Based on the observation that developers sometimes accompany submitted patches with a test case that partially exercises the new code, and this test case could be easily used as a starting point for the symbolic exploration, Marinescu and Cadar [79] provided an automatic way to generate test suites that achieve high coverage of software patches.

C. Handling Crash-Inducing Test Cases

Software companies (e.g., Microsoft) and open-source projects (e.g., Linux) often use fuzzing to improve the quality and reliability of their products [17]. Although fuzzing is good at generating crash-inducing test cases, it is usually not smart enough to analyze the importance of these test cases automatically. If fuzzing results in a bunch of raw crash-inducing test cases, it takes testers a large amount of time to find different bugs in the target program by analyzing these test cases. Due to time or budget constraints, developers prefer to fix those severe bugs.

Currently, only a few efforts focus on how to filter raw fuzzing outputs that make the fuzzing results more useful to testers. Given a large collection of test cases, each of which can trigger program bugs, Chen *et al.* [26] proposed a ranking-based method by which the test cases inducing *different* bugs are put on the top in the list. This ranking-based method is more practical compared with traditional clustering methods [80], [81]. Therefore, testers can focus on analyzing the top crash-inducing test cases. Besides filtering crash-inducing test cases directly, there are also some other methods to help reduce expensive manual work, such as generating unique crash-inducing test cases, trimming test cases, and providing useful debug information.

The uniqueness of crash-inducing test cases can be quite reliably determined by the call stack of the target thread and the address of fault-causing instruction [82]. If two distinct test cases cause the target program to crash with identical call stacks, it is very likely that these two test cases correspond to the same bug; therefore, only one of them needs to be reserved for manual analysis. On the contrary, if they cause the target program to crash at the same location but with different stack traces, it is quite possible that they correspond to two distinct bugs, and thus both of them are worth analyzing separately. Compared with recording the call stack, tracing execution path is a simpler but less reliable way to determine uniqueness. AFL [27], one of the most popular fuzzers, treats a crash-inducing test case as unique if it finds a new path or does not find a common path. This easy-to-implement approach is similar to the execution path record method which is the foundation of AFL. Producing unique crash-inducing test cases can help reduce the redundant fuzzing outputs, thus can save time and effort for manual analysis.

The size of test cases can greatly influence the efficiency of manual analysis, because larger test cases cost more time to execute and locate fault-causing instructions. For a mutation-based fuzzer, if low-quality seed files are used, the size of the generated test cases can be iteratively increased under some kinds of mutation methods. Therefore, during the process of fuzzing, periodically trimming the generated test cases can improve the overall efficiency, thus reducing the workload of manual analysis of crash-inducing test cases. The principle of trimming is simple: behavior of the processed one should be identical to the original one; in other words, they should follow the same execution path. The general steps of trimming are sequentially removing data blocks from a test case and re-evaluating the rest of the test case; those data blocks that cannot influence the execution path will be trimmed.

Evaluating the exploitability of fuzzing outputs usually needs code analysis and debugging work which can benefit from specialized tools, such as GDB, Valgrind [41], AddressSanitizer [83], etc. These tools provide runtime context (e.g., the state of call stack and registers, the address of fault-inducing instruction, etc.) of the target program or can detect particular kind of program fault such as memory error. With their assistance, testers are able to discover and evaluate program bugs more efficiently. Additionally, Pham *et al*. [84] presented a method for generating inputs which reach a given "potentially crashing" location. The test input generated by their method served as a witness of the crash.

D. Leveraging Runtime Information

As two of popular program analysis techniques, symbolic execution and dynamic taint analysis are often leveraged to make fuzzing smart because they can provide much runtime information (such as code coverage, taint data flow) and help fuzzing find "hidden" bugs [85], [86]. However, the problems they have also hinder fuzzing from pursuing higher efficiency [87]. In this section, we will discuss the problems, such as path explosion, imprecise symbolic execution faced by concolic execution and undertainting, and overtainting faced by dynamic taint analysis, and summarize the corresponding solutions. Knowing them can help readers have a more thorough understanding of smart fuzzing.

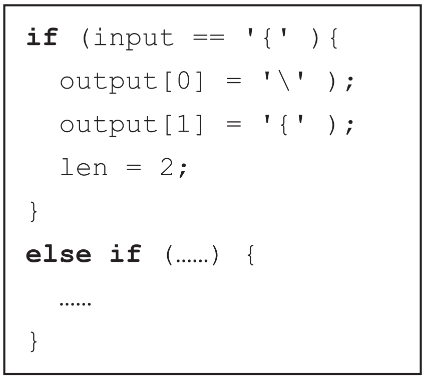
1) Path Explosion: Path explosion is an inherent and the toughest problem in symbolic execution because conditional branches in the target program are usually numerous, even a small-size application can produce a huge number of execution paths.

From the aspects of program analysis method and path search algorithm, several researches have tried to mitigate this problem. For example, function summaries [88], [89] are used to describe properties of low-level functions so that high-level functions can reuse them to reduce the number of execution paths. Redundant path pruning is used to avoid execution of those paths which have the same side effects with some previously covered paths. For instance, Boonstoppel *et al*. [90] proposed a technique for detecting and discarding large number of redundant paths by tracking read and write operations performed by the target program. The main idea behind this technique is that if a path reaches a program point under the same condition as some previously explored paths, then the path leading to an identical subsequent effect can thus be pruned. Besides, merging states [91] obtained on different paths can also reduce the path search space, but this method aggravates the solver's burden.

Heuristic search algorithms, on the other hand, can explore the most relevant execution paths as soon as possible within a limited time. For example, random path selection [92] and automatic partial loop summarization [93] were proven successful in practice mainly because they avoid being stuck when meeting some tight loop which can rapidly create new states. Another example is the control flow graph (CFG)-directed path selection [94], which utilizes static control flow graph to guide the test case generation to explore the closest uncovered branch. Experiment shows that this greedy approach can help improve coverage faster and achieve a higher percentage of code coverage. Besides, there is also generational search [38] which explores all subpaths of each expanded execution, scores them, and finally picks a path with the highest score for next execution. Considering that existing coverage-based gray-box fuzzing tools visit too many states in high-density regions; Böhme *et al*. [95] proposed and implemented several strategies to force AFL [27] to generate fewer inputs for states in a high-density region via visiting more states that are otherwise hidden in a low-density region. To mitigate the path explosion problem, DeepFuzz [96] assigned probabilities to execution paths and applied a new search heuristic that can delay path explosion effectively into deeper layers of the tested binary. Given a suite of existing test cases, Zhang *et al*. [97] leveraged test case reduction and prioritization methods to improve the efficiency of seeded symbolic execution, with the goal of gaining incremental coverage as quickly as possible.

2) Imprecise Symbolic Execution: The imprecision of symbolic execution is mainly caused by complicated program structures (e.g., pointers) modeling, library, or system calling and constraint solving. Some simplification methods are necessary to be carried out in the aforementioned areas in order to make symbolic execution practicable. Therefore, the key point for developers is to find a balance between scalability and precision.

Currently, some methods have been proposed to make symbolic execution more practical at the cost of precision. The operations of pointer are simplified in CUTE [98], which only considers equality and inequality predicates when dealing with symbolic pointer variables. Pointers are regarded as arrays in KLEE [92]. KLEE copies the present state N times, when a pointer p indirectly refers to N objects, and in each state, it implements proper read or write operation in the premise that p is not beyond the bounds of the corresponding object. It will use concrete values instead of symbolic values at the library or



```
if (input == '{' ){
  output[0] = '\' );
  output[1] = '{' );
  len = 2;
}
else if (……) {
  ……
}
```

Fig. 4. Code snippet that shows implicit data flow.

system call sites where the source code is not accessible [99]. In addition, constraint solving is added with lots of constraint optimizations [100] (e.g., SAGE uses unrelated constraint elimination, local constraint caching, flip count limit, etc., to improve the memory usage and speed when generating constraints) or even input grammar (e.g., Godefroid *et al.* [36] proposed an approach by which constraints based on input grammar can be directly generated by symbolic execution. Then, the satisfiability of these constraints will be verified by a customized constraint solver which also leverages input grammar. Therefore, it can generate highly structured inputs).

To fuzz testing floating-point (FP) code which may also result in imprecise symbolic execution, Godefroid and Kinder [102] combined a lightweight local path-insensitive “may” static analysis of FP instructions with a high-precision whole-program path-sensitive “must” dynamic analysis of non-FP instructions. Fu and Su [103] turned the challenge of testing FP code into the opportunity of applying unconstrained programming—the mathematical solution for calculating function minimum points over the entire search space. They derived a representing function from the FP code, any of whose minimum points is a test input guaranteed to exercise a new branch of the tested program.

3) Undertainting: Undertainting happens when implicit data flow in which data transmission is associated with control flow, and array operation, etc., are totally neglected. As shown in Fig. 4, during the conversion from plain text to Rich Text Format, the value of the *input* variable is transferred into *output* array without a direct assignment. Therefore, if *input* is tainted, neglecting this implicit data flow will cause undertainting. Kang *et al.* [44] have made a progress to solve this problem. According to their experiment, taint propagation for all implicit flows also leads to unacceptable-large overtainting. Therefore, they only focused on the taint propagation for complete information-preserving implicit flows like the example shown in Fig. 4.

4) Overtainting: Overtainting happens when taint propagation is not implemented at finer grained granularity. It causes taint explosion and false positive. Yadegari *et al.* [104] proposed an approach which fulfills taint propagation at bit level to mitigate this problem. Another method of dealing with this issue is to use underapproximations to check existential “for-some-path” properties. Godefroid [89] proposed a new approach to test generation where tests are derived from validity proofs of first-order logic formulas, rather than satisfying assignments of quantifier-free first-order logic formulas as most current approaches do. For more detailed information about symbolic execution and taint analysis, refer to professional literatures [39], [105].

