Computers, Materials & Continua

Tech Science Press

DOI: 10.32604/cmc.2023.042361

REVIEW

Fuzzing: Progress, Challenges, and Perspectives

Zhenhua Yu[1], Zhengqi Liu[1], Xuya Cong[1,*], Xiaobo Li[2] and Li Yin[3]

[1]Institute of Systems Security and Control, College of Computer Science and Technology, Xi'an University of Science and Technology, Xi'an, 710054, China

[2]School of Mathematics and Information Science, Baoji University of Arts and Sciences, Baoji, 721013, China

[3]Institute of Systems Engineering, Macau University of Science and Technology, Taipa, Macau, China

*Corresponding Author: Xuya Cong. Email: congxuya@xust.edu.cn

Received: 28 May 2023 Accepted: 16 October 2023 Published: 30 January 2024

ABSTRACT

As one of the most effective techniques for finding software vulnerabilities, fuzzing has become a hot topic in software security. It feeds potentially syntactically or semantically malformed test data to a target program to mine vulnerabilities and crash the system. In recent years, considerable efforts have been dedicated by researchers and practitioners towards improving fuzzing, so there are more and more methods and forms, which make it difficult to have a comprehensive understanding of the technique. This paper conducts a thorough survey of fuzzing, focusing on its general process, classification, common application scenarios, and some state-of-the-art techniques that have been introduced to improve its performance. Finally, this paper puts forward key research challenges and proposes possible future research directions that may provide new insights for researchers.

KEYWORDS

Fuzzing; vulnerability; software testing; software security

1 Introduction

As information technology evolves, software has infiltrated into more and more aspects of modern life. The booming development of software not only changes people's lives but also brings several software security problems. On February 20, 2020, Apache Tomcat [1] exposed a high-risk server file read/inclusion vulnerability (CVE-2020–1938 [2]), which caused the disclosure of sensitive data such as important configuration files and source code in web app directories on Tomcat. Under a specified condition, one can further implement remote code execution to directly control the server. If the vulnerability is exploited, it could affect millions of Tomcat servers running around the world, and the potential damage could be incalculable [3,4].

Currently, there is no broad consensus on the definition of software vulnerability. In accordance with [5,6], in this paper, software vulnerability refers to the defects or errors produced in software development and operations, and triggered under specific conditions, which can cause serious damage to the operating platform of the software, affect the normal operation of the system, and even threaten the privacy and property of users. As a key problem in software security, effective vulnerability mining

has become a research hotspot. In general, vulnerability mining technologies are mainly divided into static and dynamic analysis, as shown in Fig. 1. Static analysis does not require running a program, and its commonly used methods include static symbolic execution, data flow analysis, model checking, abstract interpretation, etc. Dynamic analysis is carried out based on actual program execution, and the methods mainly include dynamic symbol execution and fuzzing (also called fuzz testing). Static analysis technology relies heavily on prior knowledge and has some defects such as low accuracy and high false-positive rate. It has been noted that path explosion is the main problem faced by both static symbolic execution and dynamic symbolic execution. Path explosion, also known as state explosion, refers to a situation where the number of paths grows exponentially with the increase of branching conditions in the program running. Due to the existence of the path explosion problem, symbol execution is difficult to handle in large and complex applications. By comparison, fuzzing has many advantages, such as low system consumption, a low false-positive rate, and a high degree of automation. In simple terms, fuzzing mines vulnerabilities by sending malicious data to the target program and monitoring its abnormal behaviors [7].

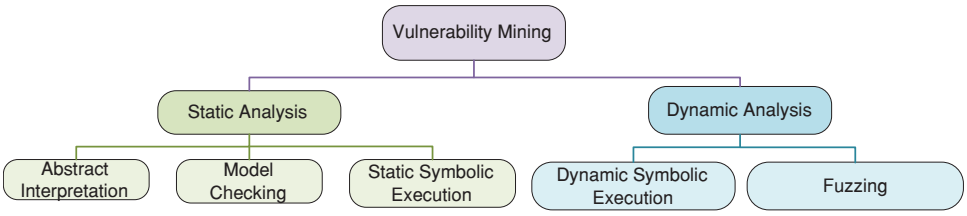

Figure 1: Common vulnerability mining techniques

In 1990, fuzzing first emerged and was used for software testing of UNIX systems to test the robustness of various UNIX applications [8]. If the target program crashes or hangs when it is fed random strings, the tool would analyze the input that causes the exception. Although fuzzing was just a typical black-box testing method at that time, it was still effective. As fuzzing evolves, some state-of-the-art techniques are introduced by researchers to carry out vulnerability mining, and the application scenarios of fuzzing are also expanding constantly. Simple but effective, fuzzing has become a testing method that mines the largest number of software vulnerabilities encountered so far. Many software manufacturers carry out fuzzing before the software is released to discover potential vulnerabilities.

Fuzzing has experienced rapid development in recent years. However, the existing reviews of fuzzing mainly focus on certain aspects, which are not conducive to a comprehensive understanding of fuzzing. To this end, a thorough overview of fuzzing is presented in this paper. We introduce the principle and general process of fuzzing, and provide an overview of the research progress of fuzzing in recent years. The key techniques applied to fuzzing are also introduced. Furthermore, the common application scenarios of fuzzing are presented, including industrial control protocols, web applications, kernels, machine learning models, and smart contracts. At last, the challenges faced by fuzzing are discussed and the future research directions are proposed. The overall contributions of this paper are summarized in Table 1.

The remainder of this paper is organized as follows. The paper starts with the background knowledge about fuzzing in Section 2. Section 3 introduces the related works that have a significant impact on the development of fuzzing. Section 4 systematically reviews the key techniques of fuzzing. Section 5 presents the common application scenarios of fuzzing. Section 6 discusses the challenges faced by fuzzing techniques and sheds light on directions for future research. The conclusion of the paper is presented in Section 7.

Table 1: Overall contributions of this paper

Section	Corresponding content
Working process	Seed file generation, test case generation, bug detection, ...
Evaluation metrics	Coverage, unique code path, pass rate, ...
Key techniques	Symbolic execution, coverage feedback, taint analysis, ...
Application scenarios	Industrial control protocol, web application, OS kernel, ...
Challenges	(1) More application scenarios (2) More complex software (3) The absence of datasets (4) Coverage guidance hits a bottleneck
Future directions	(1) Building one-stop cluster fuzzing platforms (2) Combining deep learning with fuzzing (3) Constructing vulnerability datasets (4) Combining program analysis with fuzzing

2 Background

2.1 Classification

As shown in Fig. 2, fuzzing can be classified into different kinds from different perspectives. Depending on the knowledge extent of the internal structure of the target program, fuzzing can be classified into the black box, grey box, and white box [9]. The program under testing is considered a black-box whose internal details are unknowable when testers perform black-box fuzzing. Testers input test cases into the program under testing, observe output results, and complete the test without knowing how the program works. Different from black-box fuzzing, white-box fuzzing can be thought of as being done in a transparent box whose internal details are known. The details include source code, runtime information, and design specifications, which can help testers to improve the fuzzing efficiency. Grey-box fuzzing is a kind of fuzzing between white-box fuzzing and black-box fuzzing.

Based on test case generation, fuzzing can be divided into mutation-based fuzzing and generation-based fuzzing. The former generates test cases by modifying valid inputs, and the latter constructs grammatically correct test cases by utilizing the rule definition file. Comparatively, the generation-based method can pass verification easier so that it can improve code coverage effectively.

Depending on whether the runtime information is employed to guide the test case generation or not, fuzzing can be classified into feedback fuzzing and no-feedback fuzzing. The former receives feedback from the runtime information (e.g., code coverage information) to improve the fuzzing efficiency as much as possible, while the latter does not collect any information during fuzzing, which saves time but reduces efficiency. Based on the applications of exploration strategies, fuzzing can be divided into directed fuzzing and coverage-guided fuzzing. Directed fuzzer generates test cases to explore a particular set of execution paths while coverage-guide fuzzer is designed to maximize code coverage to discover hidden vulnerabilities.

