Soft Computing (2024) 28:5493–5522
https://doi.org/10.1007/s00500-023-09306-2

APPLICATION OF SOFT COMPUTING

A systematic review of fuzzing

Xiaoqi Zhao[1] · Haipeng Qu[2] · Jianliang Xu[2] · Xiaohui Li[2] · Wenjie Lv[2] · Gai-Ge Wang[2]

Accepted: 25 September 2023 / Published online: 31 October 2023

Abstract
Fuzzing is an important technique in software and security testing that involves continuously generating a large number of test cases against target programs to discover unexpected behaviors such as bugs, crashes, and vulnerabilities. Recently, fuzzing has advanced considerably owing to the emergence of new methods and corresponding tools. However, it still suffers from low coverage, ineffective detection of specific vulnerabilities, and difficulty in deploying complex applications. Therefore, to comprehensively survey the development of fuzzing techniques and analyze their advantages and existing challenges, this paper provides a comprehensive survey of the development of fuzzing techniques, summarizes the main research issues, and provides a categorized overview of the latest research advances and applications. The paper first introduces the background and related work on fuzzing. Research issues are subsequently addressed and summarized, along with the latest research developments. Furthermore, various customized fuzzing techniques in different applications are presented. Finally, the paper discusses future research directions.

Keywords Fuzzing · Software testing · Security · Survey · Vulnerability

1 Introduction

Fuzzing, also known as fuzz testing, is a powerful software testing technique that has gained significant attention in the field of software and system security testing. It involves automatically generating a large number of test cases and feeding them into the target program to detect bugs, crashes, or vulnerabilities. Today, fuzzing has emerged as a popular technique in both academia and industry. Some prominent software companies, such as Google (Abhishek and Cris 2012; Chris et al. 2011; Max and Kostya 2016), Microsoft (Onefuzz 2020), Cisco and Adobe (Brad 2009), have developed their fuzzing tools and have successfully discovered thousands of vulnerabilities in their products. An increasing number of fuzzing studies appear at security and software engineering-related conferences and journals (Godefroid et al. 2008a; Woo et al. 2013). Designed fuzzing tools (also known as fuzzers) open sourced on GitHub and discovered many vulnerabilities in open-source software. Additionally, fuzzing has been widely employed in various renowned competitions, including the DARPA Cyber Grand Challenge (2016).

Fuzzing was proposed by Miller et al. in 1988. It was primarily employed for testing the robustness of UNIX programs (Miller et al. 1995). In 1999, it was extended to encompass security testing. During this period, blackbox fuzzing was predominantly implemented, with notable fuzzers such as PROTOS (Viide et al. 2008), SPIKE (Godefroid 2020), and Peach (Liang et al. 2018a). Blackbox fuzzing generates test cases randomly, with fast testing speed. However, it lacks access to internal program information, limiting the full exploration of deep program logic. In 2008, Godefroid et al. (2008c) developed SAGE, a whitebox fuzzer that combines symbolic execution and fuzzing techniques to generate test cases. Compared to blackbox fuzzing, whitebox fuzzing can generate test cases correlating to particular paths by exploiting program internal information. Nonetheless, software complexity and solver limitations (Avgerinos et al. 2014; Baldoni et al. 2018) present obstacles to the effectiveness of fuzzing in conducting thorough testing within a restricted time frame. Therefore, researchers have shown considerable interest in achieving a balance between the utilization of program internal information and testing efficiency. This has driven the development of greybox fuzzing. At the end of 2013, Zalewski (2013) released a greybox

✉ Haipeng Qu
quhaipeng@ouc.edu.cn

1 School of Information and Control Engineering, Qingdao University of Technology, Qingdao, China

2 College of Computer Science and Technology, Ocean University of China, Qingdao, China

fuzzer American Fuzzy Lop (AFL). AFL uses instrumentation to collect path information from the target program and uses coverage to guide test case generation during fuzzing process, which has become known as coverage-based greybox fuzzing (CGF) (HonggFuzz (2015); Serebryany 2016).

Despite the successes achieved by AFL and CGF in the field of fuzzing, there are still numerous unresolved challenges. One of the primary challenges is the limited comprehension of target programs, especially for complex programs. Fully comprehending the logic and data flow of programs is an arduous task. Consequently, this lack of comprehension impedes the exploration of in-depth paths within the program, thereby restricting the improvement of code coverage (Lou and Song 2020). Another significant challenge arises from the restrictions of fuzzing in modeling specific vulnerabilities. Fuzzing randomly generates test case, bug it frequently lacks the crucial information concerning particular vulnerability features and their locations. As a result, it struggles to accurately simulate and detect certain types of vulnerabilities (Trickel et al 2023). Additionally, the deployment of fuzzing in complex applications, and their testing efficiency are important current challenges (Beaman et al. 2022; Donaldson et al. 2023).

In recent years, fuzzing has shown a trend toward integration, diversification, and open source (Google: ClusterFuzz 2019; Serebryany 2017). Current research on fuzzing mainly focuses on general fuzzing, vulnerability-oriented fuzzing, combining fuzzing with other techniques, and fuzzing for different applications. General fuzzing aims to improve the process of fuzzing to explore deep program paths and improve code coverage. For example, Skyfire (Wang et al. 2017) improves initial test case generation to increase code coverage, AFLFast (BÖhme et al. 2019) improves energy allocation to discover more paths, FairFuzz enhances mutation strategies to improve path coverage, MooFuzz (Zhao et al. 2021) improves seed schedule for better path discovery, and AFLSmart (Pham et al. 2019) focuses on the input format of the target program to generate test cases that conform to the program's format to explore deep path. General fuzzing has better generality, but they face certain challenges in detecting specific vulnerabilities. To address this challenge, vulnerability-oriented fuzzing focuses on particular vulnerabilities and conducts relevant fuzzing research based on those vulnerability features. For example, MemLock (Wen et al. 2020) focuses on detecting uncontrollable memory consumption and uncontrollable recursive bugs. PerfFuzz (Lemieux et al. 2018) explores algorithmic complexity vulnerabilities by maximizing the edge count in the control flow graph. ConFuzz (Vinesh et al. 2020) considers the characteristics of concurrency vulnerabilities and focuses on detecting this type of vulnerability. Moreover, fuzzing is combined with other security testing techniques such as taint analysis (Bekrar et al. 2012), symbolic execution (Noller et al. 2018), machine learning (Saavedra et al. 2019), and other techniques to improve its testing performance. Fuzzing is also currently being customized for complex applications to uncover potential vulnerabilities and bugs within them.

This overview is motivated by two main points. Firstly, fuzzing has gained significant attention and undergone rapid development in recent years. It has been widely adopted across various applications and extensively utilized by numerous companies and competitions. This highlights the importance and effectiveness of fuzzing in identifying vulnerabilities and enhancing software security. Secondly, there is a lack of comprehensive surveys specifically focused on fuzzing that cover recent advancements and developments. Previous reviews (Li et al. 2018; Liang et al. 2018b; Manès et al. 2019) have provided summaries of fuzzing achievements up until 2018. Other papers (Eisele et al. 2022; Wang et al. 2020) offer systematic reviews of the historical development of fuzzing but tend to concentrate on specific types of fuzzing techniques. There is a necessity for an up-to-date and comprehensive review that encompasses the recent advancements and developments in fuzzing techniques.

This paper presents a comprehensive review of current research on fuzzing. Firstly, an overview of the basic process and classification of fuzzing is provided to offer readers a holistic understanding. The paper then proceeds to introduce CGF as a widely used and representative technique in fuzzing, establishing a solid theoretical foundation and providing technical support for subsequent research advancements. Subsequently, the latest advancements in fuzzing are categorized and discussed, exploring their applications across various domains. Finally, the paper concludes by summarizing the key findings of the reviewed research and future directions.

In this paper, we make the following main contributions.

- We provide an overview of the processes and classifications of fuzzing, give definitions of CGF and related design details.
- We discuss the research issues studied in fuzzing and categorize and survey the latest research work.
- We survey fuzzing techniques in different application scenarios.
- We summarize the challenges and future research directions of fuzzing.

The rest of the paper is organized as shown in Fig. 1. Background and related work are introduced in Sect. 2. Section 3 surveys recent fuzzing research advancements. This is followed by a review of fuzzing in applications in Sect. 4. Section 5 concludes the paper and discusses future directions.

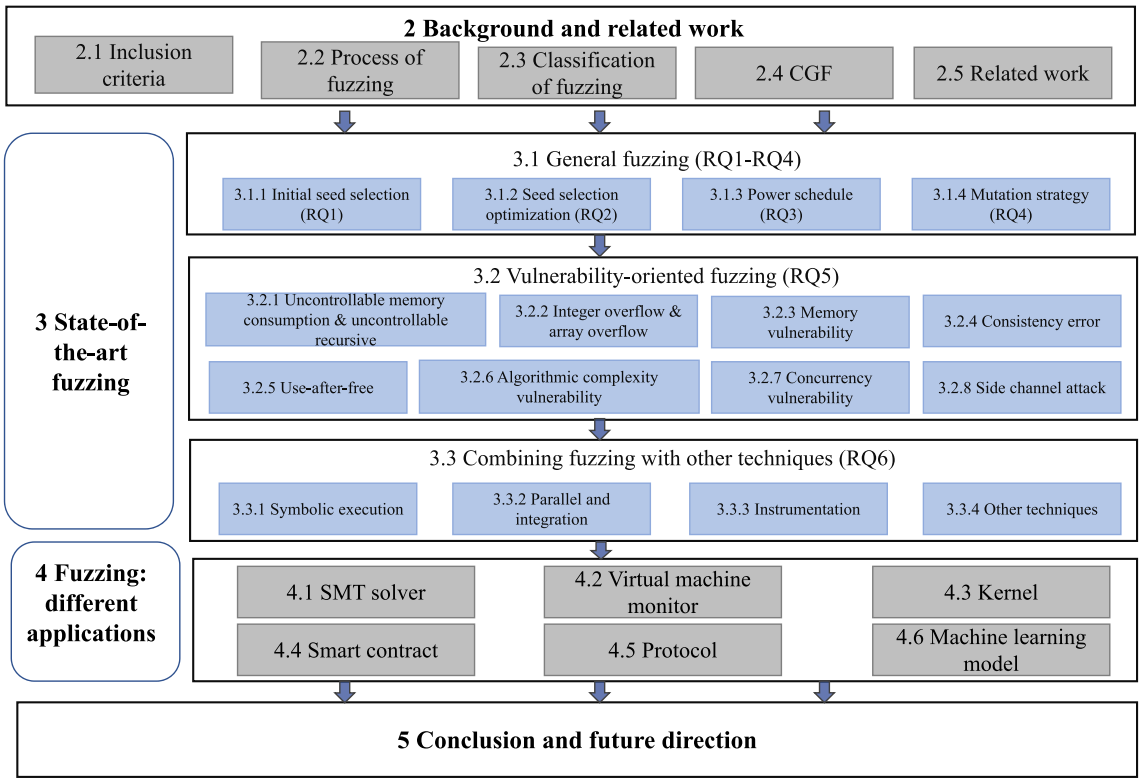


Fig. 1 Paper structure diagram

2 Background and related work

In this section, we first provide the inclusion criteria for the papers covered in this review, then provide an overview of the fuzz testing process, discuss the classification of fuzzing, and introduce the current classic coverage-based greybox fuzzing (CGF), and finally discuss related work.

2.1 Inclusion criteria

We reviewed more than 100 papers, mostly significant works published in top conferences and journals in the software engineering and security field from 2018 to 2023. We also included outstanding fuzzing papers published in industrial conferences, such as Blackhat. To ensure a comprehensive comprehension of the development of fuzzing techniques across various applications, we have gathered a number of classical fuzz testing papers, without any limitations on publication dates. In addition, we have collected top journals papers covering various fuzzing applications to offer a holistic perspective. To clearly define the scope, the inclusion criteria adopted are as follows.

(1) We primarily selected recent proceedings from security and software engineering top conferences for the period from 2017 to 2023. The relevant papers are shown in Table 1. The former includes IEEE Symposium on Security and Privacy (S&P), Usenix Security Symposium (Usenix Security), Network and Distributed System Security (NDSS), and ACM Conference on Computer and Communications Security (CCS). The latter includes International Conference on Software Engineering (ICSE), ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), and IEEE/ACM International Conference on Automated Software Engineering (ASE).

(2) In addition to the top conference papers related to security and software engineering, the review collects other conferences from industry and academia. The major conferences in industry are mainly Black Hat Europe. Other conferences include not only those related to software engineering and security, but also those related to areas such as machine learning and programming language design. We collect the papers related to fuzzing from that conference to summarize fuzzing applications in different fields. The relevant papers are shown in Table 2.

(3) We also select security and software engineering-related journal papers mainly including Computer & Security, Cybersecurity, Empirical Software Engineering, IEEE Transactions on Software Engineering, and International

Table 1 Security and software engineering top conference papers

Conference	Paper	Number	Year
IEEE Symposium on Security and Privacy (S&P)	Wang et al. (2017), Gan et al. (2018), She et al. (2022), You et al. (2019), Peng et al. (2018), Chen and Chen (2018), Liang et al. (2022), Jeong et al. (2019), Xu et al. (2020), Chen et al. (2020b), Huang et al. (2020), Nagy and Hicks (2019)	12	2018–2022
Usenix Security Symposium (Usenix Security)	Yue et al. (2020), Blazytko et al. (2019), Gan et al. (2020), Lyu et al. (2019), Schumilo et al. (2021), Pailoor et al. (2018), Schumilo et al. (2017), Chen et al. (2020), Yun et al. (2018), Chen et al. (2019b)	10	2017–2021
Network and Distributed System Security (NDSS)	Rawat et al. (2017), Wang et al. (2021), Zhang et al. (2022), Aschermann et al. (2019), Schumilo et al. (2020), Kim et al. (2020), Song et al. (2019), Stephens et al. (2016), Wang et al. (2020), Zhao et al. (2019)	10	2016–2022
ACM Conference on Computer and Communications Security (CCS)	Böhme et al. (2017), Chen et al. (2018), He et al. (2019), Chen et al. (2019a), Han and Cha (2017), Corina et al. (2017), Petsios et al. (2017)	7	2017–2019
International Conference on Software Engineering (ICSE)	Wang et al. (2019), You et al. (2019), Bugariu and Müller (2020), Wen et al. (2020), Wang et al. (2020), Nilizadeh et al. (2019), Brennan et al. (2020), Luo et al. (2021)	8	2019–2021
ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)	Lyu et al. (2022), Deng et al. (2023), Xie et al. (2019), Lemieux et al. (2018)	4	2018–2023
The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)	Li et al. (2019), Mansur et al. (2020), Li et al. (2017), Liang et al. (2018)	4	2017–2020
IEEE/ACM International Conference on Automated Software Engineering (ASE)	Lemieux and Sen (2018), Jiang et al. (2018), Zhou et al. (2020)	3	2018–2020

Journal of Computer Science and Network Security. To gather applications of fuzzing in various domains, we also selected papers from artificial intelligence journals to comprehensively and systematically summarize fuzzing techniques. The relevant papers are shown in Table 3.

(4) We collected various fuzzing papers from the arXiv platform, which have been cited by many journals and conferences. TriforceAFL (Jesse (2015)), Trinity (Jones (2010)), Syzkaller (Vyukov 2015), are open-source kernel-related fuzzers that are widely used for testing kernel. They are frequently utilized as benchmark tools in numerous kernel fuzzing-related papers. Thus, we present an overview of the primary design concepts of these tools. The relevant papers and web sources are shown in Table 4.

2.2 Process of fuzzing

The goal of fuzzing is to generate different inputs and uncover as many exceptions as possible. Before fuzz testing, the input format is known and a target program is obtained. Figure 2 illustrates the general process of fuzzing. The workflow is composed of four main stages, test case generation, target program execution, exception monitoring, and Handling exceptions.