E. Scalability in Fuzzing

Facing the size and complexity of real-world applications, modern fuzzers tend to be either scalable but not effective in exploring bugs located deeper in the target program, or not scalable yet capable of penetrating deeper in the application. Arcuri *et al.* [106] discussed the scalability of fuzzing, which under certain conditions, can fare better than a large class of partition testing techniques.

Bounimova *et al.* [107] reported experiences with constraint-based white-box fuzzing in production across hundreds of large Windows applications and over 500 machine years of computation from 2007 to 2013. They extended SAGE with logging and control mechanisms to manage multimonth deployments over hundreds of distinct application configurations. They claimed their work as the first production use of white-box fuzzing and the largest scale deployment of white-box fuzzing to date.

Some approaches leveraged application-aware fuzzing or fuzzing by cloud services methods for this issue. Rawat *et al.* [108] presented an application-aware evolutionary fuzzing strategy that does not require any prior knowledge of the application or input format. In order to maximize coverage and explore deeper paths, they leveraged control- and data-flow features based on static and dynamic analysis to infer fundamental properties of the application, which enabled much faster generation of interesting inputs compared to an application-agnostic approach.

A method for improving scalability is to reduce the scope of the analysis. Regression analysis is a well-known example where the differences between program versions serve as the basis to reduce the scope of the analysis. DiSE [109] combines two phases: static analysis and symbolic execution. The set of affected program instructions is generated in the first phase. The information generated by the static analysis is then used to direct symbolic execution to explore only the parts of the programs affected by the changes, potentially avoiding a large number of unaffected execution paths.

To test mobile apps in a scalable way, every test input needs to be running with a variety of contexts, such as: device heterogeneity, wireless network speeds, locations, and unpredictable sensor inputs. The range of values for each context, e.g., location, can be very large. Liang *et al.* [110] presented Caiipa, a cloud service for testing apps over an expanded mobile context space in a scalable way and implemented on a cluster of VMs and real devices that can emulate various combinations of contexts for mobile apps. Mobile vulnerability discovery pipeline (MVDP) [111] is also a distributed black-box fuzzing system for Android and iOS devices.

VI. Tools in Different Application Areas

Fuzzing is a practical software testing technique which has been widely applied in industry. Any software accepting user inputs can be regarded as a fuzzing target. Currently, there are various fuzzers targeting different software systems. In this

TABLE V
SUMMARIES OF THE TYPICAL FUZZERS

Name	Birth Year	Targets	Key Techniques	Platforms	Availability
Peach	2004	General purpose	Black-box mutation/generation fuzzing	Linux, Windows, MacOS	Open-source
Trinity	2004	OS Kernels	Input knowledge based black-box generation fuzzing	ARM, i386, x86-64, etc.	Open-source
beSTORM	2005	General purpose	Black-box generation fuzzing	Linux, Windows	Commercial
jsfunfuzz	2007	JavaScript engine in Firefox	Grammar-based black-box generation fuzzing, Differential testing	Linux, MacOS, Windows	Open-source
SAGE	2008	Large Windows applications	White-box generation fuzzing	Windows	Microsoft internal
Sulley	2009	Network protocols	Input knowledge based black-box generation fuzzing	Windows	Open-source
IOCTL fuzzer	2009	Kernel drivers	Input knowledge based black-box generation fuzzing	Windows	Open-source
Csmith	2011	C compilers	Grammar-based black-box generation fuzzing, Differential testing	Linux, FreeBSD, MacOS, Windows	Open-source
LangFuzz	2012	Language-agnostic interpreters (JavaScript, PHP)	Grammar-based black-box generation/mutation fuzzing	Linux	Closed-source
AFL	2013	Applications	Coverage-guide gray-box fuzzing, Genetic algorithm	Linux, Free- /Open-/NetBSD, MacOS, Solaris	Open-source
YMIR	2013	ActiveX control	Generation of fuzzing grammars using API-level concolic testing	Windows	Closed-source
vUSBf	2014	USB drivers	Input knowledge based black-box generation fuzzing	Linux	Open-source
CLsmith	2015	Many core openCL compilers	Grammar-based black-box generation fuzzing, Random differential testing and EMI testing	Linux	Open-source
Syzkaller	2016	OS Kernels	Coverage-guide gray-box generation/mutation fuzzing	Linux	Open-source
TLS-Attacker	2016	Network protocols	Grammar-based black-box mutation fuzzing, using modifiable variables and two-stage fuzzing	Linux, MacOS, Windows	Open-source
QuickFuzz	2016	Applications	Grammar-based black-box generation/mutation fuzzing	Linux	Open-source
CAB-FUZZ	2017	OS Kernels	gray-box fuzzing, using concolic testing	Linux, MacOS, Windows	Unknown
kAFL	2017	OS Kernels	Coverage-guide gray-box fuzzing, using hypervisor and Intel's Processor Trace technology	Linux, MacOS, Windows	Open-source

section, we answer RQ2 by investigating some of the most popular and efficient fuzzers classified by their application areas, i.e., the type of their target software platform. Tables V and VI summarize the typical fuzzers we are going to introduce, from application areas and problem domains, respectively.

A. General Purpose Fuzzers

1) Peach: Peach [112] is a well-known general purpose fuzzer of which the most common targets are drivers, file consumers, network protocols, embedded devices, systems, etc. It is constructed with the following components: 1) predefined input format definitions called Peach Pits, which are available as individual Pits or groups of related Pits called Pit Packs; 2) test passes, which can weight mutators to perform more test cases; and 3) minset, which helps pare down the file count for test case coverage. Peach plays a critical role in this area because it has many outstanding features, such as threat detection, out-of-the-box fuzzing definitions (Peach Pits), and scalable testing options. However, there are also some problems in Peach (specifically in open-source version). A major one is that it is time consuming to build Pit file using syntax provided by Peach to describe the target file format. Built on Peach, eFuzz [113] tested smart metering devices based on the communication protocol DLMS/COSEM, the standard protocol used in Europe, for possible faults. Honggfuzz [114] is also built on Peach.

2) beSTORM: beSTORM [115] is a commercial black-box fuzzer. It can be used to test the security of target application

TABLE VI
TYPICAL FUZZERS AND THEIR PROBLEM DOMAINS

Name	Seeds generation and selection	Input validation and coverage	Handling crash-inducing test cases	Leveraging runtime information	Scalability in fuzzing
Peach	√	√	○	×	√
Trinity	/	√	○	×	√
beSTORM	/	√	○	×	√
jsfunfuzz	/	√	○	×	√
SAGE	/	√	○	√	√
Sulley	/	√	○	×	√
IOCTL fuzzer	/	√	○	×	√
Csmith	/	√	×	×	√
LangFuzz	/	√	○	×	○
AFL	○	×	×	√	√
YMIR	/	√	○	×	○
vUSBf	/	√	○	×	○
CLsmith	/	√	○	×	○
Syzkaller	/	√	○	√	√
TLS-Attacker	/	√	○	×	√
QuickFuzz	/	√	×	×	√
CAB-FUZZ	/	×	○	√	○
kAFL	×	○	×	√	○

The meaning of the symbols in table is as follows:
1) Seeds generation and selection
√: The fuzzer can automatically generate seeds or adopt some seed selection algorithm.
○: The fuzzer provides some high-quality seeds for testers to choose (usually for mutation-based fuzzers).
×: The seeds are collected by testers manually.
/: The fuzzer does not require seeds in random (usually for grammar-based fuzzers).
2) Input validation and coverage
√: The fuzzer utilizes input grammars or other knowledge to generate test cases, or adopts some ways to pass through input validation.
○: The fuzzer uses some methods to mitigate the problem caused by input validation.
×: The fuzzer does not use any input information or approach to pass through input validation.
3) Handling crash-inducing test cases
√: The fuzzer can analyze the found bugs automatically and generate a detailed bug report.
○: The fuzzer can provide some useful information, like log file, to help subsequent bug analysis.
×: The fuzzer only generates crash-inducing test cases.
4) Leveraging runtime information
√: The fuzzer uses runtime information to guide test case generation.
×: The fuzzer does not use any feedback information during runtime.
5) Scalability in fuzzing
√: The fuzzer can test real-world applications effectively and has found plenty of bugs.
○: The fuzzer is in its experiment period and applied to some real-world programs.
×: The fuzzer can only test some experimental programs.

or to check the quality of networked hardware and software products. It does not need source code but only binaries of the target program. The fuzzing strategy beSTORM takes is to first test the likely, common failure-inducing area and then expand to a near-infinite range of attack variations; therefore, it can quickly deliver a result. beSTORM can be used in the test of protocols, applications, hardware, files, WIFI, and embedded device security assurance (EDSA). For example, it is able to find bugs in the application which implements the EDSA 402 Standards.

B. *Fuzzers for Compilers and Interpreters*

1) jsfunfuzz: jsfunfuzz [116] is a grammar-based black-box fuzzer designed for Mozilla's SpiderMonkey JavaScript engine. It is the first publicly available JavaScript fuzzer. Since developed in 2007, it has found over 2000 bugs in SpiderMonkey. It combines differential testing with detailed knowledge of the target application; therefore, it can efficiently find both correctness-related bugs and crash-triggering bugs in different JavaScript engines. However, for each new language feature, jsfunfuzz has to adjust itself to respond to the new feature in the fuzzing process.

2) Csmith: Csmith was proposed by Yang *et al*. [117] in 2011. It is a C compiler fuzzer which can generate random C programs with specified grammar according to the C99 standard. It utilizes random differential testing [118] to help find correctness bugs which are caused by potentially undefined behavior and other C-specific problems. Csmith has been used for years and has found hundreds of previously unknown bugs in both commercial and open source C compilers (e.g., GNU Compiler Collection (GCC), low level virtual machine (LLVM)). Since it is an open-source project, the latest information and version about Csmith can be accessed at [119]. Although Csmith is a practical fuzzer which is good at generating error-inducing test cases, as many other fuzzers do, it does not prioritize the bugs found according to the importance. Therefore, testers must spend a lot of time in deciding the novelty and the severity of each bug.