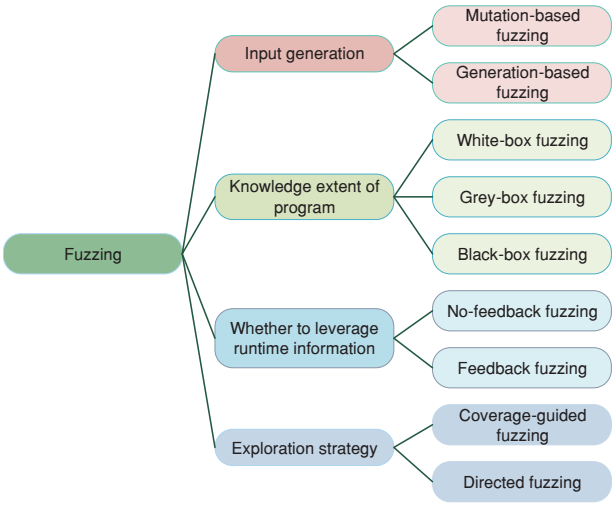

Figure 2: The categories of fuzzing

2.2 Working Process

The working process of fuzzing is depicted in Fig. 3, which consists of seed file generation, test case generation, test case execution, bug detection, and exploitability analysis. Based on the processes, researchers add some additional processes to improve fuzzing efficiency, such as seed filter and test case filter, which greatly improve the fuzzing results. The whole process of fuzzing is described in detail in the following part.

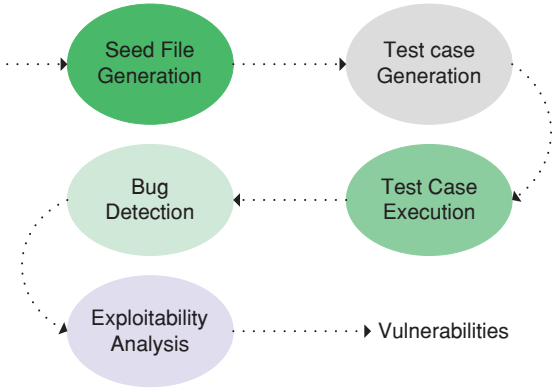

Figure 3: Working process of fuzzing

Seed file [10] is one of the critical factors affecting fuzzing performance. The sources of the seed file are web crawlers, self-build, etc. For example, testers usually construct seed files using capture tools that can capture the real protocol messages when performing fuzzing on the industrial protocol. Seed filters can select seeds to enhance path traversal and vulnerability mining. The testers typically consider execution speed, path frequency, path depth, and untraversed path branches when the seeds are selected.

Test cases are generated using two approaches: mutation-based and generation-based approaches [11]. The former mutates the seed files to obtain many test cases by executing mutation operators

including adding, modifying, removing, and so on. The latter can generate highly structured test cases by taking the input format into account. However, a large quantity of generated test cases cannot trigger bugs, and it is a waste of time to execute these test cases. To improve the fuzzing efficiency, some testers execute a test case filter before test case execution to filter test cases with a higher probability of triggering vulnerabilities.

Test cases are then input into the program under testing. A monitor can output the runtime information to guide the test case generation. When the program under testing crashes or reports some errors, the relevant information will be gathered for exploitability analysis. The exceptional information gathered when test cases are executed will be analyzed to identify if the crash is a bug. Exploitability analysis helps to identify if it is a vulnerability [12].

2.3 Evaluation

The metrics to evaluate the performance of fuzzing are coverage, unique code path, unique crash or bug, pass rate, etc. The following part presents a detailed introduction to these metrics.

Coverage is the most widely used metric for evaluating fuzzing performance, which includes basic block, function, line, instruction, and branch coverage. V-Fuzz [13] is compared with VUzzer [14] in terms of the basic block coverage as both utilize the same approach to evaluate the code coverage. Godefroid et al. [15] measured the instruction coverage to evaluate the fuzzing effectiveness of Samplefuzz. Wang et al. utilized samples gathered from a three-month crawling and the highly-structured inputs generated by Skyfire [16] as seeds of American fuzzy lop (AFL) [17] to test several open-source extensible stylesheet language transformations and extensible markup language (XML) engines. They employ line coverage and function coverage as evaluation metrics of code coverage. She et al. [18] compared the fuzzers on a 24-h fixed running time budget leveraging the sum of new edges and the speed of edge coverage to evaluate the fuzzing performance.

The unique code path presents the number of code paths explored or triggered during fuzzing. In [19], SmartSeed can trigger more unique paths compared with other seed strategies. SmartSeed + AFL discovers 30.7% more unique paths than the existing strategies, which shows that SmartSeed yields much better performance compared with state-of-the-art seed selection strategies. A unique crash or bug is the number of unique crashes and bugs discovered during fuzzing, which is a suitable evaluation criterion. In terms of the number of unique crashes, SmartSeed is compared with other seed strategies for discovering unique crashes, and is found to have great efficiency and performs better than other seed strategies.

The pass rate presents the proportion of test cases that can pass the syntax check of the program under testing. In Skyfire, only 34% of the XML inputs generated by the approach based on context-free grammar (CFG) could pass the semantic checking, while 85% and 63% of the XSL and XML inputs generated by Skyfire that learns a probabilistic context-sensitive grammar (PCSG) can pass the semantic checking and reach the application execution stage. It shows that PCSG is more useful than CFG for generating inputs satisfying semantic checking.

3 Timeline

Since its introduction in 1990, fuzzing has been one of the most widely-adopted techniques for testing the correctness and reliability of software. Fig. 4 shows the evolution of fuzzing over the past 30 years. In this section, we present the related works that have a significant impact on the development of fuzzing.

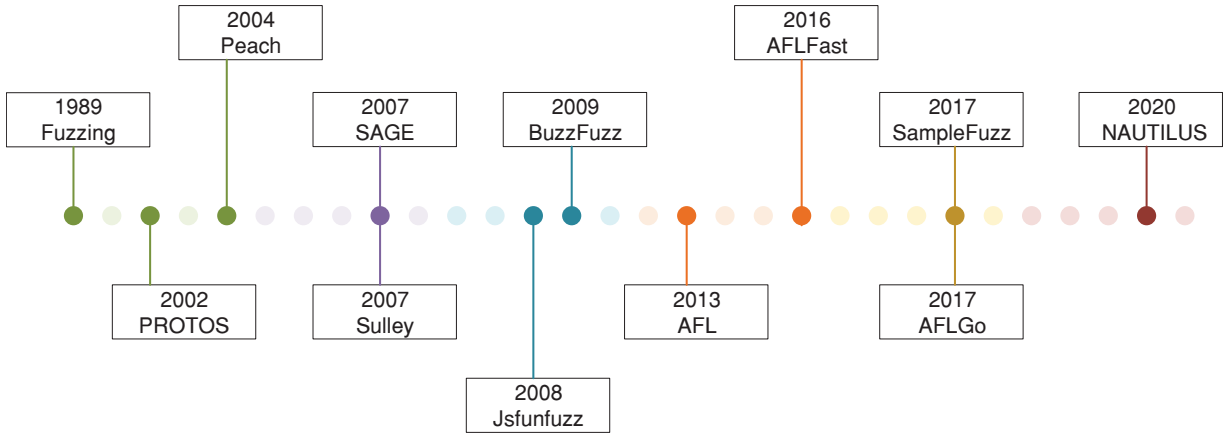

Figure 4: The evolution of fuzzing

Professor Barton Miller is regarded as the father of fuzzing [20]. In 1990, he first applied fuzz testing to test the robustness of UNIX applications. He uses pure black-box testing, sending randomly generated strings or characters to the application under testing to see if it crashes or hangs. In 1999, OUSPG (Oulu University Secure Programming Group) began to work on the PROTOS project [21] and published the PROTOS test toolset in 2002. The toolset constructs malformed packets according to protocol specification documents and so on, which can be used to test network protocols. A large number of vulnerabilities are found in the later tests.

In 2004, Michael Eddington proposed a cross-platform testing framework and released the tool Peach [22]. It supports both generation-based and mutation-based fuzzing methods. The generation-based fuzzing method can generate random data and fill a given model, and then generate malformed data. The mutation-based fuzzing method can be modified based on a given sample and produce malformed data. It is a flexible framework that maximizes code reuse. Sulley [23] is a fuzzing framework including multiple extensible components. This framework aims to simplify not only data representation but also data transmission and instrumentation. Sulley adapts a block-based data generation approach that simplifies the process for testers to design test cases for complex protocols. It also has a good scalability. However, it requires certain professional knowledge of testers and relies on manual parsing of the protocol format.