Test Case Generation Test cases can be generated using different methods, including mutation-based and generation-based methods. Mutation-based fuzzing involves taking existing valid test cases or inputs and applying random mutations to generate new test cases. Generation-based fuzzing

Table 2 Other conference papers

Conference	Paper	Number	Year
Annual Computer Security Applications Conference (ACSAC)	Güler et al. (2020), Jain et al. (2018), Liu et al. (2018)	3	2018–2020
International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)	Henderson et al. (2017), Nguyen et al. (2020), Chen et al. (2020a)	3	2017–2020
ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)	Winterer et al. (2020), Ye et al. (2021)	2	2020–2021
IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)	Li et al. (2021), Zhang et al. (2020)	2	2020–2021
International Conference on Software Testing, Validation and Verification (ICST)	Pham et al. (2020), Zhao et al. (2019)	2	2019–2020
International Symposium on Software Reliability Engineering (ISSRE)	Sun et al. (2020)	1	2020
International Conference on Machine Learning (ICML)	Odena et al. (2019)	1	2019
Black Hat Europe	Jack and Li (2016)	1	2017
International Conference on Computer Aided Verification (CAV)	Blotsky et al. (2018)	1	2018
Software Verification: International Conference	Scott et al. (2020)	1	2020
Proceedings of the ACM on Programming Languages (POPL)	Winterer et al. (2020)	1	2020
IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)	Zhang et al. (2021)	1	2021
International Conference on Sustainable Technologies for Computational Intelligence-Proceedings (ICTSCI)	(Vinesh et al. 2020)	1	2020
International Conference on Security and Privacy in Communication Systems (SecureComm)	Gascon et al. (2015)	1	2015
International Workshop on Automation of Software Test (AST Workshop)	Tsankov et al. (2012)	1	2012
USENIX Workshop on Offensive Technologies (WOOT)	Fioraldi et al. (2020)	1	2020
ACM/IEEE Design Automation Conference (DAC)	Luo et al. (2020)	1	2020
Totol	N/A	24	2012–2023

Table 3 Journal papers

Journal	Paper	Number	Year
IEEE Transactions on Software Engineering	BÖhme et al. (2019), Pham et al. (2019), Zhang et al. (2022), Manès et al. (2019)	4	2019–2022
Computer & Security	Lin et al. (2021), Beaman et al. (2022)	2	2021–2022
Cybersecurity	Li et al. (2018)	1	2018
IEEE Transactions on Reliability	Liang et al. (2018b)	1	2018
Empirical Software Engineering	Wang et al. (2021)	1	2021
Journal of Computer Science and Technology	Situ et al. (2021), Zhang et al. (2022)	2	2021
Expert System with Applications	Zhao et al. (2022)	1	2022
Mathematics	Zhao et al. (2021)	1	2021
IEEE Access	Wang et al. (2019)	1	2019
International Journal of Computer Science and Network Security	Gorbunov and Rosenbloom (2010)	1	2010
Journal of Intelligent Manufacturing	Lv et al. (2020)	1	2020
Totol	N/A	16	2010–2022

involves generating test cases from scratch based on predefined templates, grammars, or input specifications.

Target Program Execution Once new test cases are generated, they are sent to the target program for execution. To facilitate monitoring and analysis during fuzzing process, the target program is often instrumented to collect information.

Exception Monitoring During the execution of the target program, it is continuously monitored for program behaviors. This monitoring is essential to identify if the program crashes and hangs. Various techniques can be employed for exception monitoring, such as using signals, crash analysis, and violation detection. Related tools such as Sanitizers (The home for Sanitizers 2019) and MEDS (Han et al. 2018) are commonly used to detect and locate bugs.

Handling Exceptions If the target program encounters an exception during execution, the corresponding test case that triggered the exception is saved for further analysis. These test cases are considered valuable as they can potentially reveal bugs or vulnerabilities in the program.

Exception Analysis After obtaining the test cases that triggered exceptions, testers analyze and debug target program to obtain the cause of these exceptions. Debugging tools, such as GDB (1988), IDA (2003), and OllyDbg (2000), are commonly used to assist exception analysis.

2.3 Classification of fuzzing

There are different classifications of fuzzing, as shown in Fig. 3.

Fuzzing can be divided into generation-based and mutation-based (Neystadt 2008). Inputs are generated from scratch in generation-based fuzzing (Godefroid et al. 2008b), and it is necessary to provide the expected input specification of target programs. Then, fuzzers construct inputs that violate some regulations to feed target programs according to input specifications. If there is no better input specification, fuzzing might spend more time executing error-handling code and cannot reveal bugs. A mutation-based fuzzing, new test cases are derived from existing seed mutations. Generally speaking, initial valid seeds are provided, and then the fuzzer continuously uses mutation strategies (e.g., bitflip) to generate new test cases which are provided to continuously execute the target program. Compared with generation-based fuzzers, mutation-based fuzzers are relatively simple. However, it is affected by the quality of initial seeds and may be difficult to pass program verification with complex input formats.

Fuzzing can be divided into dumb and smart fuzzing (Neystadt 2008). The dumb fuzzing (Takanen et al. 2018; Ganesh et al. 2009) cannot understand input formats. Inputs are mainly generated using random mutations. Generally, a dumb fuzzer is faster than a smart fuzzer and has a relatively wide range of applications. For instance, AFL is a dumb mutation-based fuzzer. It uses mutation strategies to mod-

Table 4 arXiv papers and web sources

arXiv/Web source	Paper	Number	Year
arXiv	Wang et al. (2020), Saavedra et al. (2019), Wang et al. (2020), Wang et al. (2020)	4	2019–2020
Web source	Jones (2010), Vyukov (2015), Jesse (2015)	3	N/A
Totol	N/A	7	2012–2023

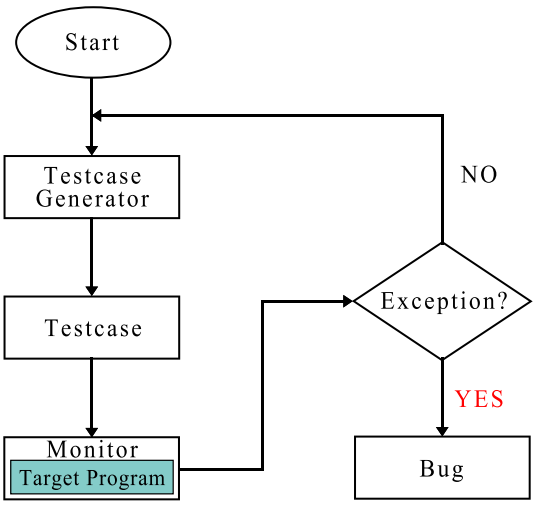


Fig. 2 The workflow of fuzzing

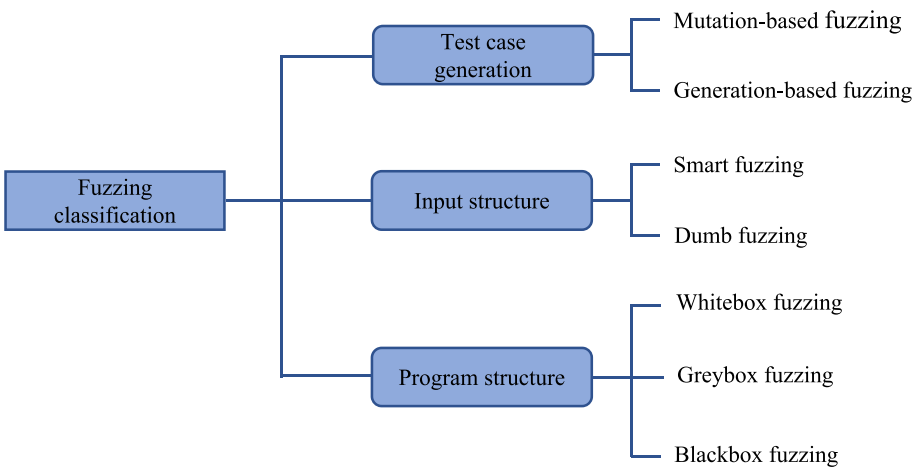
ify seed files and can fuzz pictures, audio, video, and other processing programs. A smart (model-based (Pham et al. 2016), grammar-based (Godefroid et al. 2008b), or protocol-based (Banks et al. 2006)) fuzzer leverages the input model to generate a greater proportion of valid inputs. For instance, a grammar-based fuzzer can use an abstract syntax tree to build an input model and then use node subtree mutation to transform the current subtree into a new subtree, to meet relevant requirements of grammars. However, smart fuzzers generally require users to provide an input model. The input model is specific and the structure is more complicated.

Fuzzing can be divided into whitebox, greybox, and blackbox fuzzing (Sutton et al. 2007). A whitebox fuzzer (Godefroid et al. 2008c) generally has obtained the source code of target programs. It leverages program analysis to systematically to reach critical locations of programs and increase code coverage. Symbolic execution is commonly employed in whitebox fuzzers. Since the source code of programs is available, symbolic execution is often used to constrain paths to generate test cases that meet specific constraints. Therefore, whitebox fuzzers can detect deeper bugs in the program. A blackbox fuzzer (Edwards 2001) is completely different from the white box fuzzer, it treats the program as a "black box," and it cannot perceive any information of programs. However, it can only detect bugs on the surface of the program. A greybox fuzzer (BÖhme et al. 2019) is between whitebox and blackbox fuzzer, it uses lightweight instrumentation instead of program analysis to obtain program information that is used to guide fuzzing to improve the efficiency of fuzzing. Because of its simplicity, effectiveness, and reasonable performance, greybox fuzzers have become effective testing tools.

2.4 CGF

In recent years, fuzzing also derives many professional terms, such as seed, seed queue, and mutation strategy. These concepts run through the whole process of CGF, and the related concepts are presented in Table 5.

Fig. 3 Classification of fuzzing



Based on the principle that better coverage is beneficial to detect more vulnerabilities, CGF uses coverage information to guide the fuzzer to improve coverage. CGF includes two stages: the static analysis stage and the fuzzing loop stage. In the static analysis stage, it executes compile time or dynamic binary instrumentation to obtain the instrumented target program. Algorithm 1 shows the workflow of CGF in the fuzzing loop stage. CGF uses a set of inputs provided by users as initial seeds and selects a seed to enter a continuous loop to fuzz until timeout or program terminates.

(1) A seed is selected from a seed queue (Line 4).
(2) The energy of seed is allocated (Line 5).
(3) The selected seed is mutated to generate a test case using mutation strategy (Line 7).
(4) The target program is executed with the generated test case (Line 8).
(5) The lightweight instrumentation is used to obtain coverage information to guide fuzzers, if the test case causes a crash, it is marked and added to the crash set (Lines 9–11).
(6) If the test case achieve new coverage, CGF adds it to the seed set (Lines 12–14).

In this section, we use a representative fuzzer AFL to introduce the relevant stages of CGF. The framework of AFL can be shown in Fig. 4.

Algorithm 1: CGF

```
   Input: Seed Input seeds, an instrumented target program P
   Output: a seed queue Q, a crash set C
1  Q ← seeds
2  C ← ∅
3  while TRUE do
4      S ← ChooseNext(Q)
5      E ← AssignEnergy(S)
6      for i in Range(0, E) do
7          S' ← Mutation(S)
8          status ← Run_Target(P, S')
9          if is_Crash(status) then
10             C ← C ∪ S'
11             return
12         if is_NewCoverage(status) then
13             Q ← Q ∪ S'
14             return
```

2.4.1 Instrumentation

Instrumentation is a common technique of inserting code fragments to trace the related information of programs without breaking the program logic. There are two ways of instrumentation, source code instrumentation and dynamic binary instrumentation. The former is inserting the assembly code in source code during the compilation process by static analysis or writing Clang (2007) manually. These correspond to the "$afl-gcc$" and "$afl-clang$" instrumentation ways in AFL, respectively. The latter is mainly implemented using binary dynamic instrumentation frameworks, such as Pin (Luk et al. 2005), Dynamorio (2015), PaiMei (2016), and Frida (2016).

Coverage-based greybox fuzzing techniques commonly employ instrumentation to collect coverage information, including edge and basic block coverage. For instance, HonggFuzz (2015) utilizes basic block coverage as feedback information, while AFL adopts edge (branch) coverage for feedback. Before the fuzzing loop stage, it uses instrumentation to insert code fragments. AFL preserves a 64 KB shared memory $Bitmap$ to record edge coverage information. AFL assigns a random number to each basic block in the program at compile time to indicate a unique identifier for the current basic block and uses XOR and right-shift operations on the

Table 5 Related concepts of CGF

Name	Description
Seed queue	A seed queue, also known as a seed pool, is a seed set
Seed	Seeds are the initial test cases provided by the user, or the high-quality test cases found during the fuzzing process. High quality is defined as being able to result in significant in the number of times a new path is discovered or the number of times an edge (branch) in the program is executed, etc.
Seed selection	Methods for selecting seeds in the seed queue
Energy	The number of test cases generated by a seed after mutation is called energy
Power schedule	Power schedule is also known as energy allocation, a calculation that assigns energy to seeds
Mutation strategy	The mutation strategy is a method of seed mutations
Fuzzed	Seed attribute, whether the seed has been fuzzed or not
Interesting	Fuzz testing found interesting test cases that cause crashes or increase code coverage, etc.
Favored	Seed attribute, seeds in the seed queue are marked as favored if they cover all the current edges and have the smallest execution time and seed length
Shared memory	Shared memory between the fuzzer and the target program is used to store and record the program's coverage information during the fuzzing process