3) LangFuzz: Inspired by jsfunfuzz, Holler *et al*. [120] presented LangFuzz in 2012. LangFuzz does not aim at a particular language. So far, it has been tested on JavaScript and hypertext preprocessor (PHP). After applied to JavaScript engine, LangFuzz has found more than 500 previously unknown bugs in SpiderMonkey [26]. Applied to PHP interpreter, it also discovered 18 new defects that can result in crash. LangFuzz makes use of both stochastic generation and code mutation to create test cases, but regards mutation as the primary technique. It is designed as a language-independent fuzzer, but adapting it to a new language needs some necessary changes.

4) CLsmith: Lidbury *et al*. [70] leveraged random differential testing and equivalence modulo inputs (EMI) testing to fuzzing many-core compiler and identified and reported more than 50 OpenCL compiler bugs, which is the most in commercial implementations. Specifically, they employed random differential testing to the many-core setting for generating deterministic, communicating, feature-rich OpenCL kernels, and proposed and evaluated injection of dead-by-construction code to enable EMI testing in the context of OpenCL.

Other fuzzers in this category include MongoDB's JavaScript Fuzzer which detected almost 200 bugs over the course of two release cycles [121]; Ifuzzer is another JavaScript interpreter fuzzer using genetic programming [122].

C. Fuzzers for Application Software

1) SAGE: SAGE [14] is a well-known white-box fuzzer developed by Microsoft. It is used to fuzz large file-reading Windows applications (e.g., document parsers, media players, image processors, etc.) running on x86 platform. Combining concolic execution with a heuristic search algorithm to maximize code coverage, SAGE tries its best to reveal bugs effectively. Since 2008, this tool has been running incessantly on an average of 100-plus machines/cores and fuzzing automatically a few hundred applications of Microsoft. SAGE is the first fuzzer realizing the white-box fuzzing technique and able to test real-world applications. Nowadays, Microsoft is promoting an online fuzzing project called Springfield [123]. It provides multiple methods including Microsoft white-box fuzzing technology to find bugs in the binary programs uploaded by customers. Future work of SAGE consists of improving its search method, enhancing the precision of its symbolic execution and increasing the capability of its constraint solving for discovering more bugs [38].

2) AFL: AFL [27] is a well-known code-coverage guide fuzzer. It gathers the information of runtime path coverage by code instrumentation. For open-source applications, the instrumentation is introduced at compile time and for binaries the instrumentation is introduced at runtime via a modified QEMU [124]. The test cases which can explore new execution paths have more chance to be chosen in the next round of mutation. The experimental result shows that AFL is efficient at finding bugs in real-world use cases, such as file compression libraries, common image parsing, and so on. AFL supports C, C++, Objective C, or executable programs and works on Linux-like OS. Besides, there is also some work to extend the application scenarios of AFL, such as TriforceAFL [125], which is used to fuzz kernel syscalls, WinAFL [126], which ports AFL to Windows, and the work from ORACLE [127], which uses AFL to fuzz some filesystems. Although AFL is efficient and easy to use, there is still room to improve. Like many other brute-force fuzzers, AFL provides limited code coverage when the actual input data are compressed, encrypted, or bundled with checksum. Besides, AFL takes more time when dealing with 64-bit binaries and does not support fuzzing network services directly.

3) QuickFuzz: QuickFuzz [128] leverages Haskell's QuickCheck (the well-known property-based random testing library) and Hackage (the community Haskell software repository) in conjunction with off-the-shelf bit-level mutational fuzzers to provide automatic fuzzing for more than a dozen common file formats, without providing external set of input files nor developing models for the file types involved. QuickFuzz generates invalid inputs using a mix of grammar-based and mutation-based fuzzing techniques to discover unexpected behavior in a target application.

To test the server side software, Davis *et al*. [129] presented Node.fz, which is a scheduling fuzzer for event-driven programs, embodied for server-side Node.js programs. Node.fz randomly perturbs the execution of a Node.js program, allowing Node.js developers to explore a variety of possible schedules. Work in this category also includes Dfuzzer [130], which is a fuzzer for D-bus service.

To test mobile applications, some fuzzers have been presented in recent years, such as Droid-FF [131], memory-leak fuzzer [132], DroidFuzzer [133], intent fuzzer [134], and Android Ripper MFT tool [135] for Android apps.

D. Fuzzers for Network Protocols

1) Sulley: Sulley [136] is an open-source fuzzing framework targeting network protocols. It utilizes a block-based approach to generate individual "requests." It provides lots of needed data formats for users to build protocol descriptions. Before testing, users should use these formats to define all necessary blocks which will be mutated and merged in the fuzzing process to create new test cases. Sulley can classify the detected faults, work in parallel, and trace down to a unique sequence of a test case triggering a fault. However, it is currently not well maintained. Boofuzz [137] is a successor to Sulley.

2) TLS-Attacker: Somorovsky [138] presented TLS-Attacker, an open-source framework for evaluating the security of TLS libraries. TLS-Attacker allows security engineers to create customized TLS message flows and arbitrarily modify message contents by using a simple interface to test the behavior of their libraries. It successfully found several vulnerabilities in widely used TLS libraries, including OpenSSL, Botan, and MatrixSSL.

There are some other works about fuzzing network protocols [139], [140]. T-Fuzz [141] is a model-based fuzzer for robustness testing of telecommunication protocols, Secfuzz [142] for IKE protocol, and both SNOOZE [143] and KiF [144], [145] for the VOIP/SIP protocol.

E. Fuzzers for OS Kernels

Kernel components in OS are difficult to fuzz as feedback mechanisms (i.e., guided code coverage) cannot be easily applied. Additionally, nondeterminism is due to interrupts, kernel threads, and statefulness poses problems [146]. Furthermore, if a process fuzzes its own kernel, a kernel crash highly impacts the performance of the fuzzer because of the reboot of the OS.

1) Trinity: In recent years, Trinity [147] has gained a lot of attention in the area of kernel fuzzing. It implements several methods to send syscalls semi-intelligent arguments. The methods used to generate arguments of system call are described as follows: 1) If a system call expects a certain data type as an argument (e.g., descriptor), it gets passed one; 2) if a system call only accepts certain values as an argument (e.g., a 'flags' field), it has a list of all the valid flags that may be passed; and 3) if a system call only takes a range of values, a random value passed to an argument usually fits that range. Trinity supports a variety of architectures including x86-64, SPARC-64, S390x, S390, PowerPC-64, PowerPC-32, MIPS, IA-64, i386, ARM, Aarch64, and Alpha.

2) Syzkaller: Syzkaller [15] is another fuzzer targeting Linux kernels. It depends on predefined templates which specify the argument domains of each system call. Unlike Trinity, it also makes use of code coverage information to guide the fuzzing process. Because Syzkaller combines the coverage-guided and template-based techniques, it does work better than provided only with the pattern of argument usages for system calls. This tool is under active development but the early results look impressive.

3) IOCTL Fuzzer: IOCTL Fuzzer [148] is a tool designed to automatically search for vulnerabilities in Windows kernel drivers. Currently, it supports Windows 7 (x32 and x64), 2008 Server, 2003 Server, Vista, and XP. If an IOCTL operation is conformed to conditions specified in the configuration file, the fuzzer replaces its input field with randomly generated data.

4) Kernel-AFL (kAFL): Schumilo *et al*. [149] proposed coverage-guided kernel fuzzing in an OS-independent and hardware-assisted way. They utilize a hypervisor to produce coverage and Intel's Processor Trace technology to provide control flow information on running code. They developed a framework called kAFL to assess the reliability or security of Linux, MacOS, and Windows kernel components. Among many crashes, they uncovered several flaws in the ext4 driver for Linux, the HFS and APFS file system of MacOS, and the NTFS driver of Windows.

5) CAB-FUZZ: To discover the vulnerabilities of commercial off-the-shelf (COTS) operating systems (OSes), Kim *et al.* [150] proposed CAB-FUZZ, a practical concolic testing tool to explore relevant paths that are most likely triggering bugs. This fuzzer prioritized the boundary states of arrays and loops and exploited real programs interacting with COTS OSes to construct proper contexts to explore deep and complex kernel states without debug information. It found 21 undisclosed unique crashes in Windows 7 and Windows Server 2008, including three critical vulnerabilities. Five of those found vulnerabilities have been existing for 14 years and could be triggered even in the initial version of Windows XP.

F. Fuzzers for Embedded Devices, Drivers and Components

1) YMIR: Kim *et al.* [29] proposed the automatic generation of fuzzing grammars using API-level concolic testing, and implemented a tool (named YMIR) to automate white-box fuzz testing on ActiveX controls. It takes an ActiveX control as input and delivers fuzzing grammars as its output. API-level concolic testing collects constraints at the library function level rather than the instruction level, and thus may be faster and less accurate.

2) vUSBf: vUSBf [151] was first proposed at Black Hat Europe 2014. It is a fuzzing framework for USB drivers. This framework implements a virtual USB fuzzer based on Kernel Virtual Machine (in Linux) and the USB redirection protocol in QEMU. It allows the dynamic definition of several million test cases using a simple XML configuration. Each test is marked using a unique identification and thus is reproducible. It can trigger the following bugs in Linux kernels and device drivers: null-pointer dereferences, kernel paging requests, kernel panic, bad page state, and segmentation fault. There are some other works in this area, such as a cost-effective USB testing framework [152], VDF–a targeted evolutionary fuzzer of virtual devices [153].

Besides the aforementioned fuzzers, there are also many other practical tools, including perf_fuzzer [154] for perf_event_open() system call, libFuzzer [155] for library, Modbus/TCP Fuzzer for internetworked industrial systems [156], a fuzzer for I/O buses [157], a fuzzer for digital certificates [158], Gaslight [159] for memory forensics frameworks, etc. Moreover, along with memory error detectors (e.g., Clang's AddressSanitizer [83], MemorySanitizer [160], etc.), fuzzers can be reinforced to expose more hidden bugs rather than shallow bugs.