In 2007, scalable automated guided execution (SAGE) [24], a white-box fuzzer using symbol execution was born and benefitted from advances in dynamic symbol execution and test data generation technology. By using the symbolic execution techniques, the constraints of conditional statements on input are collected during the program running and used to generate new inputs. The white-box fuzz testing has the advantage of fully understanding the internal situation of the target, which can help get higher coverage and deep vulnerability detection effect.

In 2008, JSFunfuzz [25] was born to carry out fuzzing on targets with highly structured input. This method generates random but grammatically correct JavaScript code according to the syntax model. The idea of generating input based on syntax is proposed. It becomes an important guideline for fuzzing targets with highly structured input. In 2009, Buzzfuzz [26] replaced the costly symbol execution technique with taint analysis. A taint analysis technique relates the value of a variable used in a program's attack points to a specific part of the input data to guide the mutation strategy of fuzz testing. This idea has also been used for reference in grey-box fuzzing.

AFL is an excellent coverage-guided fuzzing tool. It obtains runtime information through code instrumentation. When testing an open-source application, the instrumentation is carried out at compile-time, while for a binary program, the instrumentation is performed at runtime utilizing a modified quick emulator. There is more chance of being selected in the next mutation for test cases exploring new execution paths. AFL is a powerful tool to mine vulnerabilities in real-world applications such as common image parsing [27], file compression libraries, etc.

Bohme et al. [28] proposed AFLFast which is a coverage-based grey box fuzzer. It is observed that the majority of test cases focus on the same few paths. For example, in a portable network graphics processing program, a large portion of the test cases obtained by random mutation are invalid, and the error handling path is triggered. Thus, paths are divided into low-frequent and high-frequent by AFLFast. During fuzzing, it measures the frequency of paths, prioritizes seeds with less mutation, and assigns extra energy to seeds exploring low-frequent paths.

AFLGo [29] is a directed grey box fuzzer, which sets some fragile codes as target locations and gives priority to test cases closer to target locations. It can assign more testing resources to fragile codes by employing an appropriately directed algorithm. However, AFLGo strongly depends on the results of static analysis tools. Currently, static analysis tools still have a high false positive rate and cannot provide accurate verification. Samplefuzz is the first attempt to automatically produce input syntax from samples employing neural network-based learning techniques. A generation model for automatically learning portable document format (PDF) objects based on the recursive neural network is presented and evaluated. A great many new useful objects can be obtained by this model, and the coverage is also improved.

Traditional mutation-based fuzzers, such as AFL, often require a corpus crawled from the Internet. However, these corpora usually contain only the common grammar of these languages, and it is difficult to trigger rare bugs. The generation-based fuzzer can generate strong structural input based on a certain template or syntax and has achieved a good effect on testing language interpreters. By comparing the two types of fuzzers and combining the two ideas, NAUTILUS [30] combines grammar with code coverage feedback, which not only improves the effectiveness of fuzzing but also increases the code coverage.

4 Key Techniques

Due to the advent of fuzzing, the techniques have gained increasing applications in software testing. An increasing number of researchers and software security practitioners have introduced various techniques to solve different problems of fuzzing, which has significantly improved its performance. In this section, we systematically introduce these key techniques.

4.1 Traditional Techniques

In this section, we show how these techniques work in fuzzers. Table 2 summarizes the fuzzers based on traditional techniques, including symbolic execution, coverage feedback, taint analysis, and so on.

Table 2: Summary of fuzzers based on traditional techniques

Reference	Key techniques	Type	Target	Source/Binary
[14]	Dynamic taint analysis symbolic execution Data flow analysis	Whitebox	Linux software	Source
[17]	Coverage feedback	Whitebox	Applications	Source/Binary
[24]	Symbolic execution	Whitebox	Large windows applications	Binary
[26]	Dynamic taint analysis Coverage feedback	Greybox	Linux software	Binary
[28]	Markov chain	Greybox	Applications	Source/Binary
[29]	Simulated annealing	Greybox	Applications	Source/Binary
[30]	Grammar Coverage feedback	Greybox	Programming language interpreters	Source
[31]	Selective concolic execution	Whitebox	Applications	Binary
[32]	Symbolic execution	Whitebox	Closed source targets	Binary
[33]	Symbolic execution	Whitebox	Linux software	Binary
[34]	Coverage feedback	Greybox	Applications	Source
[35]	Coverage feedback	Greybox	Applications	Binary
[36]	Dynamic taint analysis	Whitebox	Applications	Source
[37]	Dynamic taint analysis Symbolic execution	Whitebox	Linux/Windows software	Binary
[38]	Grammar, symbolic execution	Whitebox	IE7 JavaScript interpreter	Source
[39]	Grammar	Blackbox	Interpreter	Binary
[40]	Grammar Coverage feedback	Greybox	XML and JavaScript engines	Binary
[41]	Model	Blackbox	Microsoft server network instant messaging protocol	Binary
[42]	Model, Grammar	Greybox	XML processor	Binary
[43]	Model	Greybox	Smart TV system	Binary
[44]	Particle swarm optimization	Greybox	Applications	Source/Binary
[45]	Adversarial multi-armed bandit model	Greybox	Applications	Source/Binary
[46]	Ant colony optimization	Greybox	Applications	Source

4.1.1 Symbolic Execution

Symbolic execution [47] was first proposed in 1976. Over the past few decades, it has evolved several executions from classical execution to dynamic symbolic execution and selective symbolic execution [31,48,49]. Symbolic execution substitutes program inputs with symbolic values and replaces

corresponding concrete program operations with the ones that manipulate symbolic values [50]. In 2005, the concept of dynamic symbolic execution was proposed by Godefroid et al. [51]. In 2011, Chipounov et al. [52] proposed the selective symbolic execution technique to restrict the range of symbol state differentiation in the code of "interest", which avoids unnecessary testing of other codes. Many researchers [31–33] have introduced symbolic execution into fuzzing to improve the fuzzing process.

Microsoft's SAGE starts with well-formed inputs and symbolically executes the program to exercise all possible execution paths. The input constraints on the program trace are collected to generate new inputs that execute different paths of the program. Over 20 new bugs have been discovered in real-world Windows applications, including file decoders, media players, and image processors. However, it remains imperfect for the symbolic execution accuracy and constraint-solving capability.

Fuzzing and concolic execution are leveraged by Driller [31] in a complementary way to efficiently detect bugs buried in a binary program. Fuzzing is employed to execute the compartments of an application, while concolic execution is utilized to produce inputs that pass the complex checks separating the compartments. This ensures that large-scale programs can be effectively analyzed. However, test cases are processed as generic inputs in one component and specific inputs in another, which leads to frequent calls to concolic execution, causing path explosions.

Böttinger et al. [32] combined the dynamic symbolic execution with fuzzing. The concolic execution is interleaved with constrained fuzzing in a way that only paths providing the maximum input generation frequency can be explored. Vulnerabilities in deep layers can be triggered in this method. This method cannot be directed towards a specific location, as paths are built as deep as possible through the program without direction.

SmartFuzz [33] concentrates on integer bugs. The symbolic execution is used to construct fuzzing data that can trigger non-value-preserving width conversions, arithmetic overflows, or signed/unsigned conversions. It outputs constraints as "close" as possible to the intermediate representation, with only limited optimizations in the run. The main downside of this choice is that although each optimized query is small, the total symbolic trace including all queries in a program can be several gigabytes.

4.1.2 Coverage Feedback

Intuitively, code coverage is closely related to bug coverage, and a fuzzer having higher code coverage could help to discover more bugs [53]. Coverage-guided fuzzing is the most common fuzzing technique, which is designed to maximize coverage to mine hidden vulnerabilities. Coverage information is often obtained by using program instrumentation. It can be used to choose which generated inputs to keep for fuzzing. If some inputs increase code coverage, they will be retained and fuzzed in a continuous loop.

The coverage feedback allows the fuzzer to gradually reach deeper into the code and disclose more vulnerabilities in a restricted time. Typical coverage-guided fuzzers include AFL, Libfuzzer [54], OSSFuzz [55], Honggfuzz [56], etc. These tools have mined plenty of undisclosed vulnerabilities in real-world software. The fuzz testing method based on coverage feedback is the main research direction of fuzzing at present.

The basic flow of AFL is shown in Fig. 5. AFL has been remarkably useful for exposing bugs in real-world software. However, it still has many flaws. The AFL employs a hash function and a hash bitmap to record program execution paths, which improves the efficiency of code execution. Due to the lack of a hash conflict solution, the statistics in AFL code coverage are not accurate, which affects

the subsequent process of test case selection and mutation. Although AFL solves the blindness of selection in mutation seed files to a certain extent, the mutation process is still blind when it is carried out for a certain seed file. Without effective guidance for the mutation to be executed in the desired direction, the probability of effective mutation is not high, which makes it difficult for AFL to explore deep vulnerabilities.