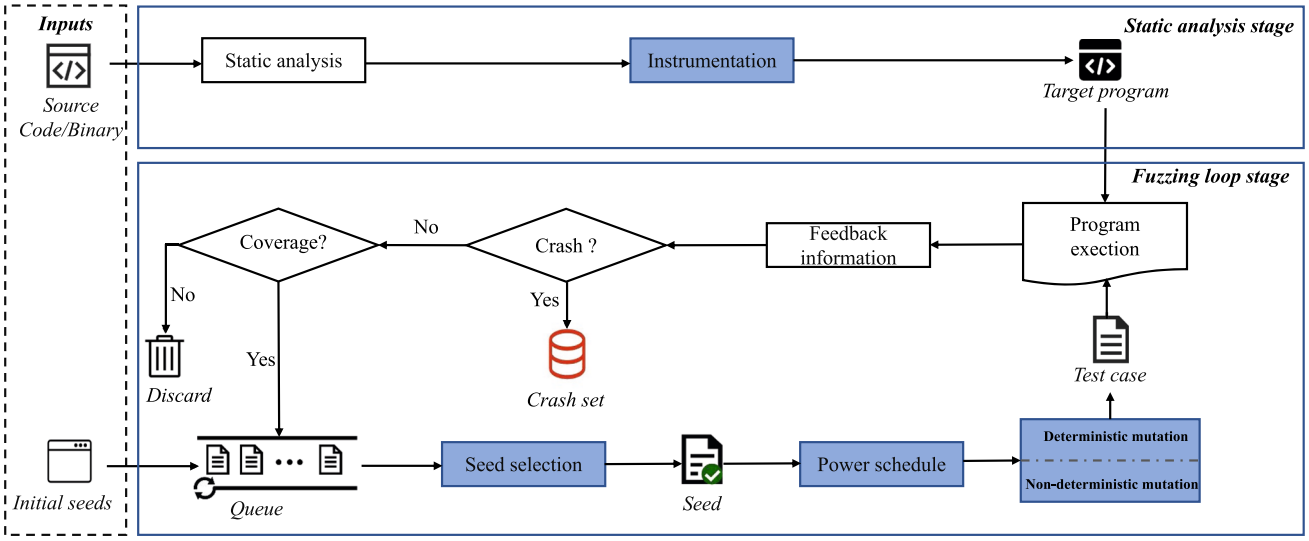


Fig. 4 The framework of AFL

Fig. 5 Coverage calculation

```
cur_location = Random();
Bitmap[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

current basic block and the previous basic block to mark each edge. Each edge is used as an offset of *Bitmap* and the value is the count of hits.

The specific formula for coverage calculation is as shown in Fig. 5.

2.4.2 Seed selection and power schedule

Seed selection refers to select seeds from the seed pool for future mutation. A perfect seed selection scheme is conducive to speeding up path discovery and bug detection. AFL gives priority to seeds that are *unfuzzed* and *favored*.

Power schedule aims at allocating energy to each seed during the fuzzing process, which determines the number of test cases generated by a seed after mutation. Reasonable energy allocation can effectively improve the discovery of new paths. If the energy of a seed is over-allocated, other seed's mutation will be affected. Conversely, if the energy of one seed is under-allocated, it will be detrimental to path discovery and potential bug detection. AFL has two energy allocations based on different mutation stages.

In the deterministic stage, energy is related to seed length. The longer the seed size, the more energy will be consumed.

In the non-deterministic stage, energy allocation depends on the running time, the number of edges, the average size of seed files, the number of cycles, and others.

2.4.3 Mutation strategy

The mutation strategy determines how to mutate and which part of the seed is mutated. Mutation strategies in AFL consist of two stages: deterministic stage and non-deterministic stage. The former includes bitflip, arithmetic, interest, and dictionary. It is applied to seeds that are selected to mutate for the first time. The latter includes havoc and splice.

The bitflip uses different flip lengths and steps. It includes bitflip 1/1, bitflip 2/1, bitflip 4/1, bitflip 8/8, bitflip 16/8, and bitflip 32/8. In the bitflip, AFL has some heuristic judgments on the file format, such as automatic detection of token and generation of effector map.

The arithmetic performs integer addition and subtraction mutations. It includes arith 8/8, arith 16/8, and arith 32/8, which mean to perform addition and subtraction operators on 8, 16, and 32 bits each time, respectively, starting from the beginning of the seed file according to the step length of each 8 bits. The big endian and little endian are also considered in arithmetic.

The interest performs substitution using pre-define interesting values. It consists of interest 8/8, interest 16/8, and interest 32/8 in steps of 8 bits, which means that starting from the beginning of the seed file, the interesting values of 8, 16, and 32 bits are replaced one at a time, depending on each 8-bit step, respectively.

The dictionary performs substitution using user-provided tokens including user extras (over), user extras (insert), and auto extras (over). The user extras (over) and user extras (insert) indicate to replace and insert into the seed file with the tokens provided by the user, respectively. The auto extras (over) use automatically generated tokens in bitflip to replace seed file.

The non-deterministic stage, in which the havoc randomly selects a random position of seed files to mutate according to mutation strategies in deterministic stage, and the splice splits each of the two seed files in two and splices the head and tail.

2.5 Related work

There are papers that systematically introduce the previous advances in fuzzing, as shown in Table 6. Li et al. (2018) conducted a comparative study that compared fuzzing with other vulnerability discovery techniques and provided an overview of the research achievements in 2018. Liang et al. (2018b) categorized and reviewed relevant papers on fuzzing from 1990 to 2017, based on different classifications and applications. Manès et al. (2019) proposed a unified and general fuzzing model to address the existing inconsistencies in the concepts related to fuzzing and provide a standardized framework for understanding and discussing fuzzing techniques. Beaman et al. (2022) examined the latest developments in fuzzing for vulnerability discovery, proposed a method for classifying fuzzers, and discussed key research challenges and potential future research directions.

Other review papers primarily focused on summarizing fuzzing in specific application domains or providing an overview of a specific type of fuzzing. Wang et al. (2020) provided the first in-depth study of directed grey-box fuzzing (DGF). They performed an extensive review of 42 state-of-the-art fuzzers, meticulously categorizing the recent advancements while also providing a comprehensive overview of the associated challenges. Eisele et al. (2022) reviewed embedded fuzzing and proposed a formal definition of embedded fuzzing, and carved out the additional challenges of embedded fuzzing compared to traditional fuzzing. Saavedra et al. (2019) and Wang et al. (2020) reviewed the application of machine learning in fuzzing.

Table 6 A summary of previous surveys on fuzzing

Name	Description	Year
Li et al. (2018)	Recent advances in fuzzing are summarized and ways to improve the fuzzing process and future work are discussed	2018
Liang et al. (2018b)	Classification and presentation of fuzzers since 1990 and discussion of future research directions	2018
Manès et al. (2019)	A taxonomy and standardized terminology are proposed and the use of "model fuzzer" is used to explain fuzzer design choices	2019
Wang et al. (2020)	Recent advances in DGF are summarized	2020
Saavedra et al. (2019)	Outlining current research on the application of machine learning to fuzzing and predicting future challenges	2020
Wang et al. (2020)	Advances in machine learning research in fuzzing are reviewed, stages in which machine learning is applicable to fuzzing are discussed	2020
Beaman et al. (2022)	Recent advances in fuzzing are reviewed, classification methods for fuzzing are proposed, and potential future research areas are discussed	2022

3 State-of-the-art fuzzing

In this section, we first summarize the research questions (RQs) of fuzzing according to the fuzzing process and then answer the following questions by using state-of-the-art fuzzing.

Table 7 General fuzzing

ID	Class	Fuzzer
1	Initial seed selection	Skyfire (Wang et al. 2017).
2	Seed selection optimization	VUzzer (Rawat et al. 2017), CollAFL (Gan et al. 2018), Cerebro (Li et al. 2019), TortoiseFuzz (Wang et al. 2020), K-Scheduler (She et al. 2022), MooFuzz (Zhao et al. 2021), NeuFuzz (Wang et al. 2019), MEUZZ (Chen et al. 2020a), AFL++hier (Wang et al. 2021), AFLGO (Böhme et al. 2017), Hawkeye (Chen et al. 2018), TOFU (Wang et al. 2020).
3	Power schedule	AFLFast (BÖhme et al. 2019), EcoFuzz (Yue et al. 2020), RegionFuzz (Situ et al. 2021), OTA (Li et al. 2021), MobFuzz (Zhang et al. 2022), AFLGo (Böhme et al. 2017), AFL++hier (Wang et al. 2021), SLIME (Lyu et al. 2022).
4	Mutation strategy	FairFuzz (Lemieux and Sen 2018), AFLTurbo (Sun et al. 2020), ProFuzzer (You et al. 2019), Superion (Wang et al. 2019), AFLSmart (Pham et al. 2019), GRIMOIRE (Blazytko et al. 2019), SLF (You et al. 2019), Steelix (Li et al. 2017), REDQUEEN (Aschermann et al. 2019), T-Fuzz (Peng et al. 2018), Angora (Chen and Chen 2018), Matryoshka (Chen et al. 2019a), GreyOne (Gan et al. 2020), PATA (Liang et al. 2022), MOPT (Lyu et al. 2019), CMFuzz (Wang et al. 2021), AMSFuzz (Zhao et al. 2022).

RQ1 How to get initial seeds?
RQ2 How to select seeds?
RQ3 How to assign energy for seeds?
RQ4 How to mutate seeds and select mutation strategies?
RQ5 How to quickly detect specific bugs?
RQ6 How to integrate other techniques to improve the performance of fuzzers?

3.1 General fuzzing

General fuzzing mainly optimizes and improves the greybox fuzzing process to improve code coverage and find more program exceptions. Its main research challenges include how to select the initial seed, how to select the seed from the seed queue, how to assign energy to the seed and what mutation strategy to adopt, based on the above research questions, the related works are shown in Table 7.

3.1.1 Initial seed selection (RQ1)

The initial seeds are user-supplied test cases prior to fuzzers, which is extremely important for mutation-based fuzzing. Because the test cases are generated by mutating the existing seeds. Ideally, a high-quality seed requires meeting the following conditions: (1) There is a good format that matches target applications. (2) The initial seed size should be as small as possible under the condition of meeting the rules so that the efficiency of running is high. (3) The initial seeds can generate test cases with deep bugs after mutation. (4) Good seeds can be used many times in constant fuzzing.

Based on the above conditions, initial seeds are generally based on the following selection criteria: (1) Some well-formatted data sets Fuzzdata (2015). (2) The proof of concept (POC) CVE (2016) of target applications. Well-formatted data sets are generally available. Some target applications provide some test cases to aid fuzzing. Specific types of applications, such as image, audio, and video processing programs (Ju et al. 2021), have related image, video, and audio libraries that can be selected as the initial seed set. Users can also use online crawlers and other techniques to crawl the required data as initial seeds after filtering, trimming, and other processing. Generally, a POC that triggers vulnerabilities in the older version of applications is a good seed, because the vulnerability is relatively risky, even if it is repaired, there may be risks. Nowadays, many testers have published some POCs of target applications on GitHub and other platforms that provide good initial seed sets. Researchers have also studied initial seed generation, and Wang et al. (2017) pro-

Table 8 Different fuzzers with seed selection optimization

Fuzzer	Seed selection strategy	Technique	Type	Open source	Year
VUzzer (Rawat et al. 2017)	Prioritize seeds with deeper execution paths	Static analysis	Greybox	✓	2017
CollAFL (Gan et al. 2018)	Prioritize non-neighboring branch seeds	Instrumentation	Greybox		2018
Cerebro (Li et al. 2019)	Prioritize high code complexity seeds	Static analysis	Greybox		2019
TortoiseFuzz (Wang et al. 2020)	Prioritize seeds that reach sensitive locations	Instrumentation	Greybox	✓	2020
K-Scheduler (She et al. 2022)	Prioritize seeds that potentially accessible CFG edge	Graph centrality analysis	Greybox	✓	2022
MooFuzz (Zhao et al. 2021)	Seed schedule based on various seed states	Intelligent optimization	Greybox		2021
NeuFuzz (Wang et al. 2019)	Prioritize seeds that execute vulnerable paths	Deep learning	Greybox	✓	2019
MEUZZ (Chen et al. 2020a)	Prioritize high-yield seeds	Machine learning	Hybrid	✓	2021
AFL++hier (Wang et al. 2021)	Hierarchical seed schedule	Reinforcement learning	Greybox fuzzer	✓	2021
AFLGO (Böhme et al. 2017)	Prioritize seeds closer to the target location	Instrumentation	Directed greybox	✓	2017
Hawkeye (Chen et al. 2018)	Prioritize seeds that cover new branches and have high similarity to the objective function and target point.	Instrumentation	Directed greybox		2018
TOFU (Wang et al. 2020)	Prioritize seeds closer to the target location	Instrumentation	Directed greybox		2020

posed Skyfire in the 2017 S&P conference, the main idea is to use a large number of existing samples to learn probabilistic context-sensitive grammars and generate highly structured test cases as the initial seeds of AFL, perform fuzzing and detect many vulnerabilities.

3.1.2 Seed selection optimization (RQ2)

Fuzzing generates high-quality test cases such as new coverage or significant features called seeds and saves them in the seed queue, how to choose which seed to use in the next round from the seed queue seriously affects fuzzing efficiency. The research related to seed selection optimization is presented in Table 8.

Seed selection optimization can drive fuzzing to evolve in different directions. VUzzer (Rawat et al. 2017) uses static analysis to extract control flow graphs, assigns weights to basic blocks in the control flow graphs, computes the weights of the seeds to quantify the depth of their execution paths, and prioritized the seeds with deeper execution paths. Gan et al. (2018) developed CollAFL, which uses lightweight instrumentation to solve the hash collision problem, and proposed untouched-neighbor-branch guided policy, untouched-neighbor-descendant guided policy, and memory-access guided policy three seed selection strategies to improve code coverage and find more bugs. Cerebro (Li et al. 2019) measures code complexity and selects seeds by combining various attributes such as the code complexity, the coverage, and the execution time. Wang et al. (2020) proposed a fuzzer called Tortoisefuzz. Based on the observation that not all coverage measurements are equal, Tortoisefuzz is proposed to discover memory corruption from function calls, basic blocks, and loops. The instrumentation is modified to optimize the input of dangerous functions, memory read and write in basic blocks, and loops to obtain sensitive edge information. The obtained information is used to guide seed optimization to find more vulnerabilities. K-Scheduler (She et al. 2022) models the seed selection problem for fuzzing

as a graph centrality analysis problem, constructing an edge horizon graph using a control flow graph and using the Katz centrality to compute the centrality score to approximate the number of uncontrolled flow graph edges that are reachable and feasible for the seed from a starting point, and preferentially scheduling the uncontrolled but potentially reachable CFG edges with seeds that have more. MooFuzz (Zhao et al. 2021) integrates vulnerability and improved coverage perspectives, divided the seed pool into exploration, search, and evaluation three states and employed different many-objective optimization schemes for seed selection for the different states.

Deep learning, machine learning, and other techniques have also been used to aid fuzzing for seed selection optimization. NeuFuzz (Wang et al. 2019) uses neural network models to predict whether a path is vulnerable or not, prioritize vulnerable path-related seeds are prioritized and seeds are selected. MEUZZ (Chen et al. 2020a) uses the use of supervised machine learning to select seeds based on past knowledge learn from past seed scheduling decisions on the same or similar procedures, to select seeds that are expected to yield a greater yield to improve the efficiency of hybrid fuzzing. Wang et al. (2021) proposed a multi-level coverage measurement approach, which models the fuzzing process as a multi-armed bandit model. They utilized the upper confidence bound (UCB1) algorithm to score the seeds based on the rarity of the seeds and the difficulty of generating new coverage paths through seed mutations on a hierarchical tree. The seeds with higher scores are selected.

Directed greybox fuzzing is designed to perform tests on pre-selected or potentially vulnerable target locations, which are mainly used in various scenarios such as vulnerability recovery and patch testing. Therefore, seeds are also often selected based on target locations in directed fuzzing. AFLGO (Böhme et al. 2017) is a directed fuzzer that dynamically calculates the distance between a seed and a user-given target location, and prioritizes seeds that are closer to the target location. Hawkeye (Chen et al. 2018) addresses the limitations of the fuzzer AFLGO by defining more accurate distances, preferring seeds that cover new branches and have greater similarity to the objective function and target point. TOFU (Wang et al. 2020) is a target-oriented directed fuzzer that performs structured mutations using knowledge of the input structure provided by the user, based on which the distances of the basic blocks are calculated and seeds that are more likely to reach the target location are selected.

3.1.3 Power schedule (RQ3)

Power schedule is to allocate energy to seeds. Energy determines the number of test cases generated by the seed mutation. Proper allocation of energy to seeds not only gives other seed mutation opportunities, it also improves the testing efficiency. The research related to power schedule is presented in Table 9. AFLFast (BÖhme et al. 2019) models the fuzzing through Markov chain and uses transfer probability to represent the probability of seed mutation to generate other test paths, gives the concept of high-frequency and low-frequency paths, and preferentially allocates more energy to the seed executing low-frequency paths to improve the efficiency of fuzzing. EcoFuzz (Yue et al. 2020) selects seeds according to the divided states of the seed pool and models the seed energy allocation problem as a multi-armed bandit problem, using reinforcement learning to allocate energy to seeds. RegionFuzz (Situ et al. 2021) adopts code metrics to evaluate vulnerable regions in the code and allocates more energy to the seeds that reach the region to improve vulnerability detection. OTA (Li et al. 2021) transforms the energy allocation of the deterministic stage of AFL into a mutation time allocation problem by using a particle swarm optimization (PSO) algorithm to optimize the mutation time. MobFuzz (Zhang et al. 2022) selects execution speed, memory consumption, and deep nested branches as optimization objectives, models fuzzing as a multi-armed bandit model, and allocates reasonable energy to seeds based on different combinations of objectives. AFL++hier (Wang et al. 2021) uses a multi-armed bandit model to allocate energy among different clusters of seeds with multi-level coverage metrics.