VII. Future Directions

In this section, we answer RQ3 by discussing some of the possible future directions of the fuzzing technique. Although we cannot accurately predict the future directions that the study of fuzzing will follow, it is possible for us to identify and summarize some trends based on the reviewed papers, which may suggest and guide directions of future research. We will discuss the future work in the following directions, with the hope that the discussion could inspire follow-up researches and practices.

A. Input Validation and Coverage

Overly complex, sloppily specified, or incorrectly implemented input languages, which describe the set of valid inputs an application has to handle, are the root causes of many security vulnerabilities [161]. Some systems are strict on input formats (e.g., network protocols, compilers and interpreters, etc.); inputs that do not satisfy the format requirement will be rejected in the early stage of execution. In order to fuzz this kind of target program, the fuzzer should generate test cases which can pass the input validation. Many researches targeted this problem and made an impressive progress, such as [162] for string bugs, [163], [164] for integer bugs, [165] for e-mail filters, and [166] for buffer bugs. Open issues in this area include dealing with FP operations (e.g., Csmith, well-known C compiler fuzzer, does not generate FP programs), applying existing techniques to other languages (e.g., applying CLP on C language), etc. Furthermore, Rawat *et al.* [108] demonstrated that inferring input properties by analyzing application behavior is a viable and scalable strategy to improve fuzzing performance, as well as a promising direction for future research in the area. As we mentioned in Section V-A, although TaintScope can locate the checksum points accurately and increase the effectiveness of fuzzing dramatically, there is still room for improvement. First, it cannot deal with digital signature and other secure check schemes. Second, its effectiveness will be highly influenced by encrypted input data. Third, it ignores control flow dependences and does not instrument all kinds of ×86 instructions. These are still open problems.

B. Smart Fuzzing

Many other program analysis techniques are merged into smart fuzzing [167], [168], such as concolic execution, dynamic taint analysis, and so on. Although these techniques bring many benefits, they also cause some problems: such as path explosion, imprecise symbolic execution in concolic test and undertainting, overtainting in dynamic taint analysis. As an example, Dolan-Gavitt *et al.* [169] have injected thousands of bugs into eight real-world programs, including bash, tshark, and the GNU Coreutils. They evaluated and found that a prominent fuzzer and a symbolic execution-based bug finder were able to locate some but not all injected bugs. Furthermore, fuzzing in a scalable and efficient way is still challenging. Bounimova *et al.* [107] presented key challenges with running white-box fuzzing on a large scale, which involve those challenges in symbolic execution, constraint generation and solving, long-running state-space searches, diversity, fault tolerance, and always-on usage. These problems all deserve to be studied in more depth.

C. Filtering Fuzzing Outputs

During a software development life cycle, time and budget for fixing bugs are usually constrained. Therefore, the main concern of the developer is to solve those severe bugs under these constraints. For example, Podgurski *et al.* [80] proposed automated support for classifying reported software failures to facilitate prioritizing them and diagnosing their causes. Zhang *et al.* [170] proposed to select test cases based on test case similarity metric to explore deep program semantics. Differential testing may be helpful to determine the cost of evaluating test results [171], [172]. In short, at present, there has been little research on filtering more important failure-inducing test cases from the large fuzzing outputs. This research direction is of practical importance.

D. Seed/Input Generation and Selection

The result of fuzzing is correlated with the quality of seed/input files. Therefore, how to select suitable seed files in order to find more bugs is an important issue. By attempting to maximize the testing coverage of the input domain, the methods or algorithms of dealing with test cases in ART [58], [173], [174]–[176] may be useful. For example, Pacheco *et al.* [53] presented a feedback-directed random test generation technique, where a predesigned input was executed and checked against a set of contracts and filters. The result of the execution determines whether the input is redundant, illegal, contract violating, or useful for generating more inputs. However, in the massive experiments, Arcuri and Briand [59] showed that ART was highly inefficient even on trivial problems when accounting for distance calculations among test cases. Classfuzz [177] mutated seeding class-files using a set of predefined mutation operators, employed Markov Chain Monte Carlo sampling to guide mutator selection, and used coverage uniqueness as a discipline for accepting representative ones. Shastry *et al.* [178] proposed to automatically construct an input dictionary by statically analyzing program control and data flow, and the input dictionary is supplied to an off-the-shelf fuzzer to influence input generation. To design and implement more effective, sound, and accurate seed generation and selection algorithms is an open research problem.

E. Combining Different Testing Methods

As discussed in Section IV, black-box and white-box/gray-box fuzzing methods have their own advantages and disadvantages. Therefore, how to combine these techniques to build a fuzzer which is both effective and efficient is an interesting research direction. There are a few attempts [124], [179], [48] in this area, for example, SYMFUZZ [51] augmented black-box mutation-based fuzzing with a white-box technique, which helps to calculate an optimal mutation ratio based on the given program-seed pairs. From microperspective, SYMFUZZ has two main steps to generate test cases, and each step uses a different fuzzing techniques which are white-box fuzzing and black-box fuzzing. However, from a macroperspective, this method can also be regarded as gray-box fuzzing. Because the mutation ratio of its black-box fuzzing process is computed during its white-box fuzzing process, so the whole fuzzing utilizes partial knowledge of the target program and can be regarded as gray-box fuzzing. It is also an interesting direction to combine fuzzing with other testing techniques. Chen *et al.* [180] reported how metamorphic testing [181]–[183], which is a relatively new testing method which looks at the relationships among the inputs and outputs of multiple program executions, detected previously unknown bugs in real-world critical applications, showing that using diverse perspectives and combining multiple methods can help software testing achieve higher reliability or security. Garn and Simos [184] showed the applicability of a comprehensive method utilizing combinatorial testing and fuzzing to system call interfaces of Linux kernel.

F. Combining Other Techniques With Fuzzing

Fuzzers are limited by the extent of test coverage and the availability of suitable test cases. Since static analysis can perform a broader search for vulnerable code patterns, starting from a handful of fuzzer-discovered program failures, Shastry *et al.* [185] implemented a simple yet effective match-ranking algorithm which used test coverage data to focus attention on matches which comprised untested code and demonstrated that static analysis could effectively complement fuzzing. Static analysis techniques, such as symbolic execution and control/data flow analysis, can provide useful structural information for fuzzing [186]; however, there are some limitations of symbolic execution for fuzzing thus leaving some open problems:

1) Only generic properties are checked—many deviations from a specified behavior are not found; and 2) many programs are not entirely amenable to symbolic execution because they give rise to hard constraints so that some parts of a program remain uncovered [187]. Havrikov [188] proposed a combination approach by which fuzzing can benefit from different lightweight analyses. The analyses leveraged multiple information sources apart from the target program, such as input and execution (e.g., descriptions of the targeted input format in the form of extended context-free grammars) or hardware counters.

Machine learning techniques are helpful for automatically generating input grammars for grammar-based fuzzing [189]. Optimization theory can also be used to build effective search strategies in fuzzing [83], [190], [191]. Genetic programming or algorithm is used in [122], [192], and [193] to guide their fuzzers. Dai *et al*. [194] proposed a novel UI fuzzing technique aiming at running apps such that different execution paths can be exercised, and this method required the tester to build a comprehensive network profile. We believe that there is still some room for improving the existing combination methods and leveraging other techniques with fuzzing.

VIII. Conclusion

Fuzzing is an automatic and effective software testing technique which is able to discover both correctness and security bugs. It can be classified into black-box, white-box, and gray-box categories according to how much information it acquires from the target program. During the fuzzing process, the basic way to find bugs is to generate numerous test cases which are hopeful to trigger bug-inducing code fragments in the target program. However, there is no fixed pattern in fuzzing to generate test cases, and thus it mostly depends on the developers' creativity. We presented a survey on fuzzing covering 171 papers published between January 1990 and June 2017. The results of the survey show that fuzzing is a thriving topic with an increasing trend of contributions on the subject. Currently, merged techniques with fuzzing include genetic algorithm, taint analysis, symbolic execution, coverage-guided methods, etc. Existing fuzzing tools have been widely applied in many kinds of industry products including compilers, network protocol, applications, kernels, etc., ranging from binaries to source codes and found tens of thousands of software bugs, many of which are exploitable. Last but not least, we discuss some open problems with regard to fuzzing. We encourage further research and practices to address these problems toward a wide adaption of fuzzing in continuous integration of a software system.

Acknowledgment

The authors thank the anonymous reviewers for their valuable comments that have helped improve this paper.

References

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

[2] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Commun. ACM*, vol. 33, pp. 32–44, 1990.

[3] B. P. Miller *et al*., "Fuzz revisited: A re-examination of the reliability of UNIX utilities and services," Dept. Comput. Sci., Univ. Wisconsin-Madison, Madison, WI, USA, Tech. Rep. #1268, 1995.

[4] J. E. Forrester and B. P. Miller, "An empirical study of the robustness of windows NT applications using random testing," in *Proc. 4th Conf. USENIX Windows Syst. Symp.*, Seattle, WA, USA, vol. 4, 2000, pp. 1–10.

[5] B. P. Miller, G. Cooksey, and F. Moore, "An empirical study of the robustness of MacOS applications using random testing," in *Proc. Int. Workshop Random Test.*, 2006, pp. 46–54.

[6] R. Hamlet, "Random testing," in *Encyclopedia of Software Engineering*. New York, NY, USA: Wiley, 1994, pp. 970–978.

[7] G. McGraw, "Silver bullet talks with Bart Miller," *IEEE Security Privacy*, vol. 12, no. 5, pp. 6–8, Sep. 2014.

[8] J. Viide *et al*., "Experiences with model inference assisted fuzzing," in *Proc. Conf. USENIX Workshop Offensive Technol.*, 2008, Art. no. 2.

[9] H. Yang, Y. Zhang, Y. Hu, and Q. Liu, "IKE vulnerability discovery based on fuzzing," *Security Commun. Netw.*, vol. 6, no. 7, pp. 889–901, 2013.

[10] J. Yan, Y. Zhang, and D. Yang, "Structurized grammar-based fuzz testing for programs with highly structured inputs," *Security Commun. Netw.*, vol. 6, no. 11, pp. 1319–1330, 2013.