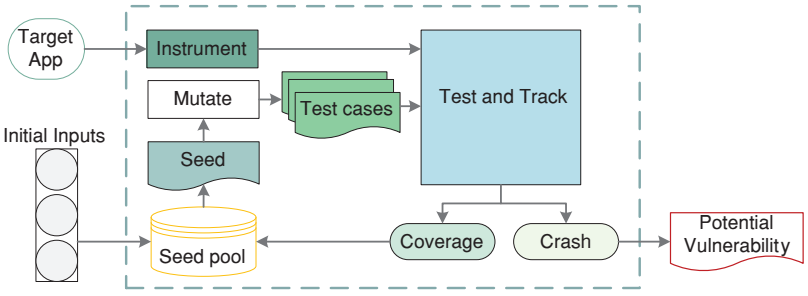

Figure 5: The basic flow of AFL

UnTracer [34] applies coverage-guided tracing to eliminate needless tracing. The start of each basic block previously uncovered is overwritten with a software interrupt signal. A test case is traced if it causes the interruption. The performance of fuzzing is improved by limiting the tracing cost to only test cases that increase coverage. However, not all the inputs that yield new coverage should be treated equally. The new code executed by some inputs is less likely to be vulnerable. The coverage cannot accurately describe the state of a program, nor can it effectively distinguish the importance of different paths.

TortoiseFuzz [35] introduces an innovative approach to input prioritization, leveraging a novel technique known as coverage accounting. This method assesses program state across three distinct granularities: basic blocks, loops, and function calls. By amalgamating the evaluation and coverage information, TortoiseFuzz significantly enhances its capacity to unearth vulnerabilities related to program memory. This advancement represents a notable stride forward in the realm of fuzz testing methodologies.

4.1.3 Taint Analysis

In 1976, Denning introduced the pioneering concept of the information flow model, offering a structured approach for formalizing the intricate processes of data processing and transmission within software programs [57]. It is this seminal work that paved the way for the development of taint analysis, a technique that draws heavily from the principles of the information flow theory. Fig. 6 shows an intuitive example of the taint analysis process.

According to whether the program needs to be run during the analysis process, taint analysis can be divided into static taint analysis [58] and dynamic taint analysis [59]. Dynamic taint analysis (DTA) is a prevalent and useful technique to track the information flow in software without accessing the source code [60], which runs a program and monitors which computations are influenced by predefined taint sources (e.g., user input) [61]. With the emergence of dynamic program instrumentation platforms, such as PIN [62], DynamoRIO [63], and Valgrind [64], dynamic taint analysis has a significant effect on dynamic program analysis. Many researchers [14,26,36,37] introduced dynamic taint analysis into fuzzing to guide the generation of test cases.

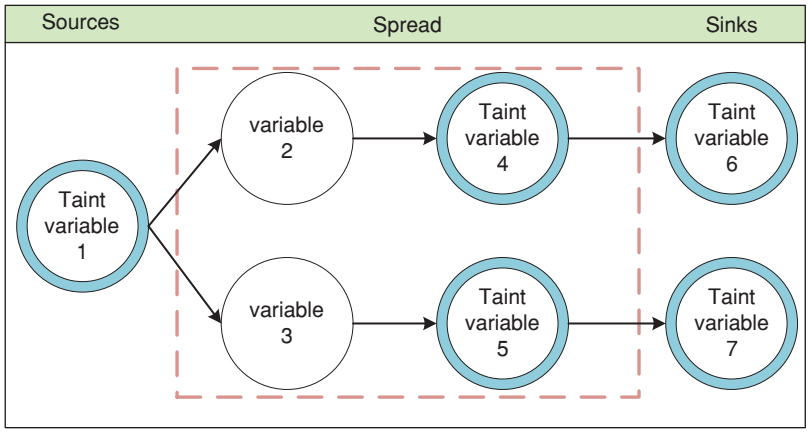

Figure 6: An intuitive example of the process of taint analysis

The core of VUzzer [14] is the dynamic taint analysis, which plays a key role in evolving new inputs. Information obtained during the taint analysis stage will be fed back to the input selection stage to improve the performance of input selection and mutation strategies. VUzzer is designed to concentrate more on generating inputs that can reach deeper paths. However, codes in deeper paths do not mean that they are more likely to be vulnerable.

BuzzFuzz [26] uses DTA to obtain taint information, which can be used to guide the generation of test cases targeting specific attack points without violating their syntactic validity. This approach effectively enhances the quality of test cases. It necessitates access to the source code, which might be unavailable or inaccessible in certain scenarios. As a result, the applicability of BuzzFuzz could be constrained by the availability of source code resources.

The checksum mechanism is commonly used in network protocols and many file formats. Traditional fuzzers are ineffective when there is a checksum mechanism to verify the integrity of inputs in the program under testing. TaintScope [36] is a directed fuzzing system designed to bypass checksum checks. It can generate inputs with a higher probability of triggering potential vulnerabilities utilizing fine-grained dynamic taint tracing that leads to a dramatic improvement in the fuzzing effectiveness.

Program analysis, taint tracking, and symbolic execution are combined in Dowser [37] to mine buffer overflow vulnerabilities. It finds the accesses that are most likely to cause buffer overflows by program analysis. The inputs influencing these accesses are discovered by taint analysis, which then makes only this set of inputs symbolic. This systematic integration of program analysis, taint tracking, and symbolic execution in Dowser facilitates the exploration of deep vulnerabilities, thereby extending its utility in detecting intricate software bugs.

4.1.4 Grammar

Well-formed inputs are required when we test the applications with highly structured input. Grammar-based fuzzing finds vulnerabilities by inputting well-formed inputs according to rules that encode application semantics [65].

Godefroid et al. [38] enhanced white-box fuzzing via a grammar-based specification of valid inputs. Constraint solving and symbolic execution are combined in their test case generation method. This combined strategy not only enhances the generation of diverse and meaningful test cases but also

contributes to uncovering complex vulnerabilities that may not be easily detected using traditional fuzzing methods.

LangFuzz [39] is a framework allowing black-box fuzzing of engines based on context-free grammar. Fed with appropriate grammar, it can be easily adapted to a new language. Fed with an appropriate set of test cases to mutate and extend, it can be easily adapted to new projects. This unique approach to black-box fuzzing through grammar-based input generation contributes to the generation of more focused and impactful test inputs, ultimately aiding in the identification of potential vulnerabilities and defects within software systems.

Superion [40] was proposed to complement mutation and grammar-blind trimming strategies in AFL. Each test input is parsed into an abstract syntax tree (AST) based on the grammar of the test inputs. Then, a grammar-aware trimming strategy is introduced, which iteratively removes each subtree in the AST of the test input and observes coverage differences. It also contains two grammar-aware mutation strategies, which improve code coverage and bug discovery ability in AFL.

The grammar and code coverage feedback are used in NAUTILUS [30] to effectively test programs with highly structured input. The combination of coverage feedback and grammar-based splicing increases the effectiveness of fuzzing. However, the source code and grammar of an application are needed. This prerequisite may limit the applicability of NAUTILUS in scenarios where access to such information is restricted or unavailable.

4.1.5 Model

The basic idea of model-based fuzzing is that instead of generating test cases manually, selected algorithms automatically generate test cases from a (set of) model (s) of the system under testing or of its environment [66]. Hsu et al. [41] proposed a model-based methodology employing an automatically synthesized formal protocol behavioral model to guide input selection. Their experiments with Microsoft server network instant messaging clients prove the effectiveness of this methodology. However, the lack of testing feedback may influence fuzzing efficiency.

BlendFuzz [42] is a model-based framework for the effective fuzzing of programs with grammatical inputs. It uses grammar to model program inputs, which enables test cases to pass the initial paring stage of the test program. The possibility of uncovering bugs and security vulnerabilities is added by constructing mutated test cases which may reach intractable corner cases. By combining grammar-based modeling and mutation techniques, BlendFuzz establishes an effective strategy for maximizing the efficiency of the fuzzing process and deepening the exploration of potential program vulnerabilities.