Table 9 Various fuzzers with power schedule

Fuzzer	Main technique/method	Type	Open source	Year
AFLFast (BÖhme et al. 2019)	Markov chain	Greybox	✓	2019
EcoFuzz (Yue et al. 2020)	Multi-armed bandit	Greybox	✓	2020
RegionFuzz (Situ et al. 2021)	Code metrics	Greybox		2021
OTA (Li et al. 2021)	Particle swarm optimization	Greybox	✓	2021
MobFuzz (Zhang et al. 2022)	Multi-armed bandit	Greybox	✓	2022
AFLGo (Böhme et al. 2017)	Simulated annealing	Directed greybox	✓	2017
AFL++hier (Wang et al. 2021)	Multi-armed bandit	Greybox	✓	2021
SLIME (Lyu et al. 2022)	Upper confidence bounds-variance aware	Greybox	✓	2022

AFLGo (Böhme et al. 2017) uses the simulated annealing (SA) algorithm to allocate more energy to seeds that are closer to the user's given target. SLIME (Lyu et al. 2022) uses an upper confidence bounds-variance aware algorithm to adaptively allocate energy based on the path and crash discovery capabilities of attribute queues by estimating the potential reward of each attribute queue.

3.1.4 Mutation strategy (RQ4)

The mutation strategy determines where the seeds are mutated, how they are mutated, and how the mutation operator is chosen. The research related to mutation strategy is presented in Table 10. FairFuzz (Lemieux and Sen 2018) uses new mutation strategies to address the limitation of low coverage of AFL. It identifies branch paths that are less frequently executed as rare branches and a mutation mask algorithm that allows mutations is proposed to reach rare branches, improving code coverage. AFLTurbo (Sun et al. 2020) employs interruptible mutation to determine mutation time, locality-based mutation to reduced mutation regions, and hotspot-aware fuzzing to identify metadata to reduce mutation overhead and improve code coverage. ProFuzzer (You et al. 2019) performs byte-level mutation and observes the results of fuzz testing to delineate input fields and infer field types, guiding seed mutations based on different field types to improve path discovery and vulnerability detection.

To generate test cases that conform to the desired format, multiple heuristic mutation strategies have been proposed to generate valid inputs. Superion (Wang et al. 2019) adds two mutation strategies to AFL to generate structured inputs, including enhanced dictionary-based mutation and tree-based mutation. Enhanced dictionary-based mutation locates the boundaries of tokens by checking whether the bytes of test cases are consecutive alphabet or digit and then inserting tokens in dictionary to every boundary. Tree-based mutation replaces the abstract syntax tree of the parsed input with a subtree. These strategies improve the effectiveness of test case generation while reducing the number of mutations. AFLSmart (Pham et al. 2019) uses a high-level virtual structure to represent seed files and divided them into chunks, then executing chunk-level smart deletion, addition, and splicing operators based on virtual input structure, to generate new valid inputs. GRIMOIRE (Blazytko et al. 2019) implements a syntax inference that is used to automatically fuzz highly structured formats during the fuzzing process without human interaction or program modification. It modifies inputs by removing element parts of the input and marks the element parts that are the same as the original input in achieving coverage, to learn the structure of inputs, and performs large-scale mutations on learned structures to generate structural inputs. SLF (You et al. 2019) performs a dependency analysis to generate valid seed inputs. Based on AFL, it flips each byte of random inputs to generate new test cases to execute the program. The mutated consecutive bytes affect the same comparison in constantly mutating and are marked as identical fields of inputs. Based on the information of input fields, the corresponding input checks are classified, including arithmetic, index/offset, count, and ITE ($If - Then - Else$) checks. A multi-goal search algorithm is used to mutate inputs to satisfy inter-dependent checks to generate valid seeds.

Sanity checks often appear in programs, such as checks on magic bytes, checksums, hashes, and others. Magic bytes are bytes that are used for comparison instructions in the inputs, such as string equality comparison. It is difficult for fuzzing to pass magic byte comparison. To pass the sanity check, corresponding mutation strategies have been proposed. Steelix (Li et al. 2017) is used to generate test cases to pass magic bytes. It collects coverage information and performs lightweight static analysis and binary instrumentation to finish comparisons. For certain multi-byte magic numbers, Steelix can accurately detect single-byte matching. By fixing its corresponding generated correct byte position, Steelix traverses the front or back bytes exhaustively to generate the corresponding magic byte, thereby reducing the mutation space. REDQUEEN (Aschermann et al. 2019) uses a lightweight tracking-based technique to generate test cases that can pass the magic byte and checksum. Based on a simple assumption that part of the inputs is directly transferred to memory or register for comparison at runtime, a strong input-state correlation exists between input and current state. REDQUEEN first tracks and recognizes the comparison instructions during the program execution, and then determines which part of the input affects the change of memory and register, and finally mutates to generate new coverage. T-Fuzz (Peng et al. 2018) uses deletion of the sanity check in target programs to ensure that fuzzing execution when fuzzing has no new coverage. The technique based on symbolic execution is used to filter false positives and reproduce the real bug in the original program.

Many fuzzers incorporate taint analysis techniques, which gather data flow information and infer which bytes in the input can influence program execution. This guidance is used to guide seed mutations, resulting in improved test case generation and increased code coverage. Angora (Chen and Chen 2018) uses byte-level taint tracking, context-sensitive branch count, search based on gradient descent, and input length exploration instead of symbolic execution to generate high-quality input to solve path constraints and increase branch coverage. Matryoshka (Chen et al. 2019a) identifies the control flow dependencies and taint flow dependencies between conditional statements and then uses three strategies, including prioritizing reachability, prioritizing satisfiability and jointing optimization for both reachability and

Table 10 Various fuzzers with mutation strategy

Fuzzer	Objective	Method/technique	Open source	Year
FairFuzz (Lemieux and Sen 2018)	Improving code coverage	Mutation mask creation algorithm	✓	2018
AFLTurbo (Sun et al. 2020)	Reduce mutation overhead	Interruptible mutation, locality-based mutation, hotspot-aware fuzzing	✓	2020
ProFuzzer (You et al. 2019)	Delineate mutation fields	Type probing		2019
Superion (Wang et al. 2019)	Grammar-aware test case generation	Enhanced dictionary-based mutation, tree-based mutation	✓	2019
AFLSmart (Pham et al. 2019)	Valid test case generation	Chunk-level mutation	✓	2019
GRIMOIRE (Blazytko et al. 2019)	Highly structured test case generation	Syntax inference	✓	2019
SLF (You et al. 2019)	Valid test case generation	Dependency analysis		2019
Steelix (Li et al. 2017)	Magic byte check	Static analysis and binary instrumentation		2017
REDQUEEN (Aschermann et al. 2019)	Passing the magic byte and checksum	Taint tracking	✓	2019
T-Fuzz (Peng et al. 2018)	Sanity check	Symbolic execution	✓	2018
Angora (Chen and Chen 2018)	Solving path constraints	Taint tracking	✓	2018
Matryoshka (Chen et al. 2019a)	Passing deeply nested branches	Path constraints prioritizing reachability, prioritizing satisfiability, jointing optimization for both reachability and satisfiability		2019
GreyOne (Gan et al. 2020)	Mutation position and strategy determination	Taint inference		2020
PATA (Liang et al. 2022)	Solving path constraints	Dynamic taint analysis		2022
MOPT (Lyu et al. 2019)	Mutation strategy schedule	Particle swarm optimization	✓	2019
CMFuzz (Wang et al. 2021)	Mutation strategy schedule	Multi-armed bandit		2021
AMSFuzz (Zhao et al. 2022)	Mutation strategy schedule	Multi-armed bandit		2022

satisfiability to solve path constraints that involve deeply nested conditional statements. Greyone (Gan et al. 2020) uses a lightweight taint inference to guide fuzzing. During the fuzzing process, the taint of variables in the program is inferred by changing the bytes of inputs. A taint-guided mutation is used to determine which bytes to mutate, which branches to explore, where to mutate, and how to mutate, and then using constraint conformance calculation to guide seed selection to explore hard-to-reach branches. PATA (Liang et al. 2022) employs dynamic taint analysis techniques to identify the dependency between input and paths. It mutates the input bytes that affect the dependency relationship to resolve constraints.

A smart mutation strategy schedule can be highly fuzzing efficient. Based on different mutation strategies that have different effectiveness for different programs, MOPT (Lyu et al. 2019) uses a particle swarm optimization algorithm to perform mutation strategy schedule. It treats each mutation operator as a particle and iteratively updates the probability of each particle through the local optimum and the global optimum. Then, MOPT integrates the updated probabilities of all particles to obtain a new probability distribution. CMFuzz (Wang et al. 2021) uses a context-aware mutation method, which dynamically extracts byte stream features for the seed file during the fuzzing process, and then uses a multi-armed bandit algorithm to select the optimal mutation operation. AMSFuzz (Zhao et al. 2022) models the mutation operator selection problem in fuzzing as a dodging slot machine model that adaptively selects mutation operators to improve the efficiency of path discovery and vulnerability detection.

3.1.5 Summary of general fuzzing

The main purpose of general fuzzing is to optimize the process of greybox fuzzing to improve the quality of test cases, increase code coverage, and detect more vulnerabilities. The current research status can be summarized into four aspects: initial seed selection, seed selection optimization, energy allocation, and mutation strategies. Initial seed selection involves selecting the initial seed inputs for mutation-based fuzzing. This is achieved through methods such as crawling, historical POCs, and grammar-based learning. Seed selection optimization is a crucial aspect of general fuzzing. Current research utilizes techniques such as static analysis, intelligent optimization, and machine learning to evaluate seeds based on criteria such as coverage improvement, vulnerability triggering potential, and reaching target locations. This evaluation helps determine the priority of seed selection, thus improving testing efficiency. Energy allocation refers to the quantity of generated test cases through seed mutation. Current research employs machine learning, optimization algorithms, and reinforcement learning techniques to allocate reasonable energy based on seed evaluation results. Mutation strategies involve specific methods for mutating seeds. Current research utilizes techniques such as taint analysis, heuristic strategies, and mutation optimization to determine mutation positions, methods, and how to optimize mutation strategies to generate well-formed test cases, thereby improving the quality and efficiency of mutations.

With the increasing complexity of software, general fuzzing research demonstrates good generality and is continuously evolving and optimizing. However, it still faces challenges in detecting specific vulnerabilities.

3.2 Vulnerability-oriented fuzzing (RQ5)

General fuzzing has shown good generality, but it lacks an advantage in discovering specific vulnerabilities. Therefore, vulnerability-oriented fuzzing aims to uncover specific types of vulnerabilities. Its core idea is to analyze the behaviors related to vulnerabilities, guide fuzzing to satisfy the corresponding behaviors, thereby enabling faster discovery of vulnerabilities of that type. Based on different types of vulnerabilities or bugs, relevant research is presented in Table 11.

3.2.1 Uncontrollable memory consumption & uncontrolled recursive

MemLock (Wen et al. 2020) is utilized for the detection of uncontrollable memory consumption and uncontrolled recursive bugs. By employing instrumentation techniques within the fuzzing process, MemLock continuously gathers information on memory consumption and recursive function calls. Test cases that achieve new coverage and cause increased memory consumption or a higher number of recursive calls are selectively chosen by MemLock. Selected test cases are then added to a seed pool and assigned a higher priority, aiming to effectively trigger vulnerabilities.

3.2.2 Integer overflow and array overflow

TIFF (Jain et al. 2018) employs dynamic taint analysis to identify data types associated with input offsets during the program execution. It performs not only coverage-oriented mutation but also bug-oriented mutation, combining input types and bug trigger conditions, using a custom mutation strategy to generate test cases to detect two types of memory corruption (integer overflow and array overflow).

3.2.3 Memory vulnerability

ovAFLow (Zhang et al. 2022) utilizes static analysis to identify potential locations in the program that may lead to memory vulnerabilities. Specifically, it focuses on memory manipulation function parameters and memory loop vari-

Table 11 Fuzzers that detect various vulnerabilities

ID	Vulnerability	Name	Open source	Year
1	Uncontrollable memory consumption and uncontrollable recursive	MemLock (Wen et al. 2020)	✓	2020
2	Integer overflow and array overflow	TIFF (Jain et al. 2018)		2018
3	Memory vulnerability	ovAFLow (Zhang et al. 2022)	✓	2022
4	Consistency error	COMFORT (Ye et al. 2021)	✓	2021
5	Use-after-free	UAFL (Wang et al. 2020)		2020
		UAFuzz (Nguyen et al. 2020)	✓	2020
		MDFuzz (Zhang et al. 2021)		2021
6	Algorithmic complexity vulnerability	PerfFuzz (Lemieux et al. 2018)	✓	2018
		SlowFuzz (Petsios et al. 2017)		2017
7	Concurrency vulnerability	Liu et al. (Liu et al. 2018)		2018
		ConFuzz (Vinesh et al. 2020)	✓	2020
		MUZZ (Chen et al. 2020)		2020
8	Side channel attack	DifFuzz (Nilizadeh et al. 2019)	✓	2019
		Brennan et al. (2020)		2020

ables. By employing taint inference, ovAFLow establishes the corresponding relationships between input bytes and these identified locations. It then guides the fuzzing process by mutating seeds to trigger memory vulnerabilities within the program.

3.2.4 Consistency error

COMFORT (Ye et al. 2021) is used to detect consistency vulnerabilities in JavaScript engines. It uses a DNN-based deep learning language model, GPT-2, to generate random JS programs, and then extracts the boundary conditions of the test programs related to JS API from the ECMA-262 specification to guide the generation of multiple test case programs. The generated test case programs are sent to multiple JS engines to detect consistency issues through inconsistency analysis.

3.2.5 Use-after-free

The triggering of use-after-free (UAF) vulnerabilities requires the sequential execution of three specific operations: memory allocation, free, and memory use. Currently, fuzzing techniques face difficulties in detecting UAF vulnerabilities. To address this problem, UAFL (Wang et al. 2020) is proposed to detect UAF. It identifies operation sequences by static typestate analysis, and not only collects the control flow information and also collects operation sequence information to guide generate seeds that cover operation sequences, to detect UAF. UAFuzz (Nguyen et al. 2020) employs bug trace flattening to extract serialized basic blocks and functions related to UAF vulnerabilities from binary programs. It prioritizes seeds that are similar to these blocks and functions and allocates more energy to them to detect UAF vulnerabilities. MDFuzz (Zhang et al. 2021) uses static analysis to identify the locations of memory allocation, deallocation, and access as targets. It calculates the distance from each basic block to the targets and selects seeds based on this distance, thereby enhancing the triggering of UAF vulnerabilities.

3.2.6 Algorithmic complexity vulnerability