[11] N. Palsetia, G. Deepa, F. A. Khan, P. S. Thilagam, and A. R. Pais, "Securing native XML database-driven web applications from XQuery injection vulnerabilities," *J. Syst. Softw.*, vol. 122, pp. 93–109, 2016.

[12] M. de Jonge and E. Visser, "Automated evaluation of syntax error recovery," in *Proc. 27th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2012, pp. 322–325.

[13] J. D. DeMott, R. J. Enbody, and W. F. Punch, "Systematic bug finding and fault localization enhanced with input data tracking," *Comput. Security*, vol. 32, pp. 130–157, 2013.

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

[15] D. Vyukov, *Syzkaller—Linux Kernel Fuzzer*. [Online]. Available: https://github.com/google/syzkaller. Accessed on: Jul. 12, 2016.

[16] D. Babic, "SunDew: Systematic automated security testing," in *Proc. 24th ACM SIGSOFT Int. SPIN Symp. Model Checking Softw.*, Santa Barbara, CA, USA, 2017, p. 10.

[17] J. DeMott, "The evolving art of fuzzing," in *Proc. DEF CON Conf.*, vol. 14, 2006, pp. 1–25.

[18] R. McNally, K. Yiu, D. Grove, and D. Gerhardy, "Fuzzing: The state of the art," DTIC Document, 2012.

[19] T. L. Munea, H. Lim, and T. Shon, "Network protocol fuzz testing for information systems and applications: A survey and taxonomy," *Multimed. Tools Appl.*, vol. 75, no. 22, pp. 14745–14757, Nov. 2016.

[20] B. Kitchenham, *Procedures for Performing Systematic Reviews*, Keele Univ., NICTA, Keele, UK, 2004.

[21] J. Webster and R. T. Watson, "Analyzing the past to prepare for the future: Writing a literature review," *MIS Quart.*, vol. 26, pp. 1–12, 2002.

[22] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling black-box mutational fuzzing," in *Proc. 2013 ACM SIGSAC Conf. Comput. Commun. Security*, New York, NY, USA, 2013, pp. 511–522.

[23] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, "A taint based approach for smart fuzzing," in *Proc. IEEE 5th Int. Conf. Softw. Test. Verification Validation 2012*, 2012, pp. 818–825.

[24] G. Wen, Y. Zhang, Q. Liu, and D. Yang, "Fuzzing the ActionScript virtual machine," in *Proc. 8th ACM SIGSAC Symp. Inf. Comput. Commun. Security*, New York, NY, USA, 2013, pp. 457–468.

[25] R. Brummayer and A. Biere, "Fuzzing and delta-debugging SMT solvers," in *Proc. 7th Int. Workshop Satisfiability Modulo Theories*, 2009, pp. 1–5.

[26] Y. Chen *et al*., "Taming compiler fuzzers," in *Proc. 34th ACM SIGPLAN Conf. Program. Lang. Design Implementation*, New York, NY, USA, 2013, pp. 197–208.

[27] *American Fuzzy Lop*, AFL. [Online]. Available: http://lcamtuf.coredump.cx/afl/. Accessed on: Jul. 12, 2016.

[28] E. Jääskelä, "Genetic algorithm in code coverage guided fuzz testing," Dept. Comput. Sci. Eng., Univ. Oulu, 2016.

[29] S. Y. Kim, S. D. Cha, and D.-H. Bae, "Automatic and lightweight grammar generation for fuzz testing," *Comput. Security*, vol. 36, pp. 1–11, 2013.

[30] J. de Ruiter and E. Poll, "Protocol state fuzzing of TLS implementations," in *Proc. 24th USENIX Security Symp.*, 2015, pp. 193–206.

[31] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Commun. ACM*, vol. 33, pp. 32–44, 1990.

[32] D. Jones, "Trinity: A linux system call fuzz tester." [Online]. Available: http://codemonkey.org.uk/projects/trinity/. Accessed on: Jul. 12, 2016.

[33] E. Bazzoli, C. Criscione, F. Maggi, and S. Zanero, "XSS PEEKER: Dissecting the XSS exploitation techniques and fuzzing mechanisms of blackbox web application scanners," in *Proc. 31st IFIP Int. Conf. Inf. Security Privacy*, Ghent, Belgium, May 30–Jun. 1, 2016, vol. 471, pp. 243–258.

[34] F. Duchène, S. Rawat, J. L. Richier, and R. Groz, "LigRE: Reverse-engineering of control and data flow models for black-box XSS detection," in *Proc. 20th Working Conf. Reverse Eng.*, 2013, pp. 252–261.

[35] W. Drewry and T. Ormandy, "Flayer: Exposing application internals," in *Proc. 1st USENIX Workshop Offensive Technol.*, Boston, MA, USA, Aug. 6, 2007, pp. 1–9.

[36] 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. Security Privacy*, 2010, pp. 497–512.

[37] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *Proc. IEEE 31st Int. Conf. Softw. Eng.*, 2009, pp. 474–484.

[38] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Proc. Netw. Distrib. Syst. Security Symp.*, San Diego, CA, USA, Feb. 10–13, 2008, pp. 1–16.

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

[40] C.-K. Luk *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, New York, NY, USA, 2005, pp. 190–200.

[41] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proc. 28th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, New York, NY, USA, 2007, pp. 89–100.

[42] R. L. J. Seagle, "A framework for file format fuzzing with genetic algorithms," Ph.D. dissertation, Univ. Tennessee, Knoxville, TN, USA, 2012.

[43] Y.-H. Choi, M.-W. Park, J.-H. Eom, and T.-M. Chung, "Dynamic binary analyzer for scanning vulnerabilities with taint analysis," *Multimed. Tools Appl.*, vol. 74, no. 7, pp. 2301–2320, 2015.

[44] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "DTA++: Dynamic taint analysis with targeted control-flow propagation," in *Proc. Netw. Distrib. Syst. Security Symp.*, San Diego, CA, USA, Feb. 6–9, 2011, pp. 1–14.

[45] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, "Finding software vulnerabilities by smart fuzzing," in *Proc. 4th IEEE Int. Conf. Softw. Test. Verification Validation*, 2011, pp. 427–430.

[46] S. K. Fayaz, T. Yu, Y. Tobioka, S. Chaki, and V. Sekar, "BUZZ: Testing context-dependent policies in stateful networks," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2016, pp. 275–289.

[47] A. Rebert *et al.*, "Optimizing seed selection for fuzzing," in *Proc. 23rd USENIX Security Symp.*, San Diego, CA, USA, 2014, pp. 861–875.

[48] U. Kargén and N. Shahmehri, "Turning programs against each other: High coverage fuzz-testing using binary-code mutation and dynamic slicing," in *Proc. 10th Joint Meeting Found. Softw. Eng.*, New York, NY, USA, 2015, pp. 782–792.

[49] H. Liang, Y. Wang, H. Cao, and J. Wang, "Fuzzing the font parser of compound documents," in *Proc. 4th IEEE Int. Conf. Cyber Security Cloud Comput.*, New York, NY, USA, Jun. 26–28, 2017, pp. 237–242.

[50] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *Proc. IEEE Symp. Security Privacy*, 2017, pp. 579–594.

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

[52] A. Arcuri, M. Z. Z. Iqbal, and L. C. Briand, "Formal analysis of the effectiveness and predictability of random testing," in *Proc. 19th Int. Symp. Softw. Test. Anal.*, Trento, Italy, Jul. 12–16, 2010, pp. 219–230.

[53] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proc. Int. Conf. Softw. Eng.*, 2007, pp. 75–84.

[54] K. Yatoh, K. Sakamoto, F. Ishikawa, and S. Honiden, "Feedback-controlled random test generation," in *Proc. Int. Symp. Softw. Test. Anal. 2015*, Baltimore, MD, USA, Jul. 12–17, 2015, pp. 316–326.

[55] FFmpeg. [Online]. Available: http://samples.ffmpeg.org/. Accessed on: Dec. 15, 2016.

[56] cwebp | WebP, cwebp, Google Developers. [Online]. Available: https://developers.google.com/speed/webp/docs/cwebp. Accessed on: Dec. 15, 2016.

[57] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, "Prospex: Protocol specification extraction," in *Proc. 30th IEEE Symp. Security Privacy*, 2009, pp. 110–125.

[58] T. Y. Chen, H. Leung, and I. K. Mak, "Adaptive random testing," in *Proc. Annu. Asian Comput. Sci. Conf.*, vol. 3321, 2004, pp. 320–329.

[59] A. Arcuri and L. C. Briand, "Adaptive random testing: An illusion of effectiveness?" in *Proc. 20th Int. Symp. Softw. Test. Anal.*, Toronto, ON, Canada, Jul. 17–21, 2011, pp. 265–275.

[60] M. Jurczyk, "Effective file format fuzzing-thoughts techniques and results," in *Proc. Black Hat Eur. Conf.*, London, U.K., 2016, pp. 1–133.

[61] H. C. Kim, Y. H. Choi, and D. H. Lee, "Efficient file fuzz testing using automated analysis of binary file format," *J. Syst. Archit.*, vol. 57, no. 3, pp. 259–268, 2011.

[62] T. Wang, T. Wei, G. Gu, and W. Zou, "Checksum-aware fuzzing combined with dynamic taint analysis and symbolic execution," *ACM Trans. Inf. Syst. Security*, vol. 14, no. 2, pp. 15:1–15:28, 2011.

[63] M. Höschele and A. Zeller, "Mining input grammars from dynamic taints," in *Proc. 31st IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2016, pp. 720–725.

[64] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: Program-state based binary fuzzing," in *Proc. 11th Joint Meeting Found. Softw. Eng.*, New York, NY, USA, 2017, pp. 627–637.

[65] X. Y. Zhu and Z. Y. Wu, "A new fuzzing technique using niche genetic algorithm," *Adv. Mater. Res.*, vol. 756, pp. 4050–4058, 2013.