Gebizli et al. [43] proposed a method to refine models employed for model-based fuzzing by taking advantage of the principles of risk-based testing. Markov Chains are utilized as system models in which state transition probabilities decide the generation of test cases. The memory leaks obtained during the previous test execution are utilized to update probability values. Promising results are observed in the context of an industrial case study.

4.1.6 Optimization Algorithms

Scheduling algorithms aim to improve the fuzzing performance with an optimized seed selection strategy and seed mutation strategy [67]. In fuzzing, many researchers consider seed scheduling, mutation scheduling, etc., as optimization problems. Numerous algorithms such as Simulated Annealing (SA) [68], Genetic Algorithms (GA) [69], Particle Swarm Optimization (PSO) [70], and Ant Colony

Optimization (ACO) [71] are utilized as scheduling algorithms to improve efficiency. Table 3 shows the optimization solutions for different fuzzing processes.

Table 3: Optimization solutions for different fuzzing processes

Paper	Year	Algorithm	Process
[28]	2016	Markov chain	Seed scheduling
[29]	2017	Simulated annealing	Seed scheduling
[44]	2019	Particle swarm Optimization	Mutation scheduling
[45]	2020	Multi-armed bandit	Seed scheduling
[46]	2021	Ant colony Optimization	Mutation scheduling

AFLFast models coverage-based grey box fuzzing as a Markov chain. Then, a monotonous power schedule is implemented to allocate energy. This can quickly come close to the minimum energy needed to find a new path. However, AFLFast cannot be adapted flexibly to the allocation strategy by the fuzz process, which raises the average energy cost of finding a new path. In addition, while AFLFast presents the transition probability in fuzzing and identifies the method for assigning energy based on the transition probability, it is not capable of providing a detailed analysis of the transition probability.

AFLGo is an AFL-based fuzzer leveraging a simulated annealing power schedule to generate test cases that can efficiently reach the target program location. The simulated annealing power schedule progressively allocates more energy to seeds close to the target location and less energy to seeds far from the target location. Compared with non-directed fuzzers, AFLGo has better results in testing the given target programs. However, it prefers the path closer to the target, which may lead to missing some potential vulnerabilities.

In [44], the authors considered mutation scheduling as an optimization problem, and a novel mutation scheduling scheme was proposed. A customized Particle Swarm Optimization algorithm was utilized by the scheme in [44] to obtain the optimal selection probability distribution of operators for improving the fuzzing effect, which can be widely utilized for the existing mutation-based fuzzers.

EcoFuzz [45] utilizes a variant of the Adversarial Multi-Armed Bandit (VAMAB) model to model coverage-based grey box fuzzing. An adaptive average-cost-based power schedule is also employed, which can effectively reduce the energy cost and achieve maximum coverage of fuzzing in a limited time. The integration of the VAMAB model and the adaptive power schedule amplifies the potential of EcoFuzz in efficiently exploring diverse execution paths, thereby enhancing the overall effectiveness of the fuzzing process.

In [46], regression grey box fuzzing (RGF) was proposed, which fuzzes the code that has changed more recently or more often. The key innovation of RGF lies in its ability to adaptively prioritize mutated bytes based on their potential to produce impactful test inputs. By intelligently allocating resources to promising bytes, RGF achieves a more efficient exploration of the input space, leading to improved code coverage and vulnerability discovery. This approach represents a significant advancement in grey box fuzzing techniques and demonstrates promising results in practical use cases.

4.2 Smart Techniques

As machine learning has evolved in cybersecurity, it has also been applied to many studies in fuzzing. Table 4 lists the smart techniques for different steps of fuzzing testing. In this section, we introduce these smart techniques.

Table 4: Summary of smart techniques for different steps of fuzzing testing

Paper	Birth year	Algorithm	Application
[15]	2017	Recurrent neural networks (RNN)	Testcase generation
[19]	2018	GAN	Testcase generation
[25]	2017	Generative adversarial network (GAN)	Testcase generation
[40]	2019	Long short-term memory (LSTM)	Seed file generation
[72]	2019	LSTM	Seed file generation
[73]	2019	LSTM	Testcase generation
[74]	2019	Wasserstein generative adversarial networks (WGAN)	Testcase generation
[75]	2020	Convolutional neural network (CNN)	Testcase filter
[76]	2018	RL	Mutation operator selection
[77]	2010	Reinforcement learning (RL)	Mutation operator selection
[78]	2018	RL	Mutation operator selection
[79]	2018	RL	Mutation operator selection

4.2.1 Applications of Deep Learning in Fuzzing

Deep learning [80,81], which includes RNN [82], LSTM [83] network, GAN [84], CNN [85] and so on, has made great progress. In cybersecurity, deep learning has been applied to fuzzing by a growing number of researchers, which provides a new way to address difficult problems in previous research.

(1) Recurrent Neural Network

Compared with ordinary neural networks, RNN is good at mining and utilizing temporal and semantic information in data. It is mainly used to process and predict sequence data. The characteristic of sequence data is that the data are correlated before and after. In an ordinary neural network, the nodes between layers are connected, while the nodes in the same network layer are not related to each other. Therefore, it cannot well express the relevance of nodes in the same layer, that is, the relevance of sequence data cannot be well represented. RNN is a feed-forward neural network with a time connection such that it has both state and time connections between channels. The input information of a neuron includes not only the output of the previous nerve cell layer but also its state in the previous channel.

In fuzzing, RNN has been studied for test case generation. Samplefuzz utilizes the Char-RNN model to learn a generative model. Given a starting prefix, new sequences (PDF objects) can be generated by sampling the distribution using the model. The model not only generates a large number of new well-formed objects but also improves code coverage, compared with various forms of random fuzzing. The complex test case generation is transferred into a language modeling task in [86]. The neural language model (NLM) based on deep RNNs are utilized to learn the structure of complex

input files. Two specific fuzzing algorithms, MetadataNeuralFuzz and DataNeuralFuzz, are proposed. The former focuses on the parsing stage, while the latter aims at the rendering stage of input file processing. The proposed method achieves significant improvements in code coverage and accuracy.

(2) Long Short-Term Memory Network

LSTM is a variant of the traditional recurrent neural network that can effectively tackle the problem of long-term dependence. LSTM adds a "gate" based on the RNN model, through which it can control whether the previous information is forgotten or remembered, to solve the gradient explosion problem caused by long text. LSTM is used to perform seed file generation and test case generation in fuzzing.

Wang et al. [72] made use of an LSTM model to learn the hidden pattern of vulnerable program paths and identify vulnerable paths to guide seed selection. More paths that are more likely to be vulnerable can be explored by these seeds selected. SeqFuzzer [73] can automatically learn frame structures according to communication traffic and generate spurious but plausible messages as fuzzing data. It uses three-layer deep LSTMs in the seq2seq model. An encoder LSTM model learns the syntax of the protocol data and a decoder LSTM model predicts the corresponding receiving protocol sequence. Rajpal et al. [87] implemented sequence-to-sequence and LSTM to predict optimal locations in the input files to execute the mutations. It can result in notably more unique code paths, code coverage, and crashes on different input formats.

(3) Generative Adversarial Network

Since its emergence, GAN has become one of the most popular research directions in deep learning. The architecture for GAN consists of two parts: a generator and a discriminator. The generator tries to generate fake data with the same distribution as real data as possible. The discriminator makes a judgment about the input of the generation data and real data, and constantly adjusts model parameters based on the loss function. A simple GAN utilizes a multi-layer perceptron (MLP) to implement the generator and discriminator. Deep convolutional generative adversarial network (DCGAN) [88] replaces the MLP of the simple GAN with CNN. TextGAN [89] and SeqGAN [90] implement discriminators through a recurrent neural network, and the generator is implemented by CNN.

Seed file is one of the critical factors influencing fuzzing effectiveness. Some researchers are focusing on providing fuzzing with a better seed file so that more crashes can be found. In [91], the performance of the AFL was improved by using novel seed files built from a GAN model. Similarly, GAN can be used to generate test cases. Li et al. [74] proposed an efficient fuzzing approach applying WGANs to construct fuzzing tests for industrial control protocols. This approach can generate similar data frames without requiring detailed protocol specifications by learning the distribution and structure of real data frames. Hu et al. [92] utilized the generative adversarial network, using an RNN with LSTM cells as the generative model and a CNN as the discriminative model, to train a generative model on the protocol messages dataset to uncover the protocol grammar. Then the model can produce fake messages, which share various similarities with the authentic ones.