Algorithmic complexity vulnerabilities occur when the worst-case time or space complexity of an application is significantly higher than the average case for specific user-controlled inputs. To detect algorithmic vulnerabilities, PerfFuzz (Lemieux et al. 2018) first extracts control flow graphs to collect information on each edge of the target program by improving instrumentation. Then, PerfFuzz selects test cases that maximize execution counts of each edge in the control flow graph (CFG) as the pathological input. SlowFuzz (Petsios et al. 2017) is based on libFuzzer (Serebryany 2016) and prioritizes inputs that increase total path length to detect algorithmic vulnerabilities.

3.2.7 Concurrency vulnerability

To detect concurrency vulnerability, Liu et al. (2018) proposed a heuristic framework that uses static analysis to find sensitive concurrent operations and determines the order of execution of sensitive operations. Each specific execution sequence is explored to trigger a potential concurrency vulnerability. ConFuzz (Vinesh et al. 2020) uses static analysis to traverse the paths in the program. Each basic block in the path is instrumented and then assigned an id and a weight based on the distance of basic blocks to thread functions. During the fuzzing process, the information related to basic

blocks is used to generate new test cases, cover more basic blocks, and detect concurrency vulnerabilities with the aid of thread sanitizer ThreadSanitizer (2019). A greybox fuzzing framework for multi-threaded programs, MUZZ (Chen et al. 2020) used coverage instrumentation, thread context instrumentation, and thread scheduling instrumentation to explore the execution state of multi-threaded programs and then prioritizes those seeds that have explored new code coverage or thread contexts to detect concurrency vulnerabilities.

3.2.8 Side channel attack

Side channel attack exploits information leakage observed during the execution of certain operations to undermine security measures. To detect side channel vulnerabilities, DifFuzz (Nilizadeh et al. 2019) employs a resource-guided heuristic algorithm to test two different versions of an application. During the fuzzing process, DifFuzz generates test cases that maximize differences in resource consumption between the versions, such as time, memory, and response size, to detect side channel attacks. Brennan et al. (2020) proposed a fuzzing technique for detecting timing side channel vulnerabilities in Java programs caused by just-in-time compilation.

3.2.9 Summary of vulnerability-oriented fuzzing

Vulnerability-oriented fuzzing refers to fuzz testing based on the known features and triggering patterns of specific vulnerabilities to improve the efficiency of detection. Based on the above research, vulnerability-oriented fuzzing firstly requires to model known vulnerabilities and extract the vulnerability information to guide test cases generation to detect the type of vulnerability. Common methods include static analysis, dynamic analysis, deep learning, and other techniques to analyze and extract semantic information, contextual information, and potential locations of vulnerabilities. Based on the obtained information, techniques such as instrumentation and taint analysis are employed to improve and optimize seed selection, energy allocation, mutation strategies, and guide fuzzing in generating test cases that better uncover the targeted vulnerabilities. Therefore, vulnerability-oriented fuzzing is of significant importance in detecting specific types of vulnerabilities.

3.3 Combining fuzzing with other techniques (RQ6)

To enhance the efficiency of fuzzing, various techniques such as symbolic execution, taint analysis, parallel, and instrumentation are commonly integrated. Table 12 provides an overview of related work that incorporates these techniques into fuzzing.

3.3.1 Symbolic execution

Hybrid fuzzing adds symbolic execution based on traditional fuzzing. It is one of the current popular fuzzing branches. Driller (Stephens et al. 2016) consists of a greybox fuzzer AFL and a symbolic execution engine Angr (Wang and Shoshitaishvili 2017). AFL can quickly generate test cases to fuzz target programs. Angr can solve the constraints in the program and generate new test cases only when AFL does not find new paths. Driller combines the advantages of AFL and the dynamic symbolic execution Angr, it avoids the difficulty of AFL to break through special boundaries and the problem of dynamic symbolic execution path explosion. QSYM (Yun et al. 2018) is a concolic testing engine tailored for hybrid fuzzing. To improve performance, it does not translate the target program into an intermediate representation but uses a dynamic binary instrument framework, Inter Pin (Luk et al. 2005), to add symbolic tracing to the target program and employs pruning basic blocks and eliminating extraneous constraints to increase the speed of hybrid fuzzing. However, it only supports x86 system architectures.

Most fuzzers are mainly coverage guided that waste a lot of time on codes without bugs and it is difficult to reach protected codes that have complex conditions. A hybrid fuzzing framework, SAVIOR (Chen et al. 2020b), is designed based on bug-driven principles. SAVIOR uses the UBSan UndefinedBehaviorSanitizer (2019) to label potential bugs for the target program and uses static analysis to find the protected code region. During the fuzzing process, it optimizes seed selection according to unexplored branches, bug labels, and difficulty degree in branch exploration. Unlike other hybrid fuzzers, SAVIOR also adds bug-guided verification to verify all possible vulnerabilities in the execution path to ensure that no vulnerability is missed. DigFuzz (Zhao et al. 2019) is a probabilistic hybrid fuzzer, which is used to address the problems of schedule between concolic execution and fuzzing. It uses Monte Carlo path optimization to quantify the difficulty of path and assigns those paths for concolic execution. Since the overhead of hybrid fuzzing is huge, PANGOLIN (Huang et al. 2020) used polyhedral path abstraction to reuse the values solved by the constraint solver based on traditional hybrid fuzzing, which can improve the efficiency of constraint solving while using the results of reusing constraint solving to guide mutation in fuzzing. QuickFuzz (Lin et al. 2021) quantifies two factors of path solution demand and solution cost, and adopted a priority-based path searching method to select the missing path to execute the mixed execution, to improve the performance of hybrid fuzzing.

Table 12 Fuzzers that integrate different techniques

ID	Technique	Fuzzer	Open source	Year
1	Symbolic execution	Driller (Stephens et al. 2016)	✓	2016
		QSYM (Yun et al. 2018)	✓	2018
		SAVIOR (Chen et al. 2020b)		2020
		DigFuzz (Zhao et al. 2019)		2019
		PANGOLIN (Huang et al. 2020)		2020
		QuickFuzz (Lin et al. 2021)		2021
2	Parallel and integration	PAFL (Liang et al. 2018)		2018
		Cupid (Güler et al. 2020)	✓	2020
		EnFuzz (Chen et al. 2019b)	✓	2019
		AFL++ (Fioraldi et al. 2020)	✓	2020
3	Instrumentation	UnTracer (Nagy and Hicks 2019)	✓	2019
		Zeror (Zhou et al. 2020)		2020
4	Taint analysis	Angora (Chen and Chen 2018)	✓	2018
		Matryoshka(Chen et al. 2019a)		2019
		GreyOne (Gan et al. 2020)		2020
		PATA (Liang et al. 2022)		2022
5	Machine learning	NeuFuzz (Wang et al. 2019)		2019
		MEUZZ(Chen et al. 2020a)	✓	2020
		AFL++hier (Wang et al. 2021)	✓	2021
		COMFORT (Ye et al. 2021)	✓	2021
6	Intelligent optimization	Cerebro (Li et al. 2019)		2019
		OTA(Li et al. 2021)	✓	2021
		MobFuzz (Zhang et al. 2022)		2022
		MOPT (Lyu et al. 2019)	✓	2019
		MooFuzz (Zhao et al. 2021)		2021
7	Reinforcement learning	EcoFuzz (Yue et al. 2020)	✓	2020
		SLIME(Lyu et al. 2022)	✓	2022
		AMSFuzz (Zhao et al. 2022)		2022

3.3.2 Parallel and integration

Parallel and integration are also often used in fuzzing to improve the efficiency of fuzzing. PAFL (Liang et al. 2018) extends the existing single-mode fuzzing optimization to an industrial parallel mode using efficient guidance information synchronization and task partitioning, allowing multiple fuzzers to work in parallel. EnFuzz (Chen et al. 2019b) implements seed synchronization to improve the effectiveness of different fuzzers, integrating multiple fuzzing strategies to improve the performance and versatility of fuzzers. Cupid (Güler et al. 2020) collects and applies empirical data from a single isolated fuzzer and automatically identifies and selects a set of fuzzers that complement each other during collaborative execution for parallelized and distributed fuzzing. AFL++ (Fioraldi et al. 2020) integrates recent research results based on AFL, adding customized mutation APIs, achieving better fuzzing speed, and supporting customized modules.

3.3.3 Instrumentation

To reduce the overhead caused by coverage tracking, UnTracer (Nagy and Hicks 2019) adds interrupt instructions before each uncovered basic block of target programs to construct a new program. Each generated test case is sent to the new program to execute, and if an interrupt is triggered, the coverage tracking will be performed subsequently, otherwise, the test case is discarded. After coverage tracing, the previous basic block that has not been reached before will be determined, and the triggered interrupt instruction is deleted to avoid invalid tracing.

Reducing instrumentation overhead can improve the performance of fuzzing. Zeror (Zhou et al. 2020) uses two mechanisms to improve the performance of fuzzing, the self-modifying tracing mechanism and the binary switch scheduling mechanism. The former is used to maintain a set of unvisited instrumented points in the fuzzing process, and once an instrumented point is visited, the point would

be removed, reducing the overhead caused by instrumentation. The latter provides a multi-granularity binary switch scheduling using a Bayesian approach to switch the different coverage granularity instrumentation to better detect vulnerability.

3.3.4 Other techniques

In the research of general fuzzing and vulnerability-oriented fuzzing, fuzzing can also be combined with taint analysis, deep learning, machine learning, intelligent optimization, and other techniques to enhance the efficiency of fuzzing. The techniques has been discussed in Sects. 3.1 and 3.2.

3.3.5 Summary of fuzzing integration with other techniques

The integration of fuzzing with other techniques is one of the important directions in current research on fuzzing. Related techniques include symbolic execution, taint analysis, parallel and integration techniques, static analysis, intelligent optimization, deep learning and machine learning. Symbolic execution can obtain the input corresponding to a specific path by analyzing the program, but faces aces difficulties such as path explosion and inefficient solving. On the contrary, fuzzing can generate a large number of test cases to quickly cover more program paths, but it is difficult to ensure the efficiency of testing for specific paths. Existing research combines symbolic execution with fuzzing to take advantage of the strengths of each technique and improve overall testing efficiency. Taint analysis can trace sensitive data flows from the sources to sink, determine which inputs can influence the program's execution. Existing research combines taint analysis and fuzzing to determine mutation locations of seeds to generate high-quality test cases and improve testing efficiency. Parallel and integration are incorporated into fuzzing can save testing and enhance fuzzing efficiency. Other techniques, such as intelligent optimization, deep learning, and machine learning, are also combined with fuzzing to assist in program analysis, power schedule, test case generation, ultimately improving the efficiency of vulnerability discovery.

4 Fuzzing: different applications

In this section, we discuss the research progress of fuzzing in diverse applications, including SMT solver, virtual machine monitor, kernel, smart contract, protocol, and machine learning model. Table 13 shows the fuzzing of different applications.

4.1 SMT solver

Satisfiability modulo theories (SMT) solvers such as CVC4 (2021), Z3 (2015) are core and complex components of program analysis, which have widely been used in many applications, such as formal verification, security analysis, automated theorem proving, and symbolic execution. SMT solvers are used to evaluate the satisfiability of SMT instances. It is a complex system combining multiple decision procedures for various theories, such as uninterpreted functions, linear/nonlinear arithmetic, bit vectors, arrays strings, and others. It is difficult to find issues in the solvers and many works start to fuzz SMT solvers by generating SMT instances continuously. Fuzzers for SMT solver are shown in Table 14.

Stringfuzzer (Blotsky et al. 2018) is an open-source automated test string SMT solver fuzzer. It can generate and transform SMT instances with string or regular expression constraints, but the satisfiability of the instances generated by Stringfuzzer is unknown. A new string solver fuzzer, StringSolversTests (Bugariu and Müller 2020) is proposed which can construct satisfiable or unsatisfiable SMT instances with known satisfiability truth. These instances are used as inputs to fuzz SMT string solvers, to discover soundness and performance, completeness, and other bugs. YinYang (Winterer et al. 2020) is a mutation-based fuzzer that mutates a set of seed formulas, and then uses the mutated formulas as inputs to fuzz SMT solvers. The tool can detect soundness bugs, crashes, invalid model bugs, segmentation faults, and other bugs. STOROM (Mansur et al. 2020) is a blackbox mutation fuzzer. It first uses seed fragments to decompose formulas in seed instances into subformulas, then recombines these subformulas to generate new formulas, and finally the generated formulas are used to create a new, satisfying SMT instances to detect critical bugs in any SMT solver. A lightweight opfuzzer (Winterer et al. 2020) is proposed which uses type-aware operator mutation to generate test cases that meet the requirements and verify the results through different tests. BanditFuzz (Scott et al. 2020) uses reinforcement learning to automatically isolate and arrange grammatical structures in the input to explore the cause of errors or performance problems in the floating-point and string solvers.

4.2 Virtual machine monitor

Virtual machine monitor (VMM), also known as a hypervisor, is a core component of cloud computing and is used to virtual CPU, memory, I/O, and devices. Due to the diverse interfaces and complex architecture of virtual machines, as well as different interaction states, it makes direct fuzzing is not effective by using traditional fuzzing. Fuzzers for VMM are shown in Table 15.

Table 13 Fuzzing of different applications

ID	Application	Fuzzer
1	SMT solver	Stringfuzzer (Blotsky et al. 2018), StringSolversTests (Bugariu and Müller 2020), YinYang (Winterer et al. 2020), STORM (Mansur et al. 2020), opfuzzer (Winterer et al. 2020), BanditFuzz (Scott et al. 2020)
2	Virtual machine monitor	VDF (Henderson et al. 2017), HYPER-CUBE (Schumilo et al. 2020), NYX (Schumilo et al. 2021)
3	Kernel	Trinity (Jones (2010)), Syzkaller (Vyukov (2015)), TriforceAFL (Jesse (2015)), IMF (Han and Cha 2017), Moonshine (Pailoor et al. 2018), HFL (Kim et al. 2020), KAFL (Schumilo et al. 2017), PeriScope (Song et al. 2019), DIFUZE (Corina et al. 2017), Razzer (Jeong et al. 2019), Krace (Xu et al. 2020)
4	Smart contract	ContractFuzzer (Jiang et al. 2018), ILF (He et al. 2019), EthPloit (Zhang et al. 2020)
5	Protocol	AutoFuzz (Gorbunov and Rosenbloom 2010), SECFUZZ (Tsankov et al. 2012), PULSAR (Gascon et al. 2015), AFLNet (Pham et al. 2020), BLSTM-DCNNFuzz (Lv et al. 2020), SeqFuzzer (Zhao et al. 2019), Peach* (Luo et al. 2020)
6	Machine learning model	CAGFuzz (Zhang et al. 2022), DeepHunter (Xie et al. 2019), TensorFuzz (Odena et al. 2019), TitanFuzz (Deng et al. 2023)

Virtual devices are stateful and only work properly when properly initialized. They have specific behaviors during normal running. To fuzz virtual device, VDF (Henderson et al. 2017) uses the record and replay methods to detect virtual device bugs by first collecting the initialized and normal running behaviors of the device, then mutating and replaying the normal running behaviors. Jack and Li (2016) designed a framework for fuzzing virtual devices. It uses AFL to obtain coverage information and customizes lightweight customized BIOS to achieve portable and efficient fuzzing testing. Schumilo et al. (2020) presented a fuzzer for hypervisors, HYPER-CUBE, which can be applied to both open and closed source hypervisors. Based on a custom operating system, a custom bytecode interpreter is deployed which does not use coverage-guided fuzzing but is able to achieve high throughput, efficiency, and code coverage. NYX (Schumilo et al. 2021) is a coverage guidance hypervisor fuzzer. It uses instrumentation tool Inter Pin (Luk et al. 2005) to obtain coverage information and leverages a fast snapshot reload mechanism to obtain the virtual machine state generated by the previous test cases. To generate better test cases, a mutation engine based on bytecode programs is implemented that users can define a specification to generate inputs for different interfaces, and the concept of affine types is proposed to narrow down the space for test case generation.