[66] X. Zhu, Z. Wu, and J. W. Atwood, "A new fuzzing method using multi data samples combination," *J. Comput.*, vol. 6, no. 5, pp. 881–888, May 2011.

[67] K. Dewey, J. Roesch, and B. Hardekopf, "Fuzzing the rust typechecker using CLP (T)," in *Proc. 30th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, Lincoln, NE, USA, Nov. 9–13, 2015, pp. 482–493.

[68] K. Dewey, J. Roesch, and B. Hardekopf, "Language fuzzing using constraint logic programming," in *Proc. 29th ACM/IEEE Int. Conf. Autom. Softw. Eng.*, New York, NY, USA, 2014, pp. 725–730.

[69] C. Cao, N. Gao, P. Liu, and J. Xiang, "Towards analyzing the input validation vulnerabilities associated with android system services," in *Proc. 31st Annu. Comput. Security Appl. Conf.*, New York, NY, USA, 2015, pp. 361–370.

[70] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, "Many-core compiler fuzzing," in *Proc. 36th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, New York, NY, USA, 2015, pp. 65–76.

[71] J. Zhao, Y. Wen, and G. Zhao, "H-Fuzzing: A new heuristic method for fuzzing data generation," in *Network and Parallel Computing*, E. Altman and W. Shi, Eds. Berlin, Germany: Springer, 2011, pp. 32–43.

[72] H. Dai, C. Murphy, and G. E. Kaiser, "CONFU: Configuration fuzzing testing framework for software vulnerability detection," *Int. J. Secure Softw. Eng.*, vol. 1, no. 3, pp. 41–55, 2010.

[73] S. Rasthofer, S. Artz, S. Triller, and M. Pradel, "Making malory behave maliciously: Targeted fuzzing of android execution environments," in *Proc. 39th Int. Conf. Softw. Eng.*, Piscataway, NJ, USA, 2017, pp. 300–311.

[74] P. Tsankov, M. T. Dashti, and D. A. Basin, "Semi-valid input coverage for fuzz testing," in *Proc. Int. Symp. Softw. Test. Anal.*, 2013, pp. 56–66.

[75] O. Bastani, R. Sharma, A. Aiken, and P. Liang, "Synthesizing program input grammars," in *Proc. 38th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, New York, NY, USA, 2017, pp. 95–110.

[76] K. Chen, Y. Zhang, and P. Liu, "Dynamically discovering likely memory layout to perform accurate fuzzing," *IEEE Trans. Rel.*, vol. 65, no. 3, pp. 1180–1194, Sep. 2016.

[77] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr, "Swarm testing," in *Proc. Int. Symp. Softw. Test. Anal.*, Minneapolis, MN, USA, Jul. 15–20, 2012, pp. 78–88.

[78] M. A. Alipour, A. Groce, R. Gopinath, and A. Christi, "Generating focused random tests using directed swarm testing," in *Proc. 25th Int. Symp. Softw. Test. Anal.*, Saarbrücken, Germany, Jul. 18–20, 2016, pp. 70–81.

[79] P. D. Marinescu and C. Cadar, "High-coverage symbolic patch testing," in *Proc. 19th Int. Workshop Model Checking Softw.*, Oxford, U.K., vol. 7385, Jul. 23–24, 2012, pp. 7–21.

[80] A. Podgurski *et al.*, "Automated support for classifying software failure reports," in *Proc. 25th Int. Conf. Softw. Eng.*, Washington, DC, USA, 2003, pp. 465–475.

[81] P. Francis, D. Leon, M. Minch, and A. Podgurski, "Tree-based methods for classifying software failures," in *Proc. 15th Int. Symp. Softw. Rel. Eng.*, Saint-Malo, Bretagne, France, Nov. 2–5, 2004, pp. 451–462.

[82] *Research Insights Volume 9—Modern Security Vulnerability Discovery*, NCC Group, 2016. [Online]. Available: https://www.nccgroup.trust/uk/our-research/research-insights-vol-9-modern-security-vulnerability-discovery/. Accessed on: Nov. 15, 2016.

[83] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Proc. USENIX Annu. Tech. Conf.*, Boston, MA, USA, Jun. 13–15, 2012, pp. 309–318.

[84] V. T. Pham, W. B. Ng, K. Rubinov, and A. Roychoudhury, "Hercules: Reproducing crashes in real-world application binaries," in *Proc. 37th IEEE Int. Conf. Softw. Eng.*, vol. 1, 2015, pp. 891–901.

[85] A. Lanzi, L. Martignoni, M. Monga, and R. Paleari, "A smart fuzzer for ×86 executables," in *Proc. 3rd Int. Workshop Softw. Eng. Secure Syst.*, 2007, pp. 1–8.

[86] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," in *Proc. 22th USENIX Security Symp.*, Washington, DC, USA, Aug. 14–16, 2013, pp. 49–64.

[87] Y. Shoshitaishvili *et al.*, "SOK: (State of) The art of war: Offensive techniques in binary analysis," in *Proc. IEEE Symp. Security Privacy*, 2016, pp. 138–157.

[88] P. Godefroid, "Compositional dynamic test generation," in *Proc. 34th Annu. ACM SIGPLAN-SIGACT Symp. Principles Program. Lang.*, New York, NY, USA, 2007, pp. 47–54.

[89] P. Godefroid, "Higher-order test generation," in *Proc. 32nd ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, New York, NY, USA, 2011, pp. 258–269.

[90] P. Boonstoppel, C. Cadar, and D. R. Engler, "RWset: Attacking path explosion in constraint-based test generation," in *Proc. 14th Int. Conf. Tools Algorithms Construction Anal. Syst.*, Budapest, Hungary, vol. 4963, Mar. 29–Apr. 6, 2008, pp. 351–366.

[91] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," in *Proc. 33rd ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, New York, NY, USA, 2012, pp. 193–204.

[92] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. 8th USENIX Symp. Operating Syst. Des. Implementation*, San Diego, CA, USA, Dec. 8–10, 2008, pp. 209–224.

[93] P. Godefroid and D. Luchaup, "Automatic partial loop summarization in dynamic test generation," in *Proc. Int. Symp. Softw. Test. Anal.*, Toronto, ON, Canada, 2011, pp. 23–33.

[94] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *Proc. 23rd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, L'Aquila, Italy, Sep. 15–19, 2008, pp. 443–446.

[95] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as Markov chain," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, New York, NY, USA, 2016, pp. 1032–1043.

[96] K. Böttinger and C. Eckert, "DeepFuzz: Triggering vulnerabilities deeply hidden in binaries," in *Proc. 13th Int. Conf. Detection Intrusions Malware Vulnerability Assessment*, San Sebastián, Spain, Jul. 7–8, 2016, pp. 25–34.

[97] 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.*, San Jose, CA, USA, Jul. 21–26, 2014, pp. 160–170.

[98] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *Proc. 10th Eur. Softw. Eng. Conf. 13th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, New York, NY, USA, 2005, pp. 263–272.

[99] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," in *Proc. 16th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, Newport Beach, CA, USA, Mar. 5–11, 2011, pp. 265–278.

[100] M. Mouzarani, B. Sadeghiyan, and M. Zolfaghari, "A smart fuzzing method for detecting heap-based vulnerabilities in executable codes," *Security Commun. Netw.*, vol. 9, no. 18, pp. 5098–5115, 2016.

[101] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proc. 29th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, New York, NY, USA, 2008, pp. 206–215.

[102] P. Godefroid and J. Kinder, "Proving memory safety of floating-point computations by combining static and dynamic program analysis," in *Proc. 19th Int. Symp. Softw. Test. Anal.*, Trento, Italy, Jul. 12–16, 2010, pp. 1–12.

[103] Z. Fu and Z. Su, "Achieving high coverage for floating-point code via unconstrained programming," in *Proc. 38th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, New York, NY, USA, 2017, pp. 306–319.

[104] B. Yadegari and S. Debray, "Bit-level taint analysis," in *Proc. 14th IEEE Int. Working Conf. Source Code Anal. Manipulation*, Victoria, BC, Canada, Sep. 28–29, 2014, pp. 255–264.

[105] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proc. 31st IEEE Symp. Security Privacy*, Berleley/Oakland, CA, USA, May 16–19, 2010, pp. 317–331.

[106] 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.

[107] E. Bounimova, P. Godefroid, and D. A. Molnar, "Billions and billions of constraints: Whitebox fuzz testing in production," in *Proc. 35th Int. Conf. Softw. Eng.*, San Francisco, CA, USA, May 18–26, 2013, pp. 122–131.

[108] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware evolutionary fuzzing," in *Proc. 24th Annu. Netw. Distrib. Syst. Security Symp.*, San Diego, CA, USA, Feb. 26–Mar. 1, 2017.

[109] S. Person, G. Yang, N. Rungta, and S. Khurshid, "Directed incremental symbolic execution," in *Proc. 32nd ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, San Jose, CA, USA, Jun. 4–8, 2011, pp. 504–515.

[110] C.-J. M. Liang *et al.*, "Caiipa: Automated large-scale mobile app testing through contextual fuzzing," in *Proc. 20th Annu. Int. Conf. Mobile Comput. Netw.*, 2014, pp. 519–530.

[111] L. W. Hao, M. S. Ramanujam, and S. P. T. Krishnan, "On designing an efficient distributed black-box fuzzing system for mobile devices," in *Proc. 10th ACM Symp. Inf. Comput. Commun. Security*, 2015, pp. 31–42.

[112] *Peach Fuzzer: Discover Unknown Vulnerabilities*, Peach, Peach Fuzzer. [Online]. Available: http://www.peachfuzzer.com/. Accessed on: Jul. 13, 2016.

[113] H. Dantas, Z. Erkin, C. Doerr, R. Hallie, and G. van der Bij, "eFuzz: A fuzzer for DLMS/COSEM electricity meters," in *Proc. 2nd Workshop Smart Energy Grid Security*, Scottsdale, AZ, USA, 2014, pp. 31–38.