(4) Convolutional Neural Network

The CNN is a deep neural network with a convolutional structure. A typical convolutional neural network mainly includes an input layer, a convolutional layer, a pooling layer, a full connection layer, and an output layer. It is essentially a map from input to output, which can learn plenty of mapping relations between the input and output without any precise mathematical expression. Zong et al. [75] proposed an approach based on a 3-layer CNN, which can filter out invalid inputs without the program

execution. In this method, the deep learning model is trained to predict whether a program can run the target vulnerable code using a new input generated by learning from previous executions.

4.2.2 Applications of RL in Fuzzing

RL [93] refers to the machine learning problem in which the agent learns the optimal behavior strategy in the interaction with the environment. Its essence is the learning problem of the most ordered decision. The agent acquires the current state of the environment and selects an action according to the current state. Then the action acts on the environment, the environment responds to the action, and the agent gets a reward. Due to the similarity between the problem of selecting mutation operators and the types of problems that can be tackled using RL, existing research has predominantly focused on leveraging RL techniques to improve the process of selecting better mutation operators.

Bottinger et al. [76] regarded fuzzing as an RL problem employing Markov decision processes and present a fuzzing method using deep Q-learning that selects highly-rewarded mutation actions. Becker et al. [77] proposed an autonomic fuzzing framework that leverages an RL model to learn the fuzzing strategy. The model adopts three different reward functions of monitoring, debugging, and tracing networks to select the best strategies for fuzzing.

For testing Commercial-Off-The-Shelf 4G/long term evolution (LTE) Android mobile devices, LTE-oriented emulation-instrumented fuzzing testbed is constructed in [78]. A fuzzer is trained using RL, which can balance its vulnerability mining efforts between exploitation and exploration, avoiding undesirable crashes owing to inconsistent process states after behavior perturbation. Drozd et al. [79] obtained state information using program monitors of low level virtual machine (LLVM) Sanitizers. This state information is used by RL to optimize mutation operators. The OpenAl Gym is integrated with libFuzzer, which can take advantage of advances in RL and fuzzing to reach a deeper coverage.

5 Application Scenarios

5.1 Fuzzing for Industrial Control Protocols

Industrial control system (ICS) is a general term that covers several types of control systems, encompassing supervisory control and data acquisition (SCADA) systems, distributed control systems (DCS), and other control system configurations [94]. The ICS protocol takes a key role in the communication between ICS components. As shown in Fig. 7, many protocols are applied to the industrial control system. Profibus, Modbus, Distributed Network Protocol (DNP) 3.0, and other protocols are common in multiple industries, which typically have few or no security capabilities such as encryption and authentication to keep data from unauthorized access or modification. Ensuring the security of ICS protocols is essential to guarantee ICS operation safety. Test case generation is a key issue in protocol fuzzing. Many efforts are focusing on test case generation.

The work in [95] leveraged coverage-guided packet crack and generation to construct test cases. Those useful packets triggering new path coverage are retained and then cracked into pieces. Higher-quality new packets are constructed by processing these pieces for further testing. A new test case generation method targeting protocol fuzzing was proposed in [96]. The construction of the protocol classification tree provides a basic unit for generating test cases. The fuzzing data for each field is generated by performing a Cartesian product between the sets of attribute values. By mutating all protocol fields, a test case for protocol is obtained.

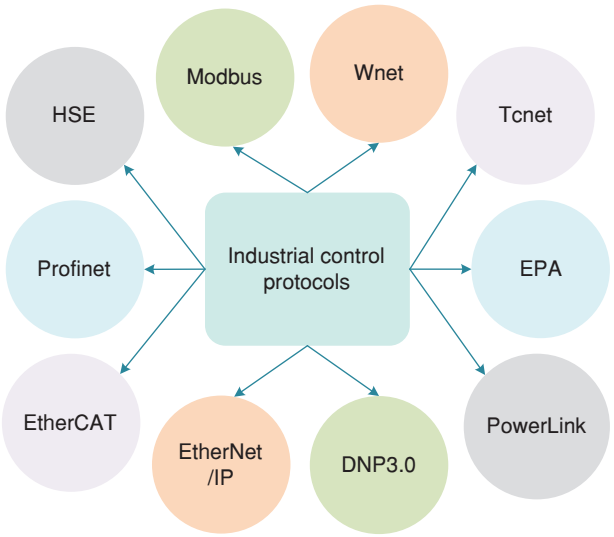

Figure 7: A summary of frequently used industrial control protocols

Lv et al. [97] proposed an efficient fuzzing method for constructing spurious yet useful protocol messages. They leverage the power of Bidirectional Long short-term memory (BLSTM) and DCGAN to learn the sequential structure of real-world protocol messages, enabling the generation of similar data frames. While the approach shows promise, smarter log analysis approaches and result analysis strategies are required to further enhance the statistical analysis efficiency and effectiveness of the generated protocol messages.

5.2 Web Fuzzing

Executed by a web server, a web application is a software application, which responds to dynamic web page requests over HTTP [98]. Web applications [99] have become increasingly vulnerable and exposed to malicious attacks with the increasing development of the Internet. Table 5 provides a summary of the common attacks exploiting web vulnerabilities. The main security problem faced by web applications is the acceptance and handling of potentially malicious and unauthenticated data. Providing malicious inputs is one of the strengths of fuzzing. Many researchers mine vulnerabilities in the web application by fuzzing.

Table 5: A summary of common attacks exploiting web vulnerabilities

Categories	Attacks
Authentication	Weak password recovery validation, insufficient authentication, brute force
Authorization	Insufficient authorization, insufficient session expiration, credential/session prediction, session fixation
Client-side attacks	Cross-site scripting, content spoofing

(Continued)

Table 5 (continued)

Categories	Attacks
Command execution	Format string attack, buffer overflow, structured query language injection, light directory access protocol injection, operating system (OS) commanding, server-side includes injection, XML path injection
Information disclosure	Path traversal, directory indexing, predictable resource location, information leakage
Logical attacks	Denial of service, abuse of functionality, insufficient anti-automation, insufficient process validation

In web browsers, the JavaScript interpreter is particularly prone to security issues. Only the semantically valid program can be accepted by a JavaScript interpreter. A built-in JavaScript grammar is needed for JavaScript interpreter fuzzing. LangFuzz takes the grammar as its input and can use its grammar to learn code fragments from a given code base. It can use grammar to randomly produce effective programs, which contain code fragments that may lead to invalid behavior.

Classical black-box web vulnerability scanners neglect the fact that a web application's state can be changed by any request. Ignoration of the state of a web application makes scanners only test a fraction of the web application. A partial model of the web application's state machine is constructed in [100]. The model can be employed to fuzz the application in a state-aware manner, thus more vulnerabilities can be discovered.

Client side validation (CSV) vulnerabilities are due to the unsecured use of untrusted data in the client code of a web application. The growth of complexity in JavaScript application makes CSV easy to appear. FLAX [101] is built to systematically discover CSV vulnerabilities by combining dynamic taint analysis with random fuzzing. It also reduces JavaScript semantics to an intermediate language with a simple type system and a few operations. Compared with tools that use symbolic execution techniques, FLAX is lightweight with no false positives.

Cross-site scripting (XSS) is one of the top vulnerabilities. KameleonFuzz [102] is the first black-box genetic algorithm (GA) driven fuzzing tool for Type-1 and 2 XSS. The generation and evolution of malicious inputs are implemented by a genetic algorithm and directed by an attack grammar. It has high XSS revealing capabilities with no false positives.

5.3 OS Kernel Fuzzing

Fig. 8 shows the number of vulnerabilities reported in the Linux kernel from 2005 to 2023, which is derived from [103]. We can see that many Linux kernel vulnerabilities have been reported every year since 2005. Kernel vulnerabilities can be fatal to a system. An attacker could exploit a kernel vulnerability to gain full privileges. It would be inefficient to apply fuzzing directly to the kernel. First, feedback mechanisms cannot be applied. Additionally, there is nondeterminism caused by interrupts, kernel threads, statefulness, and similar mechanisms, which may pose some problems. Furthermore, a kernel crash may slow down the performance of the fuzzer when a process fuzzes its kernel. To address the above problems, security researchers have come up with many solutions to applying fuzzing to the kernel.