4.3 Kernel

Kernel is an important and complex piece of system software for computers. Kernel security is usually a hot topic in some community forums. Many kernel fuzzers have been developed to test the kernel subsystem, such as file system, memory management, and device drivers and used to solve special problems and specific vulnerabilities in kernel. Fuzzers for kernel are shown in Table 16.

Trinity (Jones 2012) is a Linux system call fuzzer that uses the type information parameters provided in the Linux system call prototype definitions to passed parameter to the system calls drives the generation of test cases. However, Trinity does not keep track of coverage. Dmitry et al. Vyukov (2015) took this problem into account and developed an unsupervised coverage-guided kernel fuzzer, Syzkaller. Syzkaller uses the *gcc* port of the address sanitizer to keep track of coverage and supports other OS kernels as well. TriforceAFL Jesse (2015) used QEMU to implement code coverage and added serialization techniques to kernel APIs,

Table 14 Fuzzers for testing SMT solver

Fuzzer	Main contribution	Type	Tested target	Open source	Year
Stringfuzzer (Blotsky et al. 2018)	Testing string logic	Generation-based	String SMT solver	✓	2018
StringSolversTests (Bugariu and Müller 2020)	Automatic testing string SMT solver	Generation-based	String SMT solver	✓	2020
YinYang (Winterer et al. 2020)	Semantic fusion	Mutaiton-based	SMT solver	✓	2020
STORM (Mansur et al. 2020)	Mutation-based SMT solver testing	Mutation-based blockbox	SMT solver	✓	2020
opfuzzer (Winterer et al. 2020)	Type-aware operator mutation	Mutation-based	SMT solver		2020
BanditFuzz (Scott et al. 2020)	First machine learning-based fuzzer for SMT solvers	Mutation-based	Floating-point and string SMT solver		2020

and extended AFL to support fuzzing kernels. IMF (Han and Cha 2017) is a model-based API fuzzer that takes API calls context into account and generates random but well-structured seeds by inferring dependencies between individual APIs and combining with a mutation engine. KAFL (Schumilo et al. 2017) is a coverage-guided kernel fuzzing tool that utilizes one new feature in Intel CPU-provided hardware called Intel processor trace (PT). This hardware feature allows the CPU to gather branch tracing information to maximize code coverage within a limited time. Moonshine (Pailoor et al. 2018) implemented a seed distillation algorithm that uses static analysis to identify dependencies between function calls and then grouping them to generate seeds that ensuring code coverage.

The device drivers provide software interfaces to access hardware. Fuzzing device drive is concerned by researchers. Corina et al. (2017) proposed DIFUZE, an interface-aware fuzzing tool focusing on the *ioctls* interface provided by device drivers, which utilizes static analysis techniques to generate legal input sequences and track the execution of the driver. Song et al. (2019) presented PeriScope, a probing framework, suitable for detecting vulnerabilities that can be exploited by peripheral devices which lack underlying safeguards, in particular memory corruption bugs and double-fetch vulnerabilities, by analyzing the interaction between device drivers.

Aiming at kernel-specific challenges, Kim et al. (2020) designed a hybrid kernel fuzzer, HFL. It improves hybrid fuzzing to target the kernel's characteristics by shifting control transfer from implicit to explicit, performing inference of system call sequences to establish consistent system states, and identifying the nested parameter types of system calls. This ultimately improves the efficiency coverage of hybrid fuzzing. Considering the problem of data race in concurrency, Jeong et al. (2019) designed Razzer to achieve the efficient discovery of race in kernel using static analysis and deterministic thread interleaving techniques. Static analysis is used to analyze potential locations that exist data race in the source code. Deterministic thread interleaving is used to control thread schedule to provide accurate parallel execution information and reduce uncertainty. Xu et al. (2020) designed a coverage-based fuzzer, KRace, which replaces path coverage with alias coverage to achieve accurate data race detection.

4.4 Smart contract

Smart contracts have different characteristics compared to traditional applications, which presents a whole new challenge for the fuzzing of smart contracts. Firstly, the execution state of smart contract programs for different test cases is passed through the global *Storage* store and affects each other. Thus, it is difficult to improve global coverage by

Table 15 Fuzzers for testing VMM

Fuzzer	Main contribution	Tested target	Open source	Year
VDF (Henderson et al. 2017)	Using record and replay to fuzz virtual devices	Virtual device		2017
HYPER-CUBE (Schumilo et al. 2020)	Using custom interpreter for testing hypervisors	Hypervisor		2020
NYX (Schumilo et al. 2021)	Coverage guidance hypervisor fuzzer	Hypervisor	✓	2021

Table 16 Fuzzers for testing kernel

Fuzzer	Main contribution	Tested target	Open source	Year
Trinity (Jones (2010))	First system calls fuzzer	Kernel	✓	2010
Syzkaller (Vyukov (2015))	Coverage-based test case generation strategy	Kernel	✓	2015
TriforceAFL (Jesse (2015))	The AFL extension on the kernel testing	Kernel	✓	2015
IMF (Han and Cha 2017)	Fuzzer based on API dependency inference model	Kernel	✓	2017
KAFL (Schumilo et al. 2017)	Hardware design for using Intel CPUs to guided fuzzing	Kernel	✓	2017
Moonshine (Pailoor et al. 2018)	Implementation of seed distillation algorithm based on static analysis	Kernel	✓	2018
HFL (Kim et al. 2020)	First hybrid kernel fuzzer	Kernel		2020
DIFUZE (Corina et al. 2017)	Ioctl interface-aware fuzzing	Device driver	✓	2017
PeriScope (Song et al. 2019)	Detecting vulnerabilities by analyzing interactions between device drivers	Device driver	✓	2019
Razzer (Jeong et al. 2019)	Detecting race bugs through static analysis and thread interleaving	Kernel	✓	2019
Krace (Xu et al. 2020)	Detecting data race bugs using alias coverage	Kernel	✓	2020

providing coverage guidance feedback for test cases of individual functions. Secondly, the vulnerabilities may come from different levels of the blockchain, virtual machine, and high-level language, and there are many differences between them, which makes it challenging to detect vulnerabilities in smart contracts. Fuzzers for smart contract are shown in Table 17.

ContractFuzzer (Jiang et al. 2018) is a fuzzer that detects vulnerabilities in smart contracts. It instruments environment virtual machine (EVM) to log the runtime information of smart contracts and generates test cases that meet smart contracts grammars by learning the contract application binary interface specifications to fuzz smart contracts. Seven test oracles are defined to detect types of Ethereum smart contract vulnerabilities including gasless send, exception disorder, reentrancy, timestamp dependency, block number dependency, and dangerous delegatecall, and freezing ether vulnerabilities. ILF (He et al. 2019) is a neural network-based fuzzer for smart contracts, which aims at generating better test cases and transaction sequences. It uses a symbolic execution engine to generate a large number of transaction sequences and then trains a neural network model that captures a probabilistic fuzzing policy for generating test cases. To solve the problems of hard constraints in execution and ignoring blockchain properties on smart contracts, EthPloit (Zhang et al. 2020) generates transaction sequences to fuzz smart contracts by instrumenting EVM to better simulate blockchain behaviors and using dynamic seeding strategies to solve hard constraints.

4.5 Protocol

A protocol is a set of agreements that both sides of a communicating computer must mutually adhere to. The implementation of protocols ensures that services can be provided at a higher level. As the service has a large number of space states, it is necessary to traverse these states using a sequence of input messages, which also poses difficulties and challenges for testing protocols. There are many security issues in protocols that can easily cause serious problems, such as information leakage, denial of service, and others. Due to the diversity of protocols, the complexity of the protocol state space, and the dependency of protocol state, detecting protocol-related problems is a challenge. Fuzzers for protocol are shown in Table 18.

AutoFuzz (Gorbunov and Rosenbloom 2010) is an automated network protocol fuzzing framework that first constructs a finite state automaton (FSA) to capture communications between client and server to understand the implementation of protocols, and then learning the individual message syntax. Finally, AutoFuzz uses the FSA as a guide to fuzz the client and server protocols by modifying the input traffic. SECFUZZ (Tsankov et al. 2012) uses a modular fuzzing approach to fuzz stateful security protocols that handle encrypted traffic, and uses a series of custom mutation operators to generate test cases to detect security vulnerabilities. AFLNet (Pham et al. 2020) is a greybox fuzzer for protocols based on AFL. It uses code coverage feedback and state feedback to guide the fuzzing process.

Table 17 Fuzzers for testing smart contract

Fuzzer	Main contribution	Open source	Year
ContractFuzzer (Jiang et al. 2018)	Defining seven detection models	✓	2018
ILF (He et al. 2019)	Using neural network to fuzz smart contract	✓	2019
EthPloit (Zhang et al. 2020)	Using taint analysis and instrumentation to guide test case generation		2020

Table 18 Fuzzers for testing protocol

Fuzzer	Tested target	Open source	Year
AutoFuzz (Gorbunov and Rosenbloom 2010)	Network protocol		2010
SECFUZZ (Tsankov et al. 2012)	Security protocol	✓	2012
PULSAR (Gascon et al. 2015)	Network protocol	✓	2015
AFLNet (Pham et al. 2020)	Network protocol	✓	2020
SeqFuzzer (Zhao et al. 2019)	Industrial control protocol		2019
BLSTM-DCNNFuzz (Lv et al. 2020)	Industrial control protocol		2020
Peach*(Luo et al. 2020)	Industrial control protocol		2020

Aiming at industrial protocol, some fuzzing methods have been proposed. Zhao et al. (2019) proposed a protocol testing framework, SeqFuzzer, which trains sequence-to-sequence network models to automatically learn the frame structure of protocols to generate fake but plausible messages as test cases, sending them to perform and monitoring irregular industrial control system behaviors to find vulnerabilities. Lv et al. (2020) proposed an intelligent protocol framework, BLSTM-DCNNFuzz, which uses deep convolution generative adversarial networks to generate protocol messages to fuzz industrial control protocols. Luo et al. (2020) added coverage information to the traditional protocol fuzzer Peach, saving valuable packets and decomposing them to construct new high-quality test cases for future testing.

4.6 Machine learning model

With the development of artificial intelligence, more and more machine learning models are being used in various fields. Fuzzing techniques for machine learning models have attracted the attention of researchers. Fuzzers for machine learning model are shown in Table 19.

CAGFuzz (Zhang et al. 2022) is a coverage-guided adversarial generative fuzzing framework that uses the coverage of neurons as guide and trains an adversarial test case generator to generate as many adversarial test cases as possible under the condition of less disturbance. The generated test cases are suitable for different deep neural networks. DeepHunter (Xie et al. 2019) is coverage-guided fuzzing framework that combines five existing testing criteria including neuron coverage, k-multisection neuron coverage, neuron boundary coverage, strong neuron activation coverage, and top-k neuron coverage, to detect defects of deep neural networks. It uses a seed selection based on diversity and recency, and more fine-grained metamorphic mutation to generate test samples. That has great advantages in achieving high coverage and error detection capabilities. TensorFuzz (Odena et al. 2019) is a coverage-guided library for neural networks which is used to find numerical errors in the trained network, detect inconsistencies between models and quantized version, and display undesirable behavior in the character-level language model. It leverages fast approximate nearest neighbors algorithm to explore the activation function in neural networks as a coverage metric, and determines whether a new coverage is generated by detecting the similarity of activation vectors. Luo et al. (2021) proposed a graph-based fuzzing method which uses six different mutation strategies including graph edges addition, graph edges removal, block nodes addition, block nodes removal, tensor shape mutation, and parameters mutation to generate diversified digraph structure of deep learning models to fuzz deep learning inference engines. Monte Carlo tree search is used to search most promising mutation operators to generate new models. TitanFuzz (Deng et al. 2023) tests deep learning libraries with large language models (LLMs), which uses generative LLMs such as Codex by providing high-quality seed programs, and then uses filling LLMs such as InCoder to mutate the seed programs to generate high-quality seed inputs.

5 Conclusion and future direction

The development trends of fuzzing encompass automation, intelligence, technique integration, collaboration, diverse application domains, and open-source testing tools. Fuzzing efficiency and ease of use have improved due to automation, while intelligent techniques have enhanced test case generation and vulnerability discovery. Integration with other

Table 19 Fuzzers for testing machine learning model

Fuzzer	Main contribution	Tested target	Open source	Year
CAGFuzz (Zhang et al. 2022)	Coverage-guided adversarial generative fuzzing	Deep learning system		2019
DeepHunter (Xie et al. 2019)	Metamorphic mutation strategy	Deep neural network		2019
TensorFuzz (Odena et al. 2019)	Testing neural networks using coverage-guided fuzzing	Neural network	✓	2019
Luo et al. (2021)	Graph-based mutation to generate inputs	Deep learning inference engine	✓	2020
TitanFuzz (Deng et al. 2023)	Using LLMS to generate high-quality seed programs	Deep learning library		2023

techniques has bolstered testing effectiveness, and fuzzing has many applications in a wide range of domains. The availability of open-source fuzzers has stimulated active participation from developers and researchers. These trends underscore the significance of fuzzing as a critical technique in software and security testing. This paper provides a comprehensive review of common fuzzing processes and various types, highlighting the details of fuzzing techniques through the example of CGF and showcasing cutting-edge research. Furthermore, we discuss the diverse application areas of fuzz testing.

Future research directions can focus on the following areas.

(1) Learning-aware, smart fuzzing. Traditional approaches often primarily hinge on fixed mutation strategies or weighted test cases generated through grammatical analysis. However, smart fuzzing techniques will continuously collect and learn execution information from the target program. This deep understanding of program states will guide seed generation, improve code coverage, and enable real-time monitoring of program exceptions. By incorporating machine learning and other intelligent algorithms, smart fuzzing can adapt and optimize the fuzzing process based on continuous learning, leading to improved efficiency in vulnerability discovery.
(2) Vulnerability-aware fuzzing techniques. Despite significant advancements in CGF, there exists potential for improvement in identifying specific vulnerability types. Future fuzzing research will focus on developing vulnerability-aware techniques. These techniques will continuously capture and analyze characteristics and patterns of known vulnerabilities to guide the fuzzing process. By leveraging the information obtained from previous vulnerabilities, such as input patterns or triggers, fuzzing can be directed toward discovering similar vulnerabilities in different software systems. This strategy augments the vulnerability detection rate and facilitates targeted testing for distinct security weaknesses.
(3) Incorporating new techniques in fuzzing. New techniques can help improve the efficiency of fuzzing and the speed of discovery of vulnerabilities. New techniques, such as machine learning (Godefroid et al. 2017), reinforcement learning (Hou and Su 2022; Wang et al. 2021), intelligent optimization (Avci and Avci 2019; Wang et al. 2014; Wang and Tan 2019), LLMs (Deng et al. 2023; Wang et al. 2022), and parallel (Liang et al. 2018) techniques, are integrated into fuzzing to assist fuzzing to increase coverage and the speed of vulnerability discovery.