[114] *Honggfuzz by Google*, Honggfuzz. [Online]. Available: https://google.github.io/honggfuzz/. Accessed on: Jul. 13, 2016.

[115] *Dynamic Testing (Fuzzing) on the ISASecure EDSA Certification 402 Ethernet by beSTORM*, beSTORM. [Online]. Available: http://www.beyondsecurity.com/dynamic_fuzzing_testing_embedded_device_security_assurance_402_ethernet. Accessed on: Jul. 19, 2016.

[116] *MozillaSecurity/funfuzz*, jsfunfuzz, GitHub. [Online]. Available: https://github.com/MozillaSecurity/funfuzz. Accessed on: Dec. 16, 2016.

[117] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proc. 32nd ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, New York, NY, USA, 2011, pp. 283–294.

[118] W. M. McKeeman, "Differential testing for software," *Digit. Tech. J.*, vol. 10, no. 1, pp. 100–107, 1998.

[119] Csmith. [Online]. Available: https://embed.cs.utah.edu/csmith/. Accessed on: Dec. 16, 2016.

[120] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Proc. 21th USENIX Security Symp.*, Bellevue, WA, USA, Aug. 8–10, 2012, pp. 445–458.

[121] R. Guo, "MongoDB's JavaScript fuzzer," *Commun. ACM*, vol. 60, no. 5, pp. 43–47, 2017.

[122] S. Veggalam, S. Rawat, I. Haller, and H. Bos, "IFuzzer: An evolutionary interpreter fuzzer using genetic programming," in *Proc. Eur. Symp. Res. Comput. Security*, vol. 9878, 2016, pp. 581–601.

[123] *Project Springfield*, Springfield. [Online]. Available: https://www.microsoft.com/en-us/springfield/. Accessed on: Apr. 15, 2017.

[124] N. Stephens *et al.*, "Driller: Augmenting fuzzing through selective symbolic execution," in *Proc. 23nd Annu. Netw. Distrib. Syst. Security Symp.*, San Diego, CA, USA, Feb. 21–24, 2016, pp. 1–16.

[125] *Project Triforce: Run AFL on Everything!*, TriforceAFL. [Online]. Available: https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/june/project-triforce-run-afl-on-everything/. Accessed on: Jul. 13, 2016.

[126] ivanfratric/winafl, WinAFL, GitHub. [Online]. Available: https://github.com/ivanfratric/winafl. Accessed on: Dec. 16, 2016.

[127] *AFL Filesystem Fuzzing, Vault 2016_0.pdf*, Oracle Linux and VM Development, 2016. [Online]. Available: http://events.linuxfoundation.org/sites/events/files/slides/AFL%20filesystem%20fuzzing%2C%20Vault%202016_0.pdf. Accessed on: Jul. 13, 2016.

[128] G. Grieco, M. Ceresa, and P. Buiras, "QuickFuzz: An automatic random fuzzer for common file formats," in *Proc. Int. Symp. Haskell*, 2016, pp. 13–20.

[129] J. Davis, A. Thekumparampil, and D. Lee, "Node.Fz: Fuzzing the server-side event-driven architecture," in *Proc. 12th Eur. Conf. Comput. Syst.*, New York, NY, USA, 2017, pp. 145–160.

[130] M. Marhefka and P. Müller, "Dfuzzer: A D-bus service fuzzing tool," in *Proc. IEEE 7th Int. Conf. Softw. Test. Verification Validation Workshops*, 2014, pp. 383–389.

[131] A. Joseph, "Droid-FF: The first android fuzzing framework," in *Proc. Hack Box Security Conf.*, Amsterdam, The Netherlands, 2016. [Online]. Available: http://conference.hitb.org/hitbsecconf2016ams/sessions/hitb-lab-droid-ff-the-first-android-fuzzing-framework/

[132] H. Shahriar, S. North, and E. Mawangi, "Testing of memory leak in android applications," in *Proc. IEEE 15th Int. Symp. High-Assurance Syst. Eng.*, 2014, pp. 176–183.

[133] H. Ye, S. Cheng, L. Zhang, and F. Jiang, "DroidFuzzer: Fuzzing the android apps with intent-filter tag," in *Proc. Int. Conf. Adv. Mobile Comput. Multimed.*, 2013, pp. 1–7.

[134] R. Sasnauskas and J. Regehr, "Intent fuzzer: Crafting intents of death," in *Proc. Joint Int. Workshop Dyn. Anal. Softw. Syst. Perform. Test. Debugging Anal.*, 2014, pp. 1–5.

[135] D. Amalfitano, N. Amatucci, A. R. Fasolino, P. Tramontana, E. Kowalczyk, and A. M. Memon, "Exploiting the saturation effect in automatic random testing of android applications," in *Proc. 2nd ACM Int. Conf. Mobile Softw. Eng. Syst.*, 2015, pp. 33–43.

[136] *OpenRCE/sulley*, Sulley, GitHub. [Online]. Available: https://github.com/OpenRCE/sulley. Accessed on: Jul. 12, 2016.

[137] jtpereyda/boofuzz, Boofuzz, GitHub. [Online]. Available: https://github.com/jtpereyda/boofuzz. Accessed on: Jul. 23, 2016.

[138] J. Somorovsky, "Systematic fuzzing and testing of TLS libraries," in *Proc. 2016 ACM SIGSAC Conf. Comput. Commun. Security*, New York, NY, USA, 2016, pp. 1492–1504.

[139] D. Aitel, "The advantages of block-based protocol analysis for security testing," *Immunity Inc.*, vol. 105, pp. 349–352, 2002.

[140] T. Rontti, A. M. Juuso, and A. Takanen, "Preventing DoS attacks in NGN networks with proactive specification-based fuzzing," *IEEE Commun. Mag.*, vol. 50, no. 9, pp. 164–170, Sep. 2012.

[141] W. Johansson, M. Svensson, U. E. Larson, M. Almgren, and V. Gulisano, "T-Fuzz: Model-based fuzzing for robustness testing of telecommunication protocols," in *Proc. IEEE 7th Int. Conf. Softw. Test. Verification Validation*, 2014, pp. 323–332.

[142] P. Tsankov, M. T. Dashti, and D. Basin, "SecFuzz: Fuzz-testing security protocols," in *Proc. 7th Int. Workshop Autom. Softw. Test*, Zurich, Switzerland, 2012, pp. 1–7.

[143] G. Banks, M. Cova, V. Felmetsger, K. C. Almeroth, R. A. Kemmerer, and G. Vigna, "SNOOZE: Toward a Stateful NetwOrk prOtocol fuzZEr," in *Proc. Int. Conf. Inf. Security*, vol. 4176, 2006, pp. 343–358.

[144] H. J. Abdelnur, R. State, and O. Festor, "KiF: A stateful SIP fuzzer," in *Proc. 1st Int. Conf. Principles Syst. Appl. IP Telecommun.*, 2007, pp. 47–56.

[145] H. J. Abdelnur, R. State, and O. Festor, "Advanced fuzzing in the VoIP space," *J. Comput. Virol.*, vol. 6, no. 1, pp. 57–64, 2010.

[146] 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.*, 2013, pp. 1–12.

[147] D. Vyukov, *Trinity: A Linux System Call Fuzzer*, 2016. [Online]. Available: http://codemonkey.org.uk/projects/trinity/. Accessed on: Jul. 12, 2016.

[148] *GitHub—Cr4sh/ioctlfuzzer: Automatically Exported From code.google.com/p/ioctlfuzzer*, IOCTL. [Online]. Available: https://github.com/Cr4sh/ioctlfuzzer. Accessed on: Jul. 13, 2016.

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

[150] S. Y. Kim *et al.*, "CAB-Fuzz: Practical concolic testing techniques for COTS operating systems," in *Proc. USENIX Annu. Tech. Conf.*, Santa Clara, CA, USA, 2017, pp. 689–701.

[151] *vUSBf–QEMU/KEMU USB-Fuzzing Framework*, hucktech, Firmware Security, Feb. 8, 2016.

[152] R. van Tonder and H. A. Engelbrecht, "Lowering the USB fuzzing barrier by transparent two-way emulation," in *Proc. USENIX Workshop Offensive Technol.*, 2014, pp. 1–8.

[153] A. Henderson, H. Yin, G. Jin, H. Han, and H. Deng, "VDF: Targeted evolutionary fuzz testing of virtual devices," in *Proc. 20th Int. Symp. Res. Attacks Intrusions Defenses*, Atlanta, GA, USA, 2017, pp. 3–25.

[154] perf_fuzzer perf_event syscall fuzzer, perf_fuzzer. [Online]. Available: http://web.eece.maine.edu/∼vweaver/projects/perf_events/fuzzer/. Accessed on: Jul. 13, 2016.

[155] libFuzzer—*A Library for Coverage-Guided Fuzz Testing. LLVM 3.9 Documentation*, libFuzzer. [Online]. Available: http://www.llvm.org/docs/LibFuzzer.html. Accessed on: Jul. 13, 2016.

[156] A. G. Voyiatzis, K. Katsigiannis, and S. Koubias, "A modbus/TCP fuzzer for testing internetworked industrial systems," in *Proc. IEEE 20th Conf. Emerg. Technol. Factory Autom.*, 2015, pp. 1–6.

[157] F. L. Sang, V. Nicomette, and Y. Deswarte, "A tool to analyze potential I/O attacks against PCs," *IEEE Security Privacy*, vol. 12, no. 2, pp. 60–66, Mar. 2014.

[158] B. Chandrasekar, B. Ramesh, V. Prabhu, S. Sajeev, P. K. Mohanty, and G. Shobha, "Development of intelligent digital certificate fuzzer tool," in *Proc. Int. Conf. Cryptogr. Security Privacy*, Wuhan, China, 2017, pp. 126–130.

[159] A. Case, A. K. Das, S.-J. Park, J. R. Ramanujam, and G. G. Richard III, "Gaslight: A comprehensive fuzzing architecture for memory forensics frameworks," *Digit. Investigation*, vol. 22, pp. S86–S93, 2017.