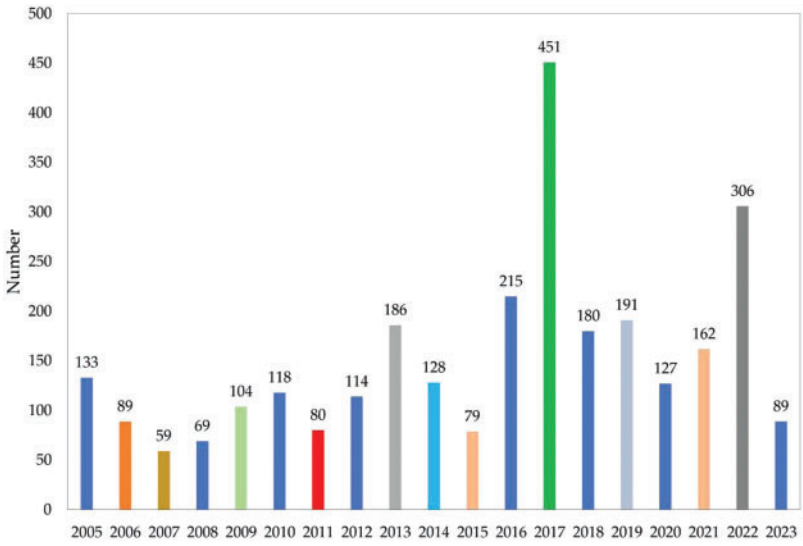

Figure 8: Linux kernel vulnerabilities by year

Syzkaller [104] is an unsupervised coverage-guided kernel fuzzer. While initially developed for the Linux kernel, Syzkaller has since been extended to support other operating systems as well. The template and code coverage information are used to guide the fuzzing process [105]. Syzkaller utilizes declarative descriptions of the system calls to present executable test cases [106]. An executable sequence of system calls can form the "Prog" structure in Syzkaller, which represents a single input for execution. A large number of Progs are grouped to form a test case corpus. Due to its effectiveness in identifying vulnerabilities in kernel code, Syzkaller is currently one of the most widely-used systems call fuzzers for the Linux kernel [107].

Aiming at the problem that existing fuzzers are limited to user space components of an operating system, kernel-AFL [108] is proposed. This tool improves coverage-guided kernel fuzzing in an os-independent and hardware-assisted way by utilizing a hypervisor (Intel VT-x) and Intel's Processor Trace (PT) technology. The first is a technique used to improve the efficiency of virtualization, which makes it possible for the virtual machine monitor to operate and control virtual machines faster, more reliable, and safer than traditional virtualization. The second PT tracing technique allows the CPU to collect partial contextual information about the operation of instructions, which is important for inferring Fuzz input. It has a better performance than TriforceAFL [109].

Many kernel fuzzers find vulnerabilities by calling random kernel application programming interface (API) functions with randomly generated parameter values. However, some challenges appear with this approach. One is that the order of kernel API calls should be valid. The other is that the parameters of API calls should have random yet well-formed values following the API specification. Inferred model-based fuzzer (IMF) [110] addresses these challenges by leveraging an automatic model inference technique. An API model is deduced by analyzing kernel API call sequences obtained from executing a regular program. Then the model is employed to generate test cases iteratively during the fuzzing process. IMF is the first model-based fuzzer running on macOS, but the technique proposed can also be employed in other operating systems.

Relying on hand-coded rules to generate valid seed sequences of system calls limits the effectiveness of the fuzzers. Therefore, MoonShine [111] is proposed, which automatically produces seeds for

OS fuzzers by extracting system call traces gathered from executions of real programs. To start with, it captures system call traces along with the coverage achieved by each call when real-world programs are executed. Moreover, the calls contributing the newest coverage are selected and the dependencies of each such call are identified. Ultimately, these calls are classified into seed programs.

Hybrid fuzzing [112] combining fuzzing and symbolic execution is effective in vulnerability mining. However, there are some challenges when it is applied to kernel fuzzing. One is that the use of a large number of indirect control shifts to support polymorphism in the kernel makes traditional fuzzing inefficient. Another problem is that inferring the right sequence (and dependency) of syscalls is difficult. Moreover, the structures in syscall arguments are nested, which makes interface templating difficult.

The study in [113] introduced the first hybrid kernel fuzzer, which enhances the efficiency of hybrid kernel fuzzing by handling the challenges mentioned above. The indirect control flows are converted to direct ones by translating the original kernel at the compilation time. The system states are reconstructed by inferring the right calling sequence. For nested structures in syscall arguments, it retrieves nested syscall arguments during runtime by leveraging the domain knowledge of how the kernel deals with the arguments.

5.4 Fuzz Testing for Deep Learning System

As deep learning systems penetrate into many fields such as autonomous driving, speech recognition, and financial markets, the discovery of their internal security vulnerabilities becomes a research hotspot.

In [114], a coverage-guided fuzzing method is applied to neural networks. The coverage metric is provided by a fast approximate nearest neighbor algorithm. Random mutations of inputs to a neural network are directed by the coverage metric toward the target of meeting user-specified constraints. The proposed method can detect numerical errors, disagreements between neural networks and their quantized versions, and unexpected behaviors in a character-level language model.

DeepHunter [115] is also a coverage-guided fuzzing framework that leverages the neuron coverage and coverage criteria proposed by DeepGauge [116]. A metamorphic mutation strategy is employed to generate images. It also applies a frequency-aware seed selection strategy based on the number of times a seed is fuzzed. DeepHunter can significantly reduce the false positive rate, while achieving high coverage and vulnerability discovering capabilities.

Deep learning fuzz (DLFuzz) [117] is the first differential fuzzing framework designed to mine vulnerabilities in deep learning systems. It continuously performs minute mutations on the inputs to improve the coverage of neurons and the prediction difference between the original and mutated inputs. This framework is valuable in uncovering incorrect behaviors of deep learning systems at an early stage, as well as in ensuring reliability and robustness.

CoCoFuzzing [118] introduces a novel approach to assess the robustness of neural code models. The method automates the generation of valid and semantically-preserving test cases using ten mutation operators. A neuron coverage-based guidance mechanism is employed to steer the test case generation process, ensuring the exploration of diverse and meaningful test scenarios. CoCoFuzzing enables the examination of the robustness and scalability of neural code models, thus contributing to the advancement of testing techniques in the context of neural code systems.

Duo [119] is a novel differential fuzzing framework that tests deep learning libraries directly at the operator level. Markov Chain Monte Carlo optimization sampling strategy and mutation operators

are adopted to generate tests. At the same time, a power schedule algorithm is used to set up the test input batch size. It can mine three types of defects, including implementation bugs, high cost of execution time, and high precision error.

Park et al. [120] proposed a mixed and constrained mutation (MCM) strategy for coverage-guided fuzzing in deep-learning systems. The mutation approach considers two primary objectives: input diversity and validity. By combining various image transformation algorithms, it achieves input diversity, while applying constraints on the transformation algorithm ensures input validity. As a result, the MCM strategy has shown remarkable improvements in both coverage and vulnerability mining, effectively enhancing the overall performance of fuzzing in deep-learning systems.

5.5 Smart Contracts Fuzzing

In short, a smart contract [121] is a computer program that automatically executes when certain conditions are satisfied. The decentralized, tamper-proof platform of blockchain is a perfect solution to the lack of a well-run platform for smart contracts, greatly promoting the development of the smart contract. Unfortunately, the current writing of smart contracts is not rigorous and standard, which leads to the possibility of internal vulnerabilities. The hidden vulnerabilities in smart contracts could cause significant financial losses to users if exploited by attackers. The infamous decentralized autonomous organization contract bug [122] resulted in a USD 60 million loss.

ContractFuzzer [123], proposed by Jiang et al., is the first fuzzing framework designed to mine security vulnerabilities [124] in smart contracts on the Ethereum platform [125]. By analyzing the application binary interfaces, it produces inputs that match the invocation grammars of the smart contracts under testing. To detect real smart vulnerabilities, new test oracles for diverse kinds of vulnerabilities are defined and Ethereum Virtual Machine (EVM) is instrumented to monitor smart contract execution. ContractFuzzer is a useful smart contract fuzzing tool with excellent practicality and applicability.