(4) Fuzzing in emerging applications. While fuzzing has been extensively applied in areas such as file fuzzing, protocol fuzzing, and kernel testing, there is a need to explore its applicability in emerging and complex applications. Future research will focus on developing customized fuzzing solutions tailored to specific applications, such as machine learning models, smart contracts, and IOT devices (D'Angelo et al. 2023, ?). These new and complex applications present unique challenges and require specialized fuzzing techniques to uncover vulnerabilities effectively.
(5) Works related to fuzzing. As fuzzing continues to gain popularity, related research areas will also evolve, such as anti-fuzzing techniques (Güler et al. 2019), exploit generation techniques (Heelan et al. 2019; Wang et al. 2018; You et al. 2017), evaluation methods, benchmarks, and metrics (Böhme and Falk 2020; Ding and Goues 2021; Li et al. 2021).

Funding This research was funded by the National Natural Science Foundation of China Grant Number 61827810.

Data availability Not applicable.

Code availability Not applicable.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

Abhishek A, Cris N (2012) Fuzzing for security. https://blog.chromium.org/2012/04/fuzzing-for-security.html. Accessed on 30 March 2021

Aschermann C, Schumilo S, Blazytko T, Gawlik R, Holz T (2019) REDQUEEN: fuzzing with input-to-state correspondence. In: Proceedings 2019 network and distributed system security symposium. https://doi.org/10.14722/ndss.2019.23371

Avci MG, Avci M (2019) An adaptive large neighborhood search approach for multiple traveling repairman problem with profits. Comput Oper Res 111:367–385. https://doi.org/10.1016/j.cor.2019.07.012

Avgerinos T, Rebert A, Cha SK, Brumley D (2014) Enhancing symbolic execution with veritesting. In: Proceedings of the 36th international conference on software engineering, pp 1083–1094. https://doi.org/10.1145/2568225.2568293

Baldoni R, Coppa E, D'elia DC, Demetrescu C, Finocchi I (2018) A survey of symbolic execution techniques. ACM Comput Surv (CSUR) 51(3):1–39

Banks G, Cova M, Felmetsger V, Almeroth K, Kemmerer R, Vigna G (2006) SNOOZE: toward a stateful network protocol fuzzer. In: International conference on information security, pp 343–358. https://doi.org/10.1007/11836810_25

Beaman C, Redbourne M, Mummery JD, Hakak S (2022) Fuzzing vulnerability discovery techniques: survey, challenges and future directions. Comput Secur 120:1–13. https://doi.org/10.1016/j.cose.2022.102813

Bekrar S, Bekrar C, Groz R, Mounier L (2012) A taint based approach for smart fuzzing. In: 2012 IEEE fifth international conference on software testing, verification and validation, pp 818–825. https://doi.org/10.1109/icst.2012.182

Blazytko T, Aschermann C, Schlögel M, Abbasi A, Schumilo S, Wörner S, Holz T (2019) GRIMOIRE: synthesizing structure while fuzzing. In: 28th USENIX security symposium, pp 1985–2002

Blotsky D, Mora F, Berzish M, Zheng Y, Kabir I, Ganesh V (2018) Stringfuzz: a fuzzer for string solvers. In: International conference on computer aided verification, pp 45–51. https://doi.org/10.1007/978-3-319-96142-2_6

Böhme M, Pham V, Roychoudhury A (2019) Coverage-based greybox fuzzing as Markov chain. IEEE Trans Softw Eng 45(5):489–506. https://doi.org/10.1109/tse.2017.2785841

Böhme M, Falk B (2020) Fuzzing: on the exponential cost of vulnerability discovery. In: Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering, pp 713–724. https://doi.org/10.1145/3368089.3409729

Böhme M, Pham VT, Nguyen MD, Roychoudhury A (2017) Directed greybox fuzzing. In: Proceedings of the 2017 ACM SIGSAC conference on computer and communications security, pp 2329–2344. https://doi.org/10.1145/3133956.3134020

Brad A (2009) Adobe reader and acrobat security initiative. https://blogs.adobe.com/security/2009/05/adobe_reader_and_acrobat_secur.html. Accessed on 30 March 2021

Brennan T, Saha S, Bultan T (2020) JVM fuzzing for JIT-induced side-channel detection. In: Proceedings of the ACM/IEEE 42nd international conference on software engineering, pp 1011–1023. https://doi.org/10.1145/3377811.3380432

Bugariu A, Müller P (2020) Automatically testing string solvers. In: Proceedings of the ACM/IEEE 42nd international conference on software engineering, pp 1459–1470. https://doi.org/10.1145/3377811.3380398

Chen Y, Ahmadi M, Farkhani RM, Wang B, Lu L (2020) MEUZZ: smart seed scheduling for hybrid fuzzing. In: International symposium on recent advances in intrusion detection, pp 77–92. https://doi.org/10.14722/ndss.2021.24486

Chen P, Chen H (2018) Angora: efficient fuzzing by principled search. In: 2018 IEEE symposium on security and privacy, pp 711–725. https://doi.org/10.1109/sp.2018.00046

Chen H, Guo S, Xue Y, Sui Y, Zhang C, Li Y, Wang H, Liu Y (2020) MUZZ: thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In: 29th USENIX security symposium, pp 2325–2342

Chen Y, Jiang Y, Ma F, Liang J, Wang M, Zhou C, Jiao X, Su Z (2019) EnFuzz: ensemble fuzzing with seed synchronization among diverse fuzzers. In: 28th USENIX security symposium, pp 1967–1983

Chen P, Liu J, Chen H (2019) Matryoshka: fuzzing deeply nested branches. In: Proceedings of the 2019 ACM SIGSAC conference on computer and communications security, pp 499–513. https://doi.org/10.1145/3319535.3363225

Chen Y, Li P, Xu J, Guo S, Zhou R, Zhang Y, Wei T, Lu L (2020) Savior: towards bug-driven hybrid testing. In: 2020 IEEE symposium on security and privacy, pp 1580–1596. https://doi.org/10.1109/sp40000.2020.00002

Chen H, Xue Y, Li Y, Chen B, Xie X, Wu X, Liu Y (2018) Hawkeye: towards a desired directed grey-box fuzzer. In: Proceedings of the 2018 ACM SIGSAC conference on computer and communications security, pp 2095–2108. https://doi.org/10.1145/3243734.3243849

Chris E, Matt M, Tavis O (2011) Fuzzing at scale. https://security.googleblog.com/2011/08/fuzzing-at-scale.html. Accessed on 30 March 2021

Cisco secure development lifecycle (2018). https://www.cisco.com/c/en/us/about/trust-center/technology-built-in-security.html#~processes. Accessed on 6 Aug 2023

Clang (2007). https://clang.llvm.org/. Accessed on 1 March 2021

Corina J, Machiry A, Salls C, Shoshitaishvili Y, Hao S, Kruegel C, Vigna G (2017) Difuze: interface aware fuzzing for kernel drivers. In: Proceedings of the 2017 ACM SIGSAC conference on computer and communications security, pp 2123–2138. https://doi.org/10.1145/3133956.3134069

CVC4 (2021). https://cvc4.github.io/. Accessed on 30 March 2021

CVE-fuzzing-poc (2016). https://github.com/geeknik/cve-fuzzing-poc. Accessed on 30 March 2021

D'Angelo G, Farsimadan E, Ficco M, Palmieri F, Robustelli A (2023) Privacy-preserving malware detection in android-based IoT devices through federated Markov chains. Futur Gener Comput Syst 148:93–105. https://doi.org/10.1016/j.future.2023.05.021

D'Angelo G, Ficco M, Robustelli A (2023) An association rules-based approach for anomaly detection on can-bus. In: International conference on computational science and its applications. Springer, pp 174–190

Darpa cyber grand challenge. https://www.darpa.mil/program/cyber-grand-challenge. Accessed on 6 Aug 2023

Deng Y, Xia CS, Peng H, Yang C, Zhang L (2023) Large language models are zero-shot fuzzers: fuzzing deep-learning libraries via large language models. In: Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis, pp 423–435

Ding ZY, Goues CL (2021) An empirical study of oss-fuzz bugs. arXiv preprint arXiv:2103.11518

Donaldson AF, Clayton B, Harrison R, Mohsin H, Neto D, Teliman V, Watson H (2023) Industrial deployment of compiler fuzzing techniques for two GPU shading languages. In: 2023 IEEE conference on software testing, verification and validation, pp 374–385. https://doi.org/10.1109/ICST57152.2023.00042

Dynamorio. https://github.com/DynamoRIO/dynamorio. Accessed on 30 March 2021

Edwards SH (2001) A framework for practical, automated black-box testing of component-based software. Softw Test Veri Reliab 11(2):97–111. https://doi.org/10.1002/stvr.224

Eisele M, Maugeri M, Shriwas R, Huth C, Bella G (2022) Embedded fuzzing: a review of challenges, tools, and solutions. Cybersecurity 5(1–18):18. https://doi.org/10.1186/s42400-022-00123-y

Fioraldi A, Maier D, Eißfeldt H, Heuse M (2020) AFL++ : combining incremental steps of fuzzing research. In: 14th USENIX workshop on offensive technologies, pp 1–12

Frida. https://frida.re/. Accessed on 30 March 2021

Fuzzdata (2015). https://github.com/MozillaSecurity/fuzzdata.git. Accessed on 30 March 2021

Ganesh V, Leek T, Rinard M (2009) Taint-based directed whitebox fuzzing. In: 2009 IEEE 31st international conference on software engineering, pp 474–484. https://doi.org/10.1109/icse.2009.5070546

Gan S, Zhang C, Chen P, Zhao B, Qin X, Wu D, Chen Z (2020) GREYONE: data flow sensitive fuzzing. In: 29th USENIX security symposium, pp 2577–2594

Gan S, Zhang C, Qin X, Tu X, Li K, Pei Z, Chen Z (2018) Collafl: path sensitive fuzzing. In: 2018 IEEE symposium on security and privacy, pp 679–696. https://doi.org/10.1109/sp.2018.00040

Gascon H, Wressnegger C, Yamaguchi F, Arp D, Rieck K (2015) Pulsar: stateful black-box fuzzing of proprietary network protocols. In: Security and privacy in communication networks: 11th EAI international conference, SecureComm 2015, Dallas, TX, USA, 26–29 Oct 2015, Proceedings 11. Springer, pp 330–347. https://doi.org/10.1007/978-3-319-28865-9_18

GDB (1988). https://www.gnu.org/software/gdb/. Accessed on 30 March 2021

Github. https://github.com/. Accessed on 6 Aug 2023

Godefroid P (2020) Fuzzing: hack, art, and science. Commun ACM 63(2):70–76. https://doi.org/10.1145/3363824

Godefroid P, Levin MY, Molnar DA (2008) Automated whitebox fuzz testing. Netw Distrib Secur Symp 8:151–166

Godefroid P, Kiezun A, Levin MY (2008) Grammar-based whitebox fuzzing. In: Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation, pp 206–215. https://doi.org/10.1145/1375581.1375607

Godefroid P, Kiezun A, Levin MY (2008) Grammar-based whitebox fuzzing. In: Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation, pp 206–215. https://doi.org/10.1145/1375581.1375607

Godefroid P, Peleg H, Singh R (2017) Learn&fuzz: machine learning for input fuzzing. In: 2017 32nd IEEE/ACM international conference on automated software engineering, pp 50–59. https://doi.org/10.1109/ase.2017.8115618

google: ClusterFuzz. https://github.com/google/clusterfuzz. Accessed on 30 March 2021

Gorbunov S, Rosenbloom A (2010) Autofuzz: automated network protocol fuzzing framework. Int J Comput Sci Netw Secur 10(8):239

Güler E, Aschermann C, Abbasi A, Holz T (2019) AntiFuzz: impeding fuzzing audits of binary executables. In: 28th USENIX security symposium, pp 1931–1947

Güler E, Görz P, Geretto E, Jemmett A, Österlund S, Bos H, Giuffrida C, Holz T (2020) Cupid: automatic fuzzer selection for collaborative fuzzing. In: Annual computer security applications conference, pp 360–372. https://doi.org/10.1145/3427228.3427266

Han H, Cha SK (2017) IMF: inferred model-based fuzzer. In: Proceedings of the 2017 ACM SIGSAC conference on computer and communications security, pp 2345–2358 https://doi.org/10.1145/3133956.3134103

Han W, Joe B, Lee B, Song C, Shin I (2018) Enhancing memory error detection for large-scale applications and fuzz testing. In: Proceedings 2018 network and distributed system security symposium. https://doi.org/10.14722/ndss.2018.23312

He J, Balunović M, Ambroladze N, Tsankov P, Vechev M (2019) Learning to fuzz from symbolic execution with application to smart contracts. In: Proceedings of the 2019 ACM SIGSAC conference on computer and communications security, pp 531–548. https://doi.org/10.1145/3319535.3363230

Heelan S, Melham T, Kroening D (2019) Gollum: modular and greybox exploit generation for heap overflows in interpreters. In: Proceedings of the 2019 ACM SIGSAC conference on computer and communications security, pp 1–18. https://doi.org/10.1145/3319535.3354224

Henderson A, Yin H, Jin G, Han H, Deng H (2017) VDF: targeted evolutionary fuzz testing of virtual devices. In: International symposium on research in attacks, intrusions, and defenses, pp 3–25. https://doi.org/10.1007/978-3-319-66332-6_1

HonggFuzz (2015). https://honggfuzz.dev/. Accessed on 30 March 2021

Hou L, Su Y (2022) Swarm activity-based dynamic PSO for distribution decision. Int J Autom Control 16(3/4):503–517. https://doi.org/10.1504/ijaac.2022.10046277

Huang H, Yao P, Wu R, Shi Q, Zhang C (2020) PANGOLIN: incremental hybrid fuzzing with polyhedral path abstraction. In: 2020 IEEE symposium on security and privacy, pp 1613–1627. https://doi.org/10.1109/sp40000.2020.00063

IDA (2003). https://www.hex-rays.com/products/ida/. Accessed on 30 March 2021

Jack T, Li M (2016) When virtualization encounter AFL. In: Black Hat Europe

Jain V, Rawat S, Giuffrida C, Bos H (2018) TIFF: using input type inference to improve fuzzing. In: Proceedings of the 34th annual computer security applications conference, pp 505–517. https://doi.org/10.1145/3274694.3274746

Jeong DR, Kim K, Shivakumar B, Lee B, Shin I (2019) Razzer: finding kernel race bugs through fuzzing. In: 2019 IEEE symposium on security and privacy, pp 754–768. https://doi.org/10.1109/sp.2019.00017

Jesse H. TriforceAFL. https://github.com/nccgroup/TriforceAFL. Accessed on 30 March 2021

Jiang B, Liu Y, Chan W (2018) ContractFuzzer: fuzzing smart contracts for vulnerability detection. In: 2018 33rd IEEE/ACM international conference on automated software engineering, pp 259–269. https://doi.org/10.1145/3238147.3238177

Jones D. trinity. https://github.com/kernelslacker/trinity. Accessed on 30 March 2021

Ju Y, Dong J, Chen S (2021) Recovering surface normal and arbitrary images: a dual regression network for photometric stereo. IEEE Trans Image Process 30:3676–3690. https://doi.org/10.1109/TIP.2021.3064230

Kim K, Jeong DR, Kim CH, Jang Y, Shin I, Lee B (2020) HFL: hybrid fuzzing on the Linux kernel. In: Proceedings of the 2020 annual network and distributed system security symposium, pp 1–17. https://doi.org/10.14722/ndss.2020.24018

Lemieux C, Padhye R, Sen K, Song D (2018) PerfFuzz: automatically generating pathological inputs. In: Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis, pp 254–265. https://doi.org/10.1145/3213846.3213874

Lemieux C, Sen K (2018) FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage. In: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering, pp 475–485. https://doi.org/10.1145/3238147.3238176

Li J, Zhao B, Zhang C (2018) Fuzzing: a survey. Cybersecurity 1(1):1–13. https://doi.org/10.1186/s42400-018-0002-y

Liang H, Pei X, Jia X, Shen W, Zhang J (2018) Fuzzing: state of the art. IEEE Trans Reliab 67(3):1199–1218. https://doi.org/10.1109/tr.2018.2834476

Liang H, Pei X, Jia X, Shen W, Zhang J (2018) Fuzzing: state of the art. IEEE Trans Reliab 67(3):1199–1218. https://doi.org/10.1145/3457913.3457934

Liang J, Jiang Y, Chen Y, Wang M, Zhou C, Sun J (2018) PAFL: extend fuzzing optimizations of single mode to industrial parallel mode. In: Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering, pp 809–814. https://doi.org/10.1145/3236024.3275525

Liang J, Wang M, Zhou C, Wu Z, Jiang Y, Liu J, Liu Z, Sun J (2022) PATA: fuzzing with path aware taint analysis. In: 2022 IEEE symposium on security and privacy, pp 1–17. https://doi.org/10.1109/sp46214.2022.9833594

Li Y, Chen B, Chandramohan M, Lin SW, Liu Y, Tiu A (2017) Steelix: program-state based binary fuzzing. In: Proceedings of the 2017 11th joint meeting on foundations of software engineering, pp 627–637. https://doi.org/10.1145/3106237.3106295

Li Y, Ji S, Chen Y, Liang S, Lee WH, Chen Y, Lyu C, Wu C, Beyah R, Cheng P, Lu K, Wang T (2021) UNIFUZZ: a holistic and pragmatic metrics-driven platform for evaluating fuzzers. In: 30th USENIX security symposium, pp 1–18

Lin P, Hong Z, Li Y, Wu L (2021) A priority based path searching method for improving hybrid fuzzing. Comput Secur 105:1–17. https://doi.org/10.1016/j.cose.2021.102242

Li X, Sun L, Qu H, Jang R, Yan Z (2021) OTA: an operation-oriented time allocation strategy for greybox fuzzing. In: 28th IEEE international conference on software analysis, evolution and reengineering, pp 108–118. https://doi.org/10.1109/saner50967.2021.00019

Liu C, Zou D, Luo P, Zhu BB, Jin H (2018) A heuristic framework to detect concurrency vulnerabilities. In: Proceedings of the 34th annual computer security applications conference, pp 529–541. https://doi.org/10.1145/3274694.3274718

Li Y, Xue Y, Chen H, Wu X, Zhang C, Xie X, Wang H, Liu Y (2019) Cerebro: context-aware adaptive fuzzing for effective vulnerability detection. In: Proceedings of the 2019 27th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering, pp 533–544. https://doi.org/10.1145/3338906.3338975

Lou B, Song J (2020) A study on using code coverage information extracted from binary to guide fuzzing. Int J Comput Sci Secur 14(5):200–210

Luk CK, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi VJ, Hazelwood K (2005) Pin: building customized program analysis tools with dynamic instrumentation. ACM SIGPLAN Not 40(6):190–200. https://doi.org/10.1145/1065010.1065034