[160] E. Stepanov and K. Serebryany, "MemorySanitizer: Fast detector of uninitialized memory use in C++," in *Proc. 13th Annu. IEEE/ACM Int. Symp. Code Gener. Optim.*, San Francisco, CA, USA, Feb. 7–11, 2015, pp. 46–55.

[161] E. Poll, J. D. Ruiter, and A. Schubert, "Protocol state machines and session languages: Specification, implementation, and security flaws," in *Proc. IEEE Security Privacy Workshops*, 2015, pp. 125–133.

[162] S. Rawat and L. Mounier, "An evolutionary computing approach for hunting buffer overflow vulnerabilities: A case of aiming in dim light," in *Proc. Eur. Conf. Comput. Netw. Defense*, 2010, pp. 37–45.

[163] R. B. Dannenberg *et al.*, "As-If infinitely ranged integer model," in *Proc. IEEE 21st Int. Symp. Softw. Rel. Eng.*, 2010, pp. 91–100.

[164] T. Wang, T. Wei, Z. Lin, and W. Zou, "IntScope: Automatically detecting integer overflow vulnerability in X86 binary using symbolic execution," in *Proc. Netw. Distrib. Syst. Security Symp.*, 2009, pp. 1–14.

[165] S. Palka and D. McCoy, "Fuzzing E-mail filters with generative grammars and n-gram analysis," in *Proc. Workshop Offensive Technol.*, 2015, pp. 1–10.

[166] M. Mouzarani, B. Sadeghiyan, and M. Zolfaghari, "A smart fuzzing method for detecting heap-based buffer overflow in executable codes," in *Proc. IEEE 21st Pacific Rim Int. Symp. Dependable Comput.*, 2015, pp. 42–49.

[167] C. C. Yeh, H. Chung, and S. K. Huang, "CRAXfuzz: Target-aware symbolic fuzz testing," in *Proc. IEEE 39th Annu. Comput. Softw. Appl. Conf.*, vol. 2, 2015, pp. 460–471.

[168] S. K. Huang, M. H. Huang, P. Y. Huang, H. L. Lu, and C. W. Lai, "Software crash analysis for automatic exploit generation on binary programs," *IEEE Trans. Rel.*, vol. 63, no. 1, pp. 270–289, Mar. 2014.

[169] B. Dolan-Gavitt *et al.*, "LAVA: Large-scale automated vulnerability addition," in *Proc. IEEE Symp. Security Privacy*, 2016, pp. 110–121.

[170] D. Zhang *et al.*, "SimFuzz: Test case similarity directed deep fuzzing," *J. Syst. Softw.*, vol. 85, no. 1, pp. 102–111, 2012.

[171] W. M. McKeeman, "Differential testing for software," *Digit. Tech. J.*, vol. 10, no. 1, pp. 100–107, 1998.

[172] S. Kyle, H. Leather, B. Franke, D. Butcher, and S. Monteith, "Application of domain-aware binary fuzzing to aid android virtual machine testing," in *Proc. 11th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ.*, New York, NY, USA, 2015, pp. 121–132.

[173] T. Y. Chen, F.-C. Kuo, H. Liu, and W. E. Wong, "Code coverage of adaptive random testing," *IEEE Trans. Rel.*, vol. 62, no. 1, pp. 226–237, Mar. 2013.

[174] T. Y. Chen, F.-C. Kuo, and H. Liu, "Application of a failure driven test profile in random testing," *IEEE Trans. Rel.*, vol. 58, no. 1, pp. 179–192, Mar. 2009.

[175] A. F. Tappenden and J. Miller, "A novel evolutionary approach for adaptive random testing," *IEEE Trans. Rel.*, vol. 58, no. 4, pp. 619–633, Dec. 2009.

[176] E. Rogstad and L. C. Briand, “Clustering deviations for black box regression testing of database applications,” *IEEE Trans. Rel.*, vol. 65, no. 1, pp. 4–18, Mar. 2016.

[177] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, “Coverage-directed differential testing of JVM implementations,” in *Proc. 37th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, New York, NY, USA, 2016, pp. 85–99.

[178] B. Shastry *et al.*, “Static program analysis as a fuzzing aid,” in *Proc. 20th Int. Symp. Res. Attacks Intrusions Defenses*, Atlanta, GA, USA, 2017, pp. 26–47.

[179] V.-T. Pham, M. Böhme, and A. Roychoudhury, “Model-based whitebox fuzzing for program binaries,” in *Proc. 31st IEEE/ACM Int. Conf. Autom. Softw. Eng.*, New York, NY, USA, 2016, pp. 543–553.

[180] T. Y. Chen *et al.*, “Metamorphic testing for cybersecurity,” *Computer*, vol. 49, no. 6, pp. 48–55, Jun. 2016.

[181] T. Y. Chen, T. H. Tse, and Z. Zhou, “Fault-based testing without the need of oracles,” *Inf. Softw. Technol.*, vol. 45, no. 1, pp. 1–9, 2003.

[182] H. Liu, F.-C. Kuo, D. Towey, and T. Y. Chen, “How effectively does metamorphic testing alleviate the oracle problem?” *IEEE Trans. Softw. Eng.*, vol. 40, no. 1, pp. 4–22, Jan. 2014.

[183] T. Y. Chen *et al.*, “Metamorphic testing: A review of challenges and opportunities,” *ACM Comput. Surv.*, vol. 51, no. 1, pp. 4:1–4:27, Jan. 2018.

[184] B. Garn and D. E. Simos, “Eris: A tool for combinatorial testing of the linux system call interface,” in *Proc. IEEE 7th Int. Conf. Softw. Test. Verification Validation Workshops*, 2014, pp. 58–67.

[185] B. Shastry, F. Maggi, F. Yamaguchi, K. Rieck, and J.-P. Seifert, “Static exploration of taint-style vulnerabilities found by fuzzing,” in *Proc. 11th USENIX Workshop Offensive Technol.*, Vancouver, BC, Canada, 2017.

[186] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner, and R. Ramler, “GRT: Program-analysis-guided random testing (T),” in *Proc. 30th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, Lincoln, NE, USA, vol. 2015, Nov. 9–13, 2015, pp. 212–223.

[187] E. Alatawi, T. Miller, and H. Søndergaard, “Using metamorphic testing to improve dynamic symbolic execution,” in *Proc. 24th Australas. Softw. Eng. Conf.*, 2015, pp. 38–47.

[188] N. Havrikov, “Efficient fuzz testing leveraging input, code, and execution,” in *Proc. 39th Int. Conf. Softw. Eng.*, Buenos Aires, Argentina, vol. 2017, May 20–28, 2017, pp. 417–420.

[189] P. Godefroid, H. Peleg, and R. Singh, “Learn&Fuzz: Machine learning for input fuzzing,” in *Proc. 32nd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, Urbana, IL, USA, Oct. 30–Nov. 3, 2017, pp. 50–59.

[190] K. Böttinger, “Fuzzing binaries with Lévy flight swarms,” *EURASIP J. Inf. Security*, vol. 2016, no. 1, Nov. 2016, Art. no. 28.

[191] K. Böttinger, “Hunting bugs with levy flight foraging,” in *Proc. IEEE Security Privacy Workshops*, 2016, pp. 111–117.

[192] F. Duchene, S. Rawat, J.-L. Richier, and R. Groz, “KameleonFuzz: Evolutionary fuzzing for black-box XSS detection,” in *Proc. ACM Conf. Data Appl. Security Privacy*, 2014, pp. 37–48.

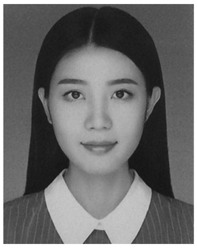
[193] F. Duchene, R. Groz, S. Rawat, and J. L. Richier, “XSS vulnerability detection using model inference assisted evolutionary fuzzing,” in *Proc. IEEE 5th Int. Conf. Softw. Test. Verification Validation*, 2012, pp. 815–817.

[194] S. Dai, A. Tongaonkar, X. Wang, A. Nucci, and D. Song, “NetworkProfiler: Towards automatic fingerprinting of Android apps,” in *Proc. IEEE Int. Conf. Comput. Commun.*, 2013, pp. 809–817.

Hongliang Liang (M’14) received the Ph.D. degree in computer science from the University of Chinese Academy of Sciences, Beijing, China, in 2002.

He is currently an Associate Professor with Beijing University of Posts and Telecommunications, Beijing, China. His research interests include system software, program analysis, software security, and artificial intelligence.

Dr. Liang is a senior member of China Computer Federation.



Xiaoxiao Pei received the M.Sc. degree in computer science from the Beijing University of Posts and Telecommunications, Beijing, China, in 2018.

Her research interests include fuzzing and symbolic execution.

Xiaodong Jia received the M.Sc. degree in computer science from the Beijing University of Posts and Telecommunications, Beijing, China, in 2018.

Her research interests include taint analysis and Android security.

Wuwei Shen (M’00) received the Ph.D. degree in computer science from the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI, USA, in 2001.

He is currently an Associate Professor with Western Michigan University, Kalamazoo, MI, USA. His research interests include object-oriented analysis and design modeling, model consistency checking, model-based software testing, assurance-based software development, and software certification.

Dr. Shen received the research faculty fellowship from the Air Force Research Lab, in 2015 and 2016. He was the recipient of a senior research award from the U.S. National Research Council Research Associateship Programs to work on the assurance-based software development for mission critical systems in the AFRL, Rome, NY, USA, in 2017 and 2018.

Jian Zhang (SM’09) is a Research Professor with the Institute of Software, Chinese Academy of Sciences, Beijing, China. His main research interests include automated reasoning, constraint satisfaction, program analysis, and software testing.

He has served on the program committees of some 70 international conferences. He also serves on the editorial boards of several journals, including the IEEE TRANSACTIONS ON RELIABILITY, *Frontiers of Computer Science*, and the *Journal of Computer Science and Technology*. He is a senior member of ACM and is a distinguished member of the China Computer Federation.