Luo W, Chai D, Run X, Wang J, Fang C, Chen Z (2021) Graph-based fuzz testing for deep learning inference engines. In: Proceedings of the 43rd international conference on software engineering, pp 288–299. https://doi.org/10.1109/ICSE43902.2021.00037

Luo Z, Zuo F, Shen Y, Jiao X, Chang W, Jiang Y (2020) ICS protocol fuzzing: coverage guided packet crack and generation. In: 2020 57th ACM/IEEE design automation conference, pp 1–6. https://doi.org/10.1109/DAC18072.2020.9218603

Lv W, Xiong J, Shi J, Huang Y, Qin S (2020) A deep convolution generative adversarial networks based fuzzing framework for industry control protocols. J Intell Manuf 32:441–457. https://doi.org/10.1007/s10845-020-01584-z

Lyu C, Ji S, Zhang C, Li Y, Lee WH, Song Y, Beyah R (2019) MOPT: optimized mutation scheduling for fuzzers. In: 28th USENIX security symposium, pp 1949–1966

Lyu C, Liang H, Ji S, Zhang X, Zhao B, Han M, Li Y, Wang Z, Wang W, Beyah R (2022) SLIME: program-sensitive energy allocation for fuzzing. In: Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis, pp 365–377. https://doi.org/10.1145/3533767.3534385

Manès VJ, Han H, Han C, Cha SK, Egele M, Schwartz EJ, Woo M (2019) The art, science, and engineering of fuzzing: a survey. IEEE Trans Softw Eng 47(11):2312–2331. https://doi.org/10.1109/tse.2019.2946563

Mansur MN, Christakis M, Wüstholz V, Zhang F (2020) Detecting critical bugs in SMT solvers using blackbox mutational fuzzing. In: Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering, pp 701–712. https://doi.org/10.1145/3368089.3409763

Max M, Kostya S (2016) Guided in-process fuzzing of Chrome components. https://security.googleblog.com/2016/08/guided-in-process-fuzzing-of-chrome.html. Accessed on 30 March 2021

Miller BP, Koski D, Lee CP, Maganty V, Murthy R, Natarajan A, Steidl J (1995) Fuzz Revisited: A re-examination of the reliability of UNIX utilities and services. Comput Sci Dept, University of Wisconsin. 1–23

Nagy S, Hicks M (2019) Full-speed fuzzing: reducing fuzzing overhead through coverage-guided tracing. In: 2019 IEEE symposium on security and privacy, pp 787–802. https://doi.org/10.1109/sp.2019.00069

Neystadt J (2008) Automated penetration testing with white-box fuzzing. Microsoft, February

Nguyen MD, Bardin S, Bonichon R, Groz R, Lemerre M (2020) Binary-level directed fuzzing for use-after-free vulnerabilities. In: 23rd International symposium on research in attacks, intrusions and defenses, pp 47–62

Nilizadeh S, Noller Y, Păsăreanu CS (2019) DifFuzz: Differential fuzzing for side-channel analysis. In: Proceedings of the 41st international conference on software engineering, pp 176–187. https://doi.org/10.1109/ICSE.2019.00034

Noller Y, Kersten R, Păsăreanu CS (2018) Badger: complexity analysis with fuzzing and symbolic execution. In: Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis, pp 322–332. https://doi.org/10.1145/3213846.3213868

Odena A, Olsson C, Andersen D, Goodfellow I (2019) TensorFuzz: debugging neural networks with coverage-guided fuzzing. In: International conference on machine learning, pp 4901–4911

OllyDbg (2000). http://domoticx.com/windows-debugger-ollydbg-software/. Accessed on 30 March 2021

Onefuzz (2020). https://github.com/microsoft/onefuzz. Accessed on 23 March 2021

Pailoor S, Aday A, Jana S (2018) MoonShine: optimizing OS fuzzer seed selection with trace distillation. In: 27th USENIX security symposium, pp 729–743

PaiMei. https://github.com/OpenRCE/https://github.com/OpenRCE/paimei. Accessed on 30 March 2021

Peng H, Shoshitaishvili Y, Payer M (2018) T-Fuzz: fuzzing by program transformation. In: 2018 IEEE symposium on security and privacy, pp 697–710. https://doi.org/10.1109/SP.2018.00056

Petsios T, Zhao J, Keromytis AD, Jana S (2017) SlowFuzz: automated domain-independent detection of algorithmic complexity vulnerabilities. In: Proceedings of the 2017 ACM SIGSAC conference on computer and communications security, pp 2155–2168. https://doi.org/10.1145/3133956.3134073

Pham VT, Böhme M, Roychoudhury A (2016) Model-based white-box fuzzing for program binaries. In: Proceedings of the 31st IEEE/ACM international conference on automated software engineering, pp 543–553. https://doi.org/10.1145/2970276.2970316

Pham VT, Böhme M, Roychoudhury A (2020) AFLNet: a greybox fuzzer for network protocols. In: 2020 IEEE 13th international conference on software testing, validation and verification, pp 460–465. https://doi.org/10.1109/icst46399.2020.00062

Pham VT, Böhme M, Santosa AE, Caciulescu AR, Roychoudhury A (2019) Smart greybox fuzzing. IEEE Trans Softw Eng. https://doi.org/10.1109/TSE.2019.2941681

Rawat S, Jain V, Kumar A, Cojocar L, Giuffrida C, Bos H (2017) VUzzer: application-aware evolutionary fuzzing. In: 24th Annual network and distributed system security symposium, pp 1–14. https://doi.org/10.14722/ndss.2017.23404

Saavedra GJ, Rodhouse KN, Dunlavy DM, Kegelmeyer PW (2019) A review of machine learning applications in fuzzing, pp 1–12. arXiv preprint arXiv:1906.11133

Schumilo S, Aschermann C, Abbasi A, Worner S, Holz T (2020) HYPER-CUBE: high-dimensional hypervisor fuzzing. In: 27th Annual network and distributed system security symposium, pp 23–26. https://doi.org/10.14722/ndss.2020.23096

Schumilo S, Aschermann C, Abbasi A, Wörner S, Holz T (2021) NYX: greybox hypervisor fuzzing using fast snapshots and affine types. In: 30th USENIX security symposium

Schumilo S, Aschermann C, Gawlik R, Schinzel S, Holz T (2017) kAFL: hardware-assisted feedback fuzzing for OS kernels. In: 26th USENIX security symposium, pp 167–182

Scott J, Mora F, Ganesh V (2020) Banditfuzz: a reinforcement-learning based performance fuzzer for SMT solvers. In: Software verification: 12th international conference, VSTTE 2020, and 13th international workshop, pp 68–86. https://doi.org/10.1007/978-3-030-63618-0_5

Serebryany K (2016) Continuous fuzzing with libFuzzer and AddressSanitizer. In: 2016 IEEE cybersecurity development, pp 157–157. https://doi.org/10.1109/secdev.2016.043

Serebryany K (2017) OSS-Fuzz—Google's continuous fuzzing service for open source software. In: 26th USENIX security symposium, pp 1–28

She D, Shah A, Jana S (2022) Effective seed scheduling for fuzzing with graph centrality analysis. In: 2022 IEEE symposium on security and privacy, pp 2194–2211. https://doi.org/10.1109/sp46214.2022.9833761

Situ LY, Zuo ZQ, Guan L, Wang LZ, Li XD, Shi J, Liu P (2021) Vulnerable region-aware greybox fuzzing. J Comput Sci Technol 36:1212–1228. https://doi.org/10.1007/s11390-021-1196-0

Song D, Hetzelt F, Das D, Spensky C, Na Y, Volckaert S, Vigna G, Kruegel C, Seifert JP, Franz M (2019) PeriScope: an effective probing and fuzzing framework for the hardware-OS boundary. In: Proceedings 2019 network and distributed system security symposium, pp 1–15. https://doi.org/10.14722/ndss.2019.23176

Stephens N, Grosen J, Salls C, Dutcher A, Wang R, Corbetta J, Shoshitaishvili Y, Kruegel C, Vigna, G (2016) Driller: augmenting fuzzing through selective symbolic execution. In: 23rd Annual network and distributed system security symposium, pp 1–16. https://doi.org/10.14722/ndss.2016.23368

Sun L, Li X, Qu H, Zhang X (2020) AFLTurbo: speed up path discovery for greybox fuzzing. In: 2020 IEEE 31st international symposium on software reliability engineering, pp 81–91. https://doi.org/10.1109/issre5003.2020.00017

Sutton M, Greene A, Amini P (2007) Fuzzing: brute force vulnerability discovery. Pearson Education, London

Takanen A, Demott JD, Miller C, Kettunen A (2018) Fuzzing for software security testing and quality assurance. Artech House, Norwood

The home for Sanitizers (2019). https://github.com/google/sanitizers. Accessed on 30 March 2021

ThreadSanitizer (2019). https://clang.llvm.org/docs/ThreadSanitizer.html. Accessed on 30 March 2021

Trickel E, Pagani F, Zhu C, Dresel L, Vigna G, Kruegel C, Wang R, Bao T, Shoshitaishvili Y, Doupé A (2023) Toss a fault to your witcher: applying grey-box coverage-guided mutational fuzzing to detect SQL and command injection vulnerabilities. In: 2023 IEEE symposium on security and privacy (SP), pp 2658–2675. https://doi.org/10.1109/sp46215.2023.10179317

Tsankov P, Dashti MT, Basin D (2012) SECFUZZ: fuzz-testing security protocols. In: 2012 7th international workshop on automation of software test, pp 1–7. https://doi.org/10.1109/iwast.2012.6228985

UndefinedBehaviorSanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html. Accessed on 30 March 2021

Viide J, Helin A, Laakso M, Pietikäinen P, Seppänen M, Halunen K, Puuperä R, Röning J (2008) Experiences with model inference assisted fuzzing. In: 2nd USENIX workshop on offensive technologies, vol 2, pp 1–2

Vinesh N, Rawat S, Bos H, Giuffrida C, Sethumadhavan M (2020) Confuzz—a concurrency fuzzer. In: 1st International conference on sustainable technologies for computational intelligence-proceedings of ICTSCI 2019, pp 667–691. https://doi.org/10.1007/978-981-15-0029-9_53

Vyukov D. Syzkaller. https://github.com/google/syzkaller. Accessed on 30 March 2021

Wang, J, Chen B, Wei L, Liu Y (2019) Superion: grammar-aware greybox fuzzing. In: 2019 IEEE/ACM 41st international conference on software engineering, pp 724–735. https://doi.org/10.1109/icse.2019.00081

Wang GG, Tan Y (2019) Improving metaheuristic algorithms with information feedback models. IEEE Trans Cybern 49(2):542–555. https://doi.org/10.1109/TCYB.2017.2780274

Wang GG, Guo L, Gandomi AH, Hao GS, Wang H (2014) Chaotic krill herd algorithm. Inf Sci 274:17–34. https://doi.org/10.1016/j.ins.2014.02.123

Wang Y, Wu Z, Wei Q, Wang Q (2019) NeuFuzz: efficient fuzzing with deep neural network. IEEE Access 7:36340–36352. https://doi.org/10.1109/access.2019.2903291

Wang Y, Jia P, Liu L, Huang C, Liu Z (2020) A systematic review of fuzzing based on machine learning techniques. PLoS ONE 15(8):1–20. https://doi.org/10.1371/journal.pone.0237749

Wang L, Pan Z, Wang J (2021) A review of reinforcement learning based intelligent optimization for manufacturing scheduling. Complex Syst Model Simul 1(4):257–270. https://doi.org/10.23919/CSMS.2021.0027

Wang X, Hu C, Ma R, Tian D, He J (2021) CMFuzz: context-aware adaptive mutation for fuzzers. Empir Softw Eng 26(1):1–34. https://doi.org/10.1007/s10664-020-09927-3

Wang F, Wang X, Sun S (2022) A reinforcement learning level-based particle swarm optimization algorithm for large-scale optimization. Inf Sci 602:298–312

Wang J, Chen B, Wei L, Liu Y (2017) Skyfire: data-driven seed generation for fuzzing. In: 2017 IEEE symposium on security and privacy, pp 579–594. https://doi.org/10.1109/SP.2017.23

Wang Y, Jia X, Liu Y, Zeng K, Bao T, Wu D, Su P (2020) Not all coverage measurements are equal: fuzzing by coverage accounting for input prioritization. In: 27th Annual network and distributed system security symposium, pp 1–17. https://doi.org/10.14722/ndss.2020.24422

Wang Z, Liblit B, Reps T (2020) TOFU: target-orienter fuzzer. arXiv preprint arXiv:2004.14375

Wang F, Shoshitaishvili Y (2017) Angr—the next generation of binary analysis. In: 2017 IEEE cybersecurity development, pp 8–9. https://doi.org/10.1109/SecDev.2017.14

Wang J, Song C, Yin H (2021) Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing. In: Network and distributed system security symposium, pp 1–17. https://doi.org/10.14722/ndss.2021.24486

Wang H, Xie X, Li Y, Wen C, Li Y, Liu Y, Qin S, Chen H, Sui Y (2020) Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In: 42nd International conference on software engineering, pp 999–1010. https://doi.org/10.1145/3377811.3380386

Wang Y, Zhang C, Xiang X, Zhao Z, Li W, Gong X, Liu B, Chen K, Zou W (2018) Revery: From proof-of-concept to exploitable. In: Proceedings of the 2018 ACM SIGSAC conference on computer and communications security, pp 1914—1927. https://doi.org/10.1145/3243734.3243847

Wang P, Zhou X, Lu K, Yue T, Liu Y (2020) Sok: the progress, challenges, and perspectives of directed greybox fuzzing. In: Challenges, and perspectives of directed greybox fuzzing

Wen C, Wang H, Li Y, Qin S, Liu Y, Xu Z, Chen H, Xie X, Pu G, Liu T (2020) MemLock: memory usage guided fuzzing. In: 42nd International conference on software engineering, pp 765–777 . https://doi.org/10.1145/3377811.3380396

Winterer D, Zhang C, Su Z (2020) On the unusual effectiveness of type-aware operator mutations for testing SMT solvers. Proc ACM Program Lang 4:1–25. https://doi.org/10.1145/3428261

Winterer D, Zhang C, Su Z (2020) Validating SMT solvers via semantic fusion. In: Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation, pp 718–730. https://doi.org/10.1145/3385412.3385985

Woo M, Cha SK, Gottlieb S, Brumley D (2013) Scheduling black-box mutational fuzzing. In: Proceedings of the 2013 ACM SIGSAC conference on computer and communications security, pp 511–522. https://doi.org/10.1145/2508859.2516736

Xie X, Ma L, Juefei-Xu F, Xue M, Chen H, Liu Y, Zhao J, Li B, Yin J, See S (2019) DeepHunter: a coverage-guided fuzz testing framework for deep neural networks. In: Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis, pp 146–157. https://doi.org/10.1021/acs.jcim.8b00542.s002

Xu M, Kashyap S, Zhao H, Kim T (2020) Krace: data race fuzzing for kernel file systems. In: 2020 IEEE symposium on security and privacy, pp 1643–1660. https://doi.org/10.1109/sp40000.2020.00078

Ye G, Tang Z, Tan SH, Huang S, Fang D, Sun X, Bian L, Wang H, Wang Z (2021) Automated conformance testing for JavaScript engines via deep compiler fuzzing. In: 42nd ACM SIGPLAN conference on programming language design and implementation, pp 435–450

You W, Liu X, Ma S, Perry D, Zhang X, Liang B (2019) SLF: fuzzing without valid seed inputs. In: 2019 IEEE/ACM 41st international conference on software engineering, pp 712–723. https://doi.org/10.1109/icse.2019.00080

You W, Wang X, Ma S, Huang J, Zhang X, Wang X, Liang B (2019) ProFuzzer: on-the-fly input type probing for better zero-day vulnerability discovery. In: 2019 IEEE symposium on security and privacy, pp 769–786. https://doi.org/10.1109/sp.2019.00057

You W, Zong P, Chen K, Wang X, Liao X, Bian P, Liang B (2017) SemFuzz: semantics-based automatic generation of proof-of-concept exploits. In: Proceedings of the 2017 ACM SIGSAC conference on computer and communications security, pp 2139–2154. https://doi.org/10.1145/3133956.3134085

Yue T, Wang P, Tang Y, Wang E, Yu B, Lu K, Zhou X (2020) EcoFuzz: adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In: 29th USENIX security symposium, pp 2307–2324

Yun I, Lee S, Xu M, Jang Y, Kim T (2018) QSYM: a practical concolic execution engine tailored for hybrid fuzzing. In: 27th USENIX security symposium, pp 745–761

Z3 (2015). https://en.wikipedia.org/wiki/Z3_Theorem_Prover. Accessed on 30 March 2021

Zhang G, Wang PF, Yue T, Kong XD, Zhou X, Lu K (2022) ovAFLow: detecting memory corruption bugs with fuzzing-based taint inference. J Comput Sci Technol 37(2):405–422. https://doi.org/10.1007/s11390-021-1600-9

Zhang P, Ren B, Dong H, Dai Q (2022) CAGFuzz: coverage-guided adversarial generative fuzzing testing for image-based deep learning systems. IEEE Trans Softw Eng 48(11):4630–4646. https://doi.org/10.1109/TSE.2021.3124006

Zhang Q, Wang Y, Li J, Ma S (2020) Ethploit: from fuzzing to efficient exploit generation against smart contracts. In: 2020 IEEE 27th international conference on software analysis, evolution and reengineering, pp 116–126. https://doi.org/10.1109/SANER48275.2020.9054822

Zhang G, Wang P, Yue T, Kong X, Huang S, Zhou X, Lu K (2022) MobFuzz: adaptive multi-objective optimization in gray-box fuzzing. In: Network and distributed systems security symposium 2022, pp 1–18. https://doi.org/10.14722/ndss.2022.24314

Zhang Y, Wang Z, Yu W, Fang B (2021) Multi-level directed fuzzing for detecting use-after-free vulnerabilities. In: 2021 IEEE 20th international conference on trust, security and privacy in computing and communications, pp 569–576. https://doi.org/10.1109/trustcom53373.2021.00087

Zhao X, Qu H, Lv W, Li S, Xu J (2021) MooFuzz: many-objective optimization seed schedule for fuzzer. Mathematics 9:1–19. https://doi.org/10.3390/math9030205

Zhao X, Qu H, Xu J, Li S, Wang GG (2022) AMSFuzz: an adaptive mutation schedule for fuzzing. Expert Syst Appl 208:1–11. https://doi.org/10.1016/j.eswa.2022.118162

Zhao L, Duan Y, Yin H, Xuan J (2019) Send hardest problems my way: probabilistic path prioritization for hybrid fuzzing. In: Proceedings 2019 network and distributed system security symposium. https://doi.org/10.14722/ndss.2019.23504

Zhao H, Li Z, Wei H, Shi J, Huang Y (2019) SeqFuzzer: an industrial protocol fuzzing framework from a deep learning perspective. In: 2019 12th IEEE conference on software testing, validation and verification, pp 59–67. https://doi.org/10.1109/ICST.2019.00016

Zhou C, Wang M, Liang J, Liu Z, Jiang Y (2020) Zeror: speed up fuzzing with coverage-sensitive tracing and scheduling. In: 2020 35th IEEE/ACM international conference on automated software engineering, pp 858–870. https://doi.org/10.1145/3324884.3416572

Zlewski C. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl. Accessed on 1 March 2021