He et al. [126] presented a new fuzzing approach based on imitation learning [127], which learns an efficient and effective fuzzer from a symbolic execution expert. First, a large number of quality inputs are generated using a symbolic execution expert. Then, a fuzzing policy represented by neural networks is trained over these inputs. Finally, the learned policy can be employed to generate inputs for fuzzing smart contracts. This method shows its high coverage and vulnerability detection ability in smart contracts fuzzing.

sFuzz [128] introduces an innovative approach to the fuzz testing landscape by specializing in the assessment of smart contracts within the Ethereum ecosystem. By harnessing the power of AFL-based fuzzing, it incorporates a streamlined adaptive strategy for seed selection. This strategy, characterized by its lightweight design, contributes to the dynamic evolution of the fuzzing process. Built based on Aleth [129], it can be extended to various EVMs and oracles, and fuzzing strategies. sFuzz can discover vulnerabilities effectively and achieve high code coverage.

There are two main challenges, unobservable constraints, and blockchain effects in smart contract exploit generation. To address these problems, ETHPLOIT [130] uses taint constraints, EVM instrumentation, and dynamic seed strategy, to enable more effective exploit generation. It implements a dynamic seed strategy that uses runtime values as feedback to find the cryptographic function solution from execution histories. Instrumented EVM is leveraged to provide custom configurations. It also utilizes a taint analysis to guide the generation of a transaction sequence, which minimizes search space and improves the efficiency of the fuzzing process.

6 Challenges and Future Directions

By analyzing the existing research results, we can identify and summarize some trends based on the reviewed paper, which may inspire further works and new ideas.

As the demand for software functionality increases, so does the complexity of software. This complexity leads to an increase in the number of execution paths in the software. Therefore, generating high-quality test cases and minimizing unnecessary testing costs have become pressing issues to address. The combination of vulnerability prediction based on deep learning is still in its infancy, and it is believed that this direction will flourish in the future.

An increasing number of researchers are adopting deep learning methods to address the challenges in fuzzing. However, these learning-based approaches heavily rely on datasets. Presently, most studies use self-constructed datasets, which often suffer from accuracy and reliability issues. The absence of a precise and open-source benchmark dataset hinders the widespread application of deep learning in fuzzing. Constructing a unified and standardized open-source vulnerability dataset that covers all types of vulnerabilities is a hot topic.

Currently, coverage-guided grey box fuzzing stands as one of the most effective fuzzing techniques. In testing, it aims to cover as many program paths as possible within a limited time budget, ideally leading to the discovery of more vulnerabilities. However, the efficiency improvement of coverage-guided grey box fuzzing, backed by conventional technologies, has reached a bottleneck due to the absence of new technology advancements. The generation of high-coverage test cases is still an important point of research. With the development of the program analysis technology, the two may be more closely combined in the future.

7 Conclusions

Fuzzing stands out as one of the most successful techniques for vulnerability mining. Over the past three decades, it has undergone significant evolution, leading not only to improvements in the existing solutions but also to the emergence of new ideas, and in some instances, groundbreaking practical breakthroughs. This paper offers a comprehensive overview of fuzzing, covering background knowledge, representative fuzzers, key techniques, and diverse application scenarios. By analyzing the current research landscape, we summarize the challenges encountered by fuzzing and propose the potential future research directions. We anticipate that researchers will delve into these challenges to enhance the efficiency of fuzz testing techniques and improve system security [131]. We suggest that future work should be explored in the following aspects: building one-stop cluster fuzzing platforms, combining deep learning with fuzzing, constructing unified standardized datasets, and leveraging program analysis.

Acknowledgement: None.

Funding Statement: This work was supported in part by the National Natural Science Foundation of China under Grants 62273272, 62303375, and 61873277, in part by the Key Research and Development Program of Shaanxi Province under Grant 2023-YBGY-243, in part by the Natural Science Foundation of Shaanxi Province under Grant 2020JQ-758, in part by the Youth Innovation Team of Shaanxi Universities, and in part by the Special Fund for Scientific and Technological Innovation Strategy of Guangdong Province under Grant 2022A0505030025.

Author Contributions: Conceptualization: Zhenhua Yu; methodology: Zhengqi Liu; formal analysis: Xuya Cong and Xiaobo Li; original draft preparation: Zhengqi Liu; review and editing: Zhenhua Yu and Xuya Cong; visualization: Xiaobo Li; project administration: Zhenhua Yu; funding acquisition: Zhenhua Yu. All authors reviewed the results and approved the final version of the manuscript.

Availability of Data and Materials: Not applicable.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] Tomcat, "Apache Tomcat," 2023. [Online]. Available: https://tomcat.apache.org/ (accessed on 26/03/2023)

[2] CVE-2020-1938, 2023. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2020-1938 (accessed on 26/03/2023)

[3] Z. Yu, X. Duan, X. Cong, X. Li and L. Zheng, "Detection of actuator enablement attacks by Petri nets in supervisory control systems," *Mathematics*, vol. 11, no. 4, pp. 943, 2023. https://doi.org/10.3390/math11040943

[4] X. Cong, M. P. Fanti, A. M. Mangini and Z. Li, "Critical observability verification and enforcement of labeled Petri nets by using basis markings," *IEEE Transactions on Automatic Control*, pp. 1–8, 2023. https://doi.org/10.1109/TAC.2023.3292747

[5] Definition of software bug, 2023. [Online]. Available: https://www.lawinsider.com/dictionary/software-bug (accessed on 26/03/2023)

[6] CVE definition, 2023. [Online]. Available: https://nvd.nist.gov/vuln (accessed on 26/03/2023)

[7] M. Böhme, C. Cadar and A. Roychoudhury, "Fuzzing: Challenges and reflections," *IEEE Software*, vol. 38, no. 3, pp. 79–86, 2020.

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

[9] X. Zhu, S. Camtepe and Y. Xiang, "Fuzzing: A survey for roadmap," *ACM Computing Surveys*, vol. 54, no. 11, pp. 1–36, 2022.

[10] V. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1980–1997, 2019.

[11] J. Liang, Y. Jiang, M. Wang, X. Jiao, Y. Chen *et al.,* "DeepFuzzer: Accelerated deep greybox fuzzing," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 6, pp. 2675–2688, 2019.

[12] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele *et al.,* "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2021.

[13] Y. Li, S. Ji, C. Lyu, Y. Chen, J. Chen *et al.,* "V-Fuzz: Vulnerability prediction-assisted evolutionary fuzzing for binary programs," *IEEE Transactions on Cybernetics*, vol. 52, no. 5, pp. 3745–3756, 2020.

[14] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida *et al.,* "VUzzer: Application-aware evolutionary fuzzing," in *Proc. of NDSS*, San Diego, CA, USA, pp. 1–14, 2017.

[15] P. Godefroid, H. Peleg and R. Singh, "Learn&Fuzz: Machine learning for input fuzzing," in *Proc. of ASE*, Urbana, IL, USA, pp. 50–59, 2017.

[16] J. Wang, B. Chen, L. Wei and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *Proc. of S&P*, San Jose, CA, USA, pp. 579–594, 2017.

[17] AFL, "American fuzzy lop," 2015. [Online]. Available: http://lcamtuf.coredump.cx/afl/ (accessed on 26/03/2023)

[18] D. She, K. Pei, D. Epstein, J. Yang, B. Ray *et al.,* "NEUZZ: Efficient fuzzing with neural program smoothing," in *Proc. of S&P*, San Francisco, CA, USA, pp. 803–817, 2019.

[19] C. Lv, S. Ji, Y. Li, J. Zhou, J. Chen *et al.,* "SmartSeed: Smart seed generation for efficient fuzzing," arXiv:1807.02606, 2018.

[20] S. Michael, G. Adam and A. Pedram, "What is fuzzing," in *Fuzzing: Brute Force Vulnerability Discovery*, 1st ed., Boston, MA, USA: Addison-Wesley Professional, pp. 21–23, 2007.
[21] R. Kaksonen, M. Laakso and A. Takanen, "Software security assessment through specification mutations and fault injection," in *Communications and Multimedia Security Issues of the New Century*, pp. 173–183, Berlin, Germany: Springer, 2001.
[22] Gitlab, "Peach-fuzzer-community," 2021. [Online]. Available: https://gitlab.com/peachtech/peach-fuzzer-community (accessed on 26/03/2023)
[23] A. Pedram and P. Aaron, "Sulley," 2023. [Online]. Available: https://github.com/OpenRCE/sulley (accessed on 26/03/2023)
[24] P. Godefroid, M. Y. Levin and D. A. Molnar, "Automated whitebox fuzz testing," in *Proc. of NDSS*, San Diego, California, USA, pp. 151–166, 2008.
[25] funfuzz, "Javascript engine fuzzers," 2023. [Online]. Available: https://github.com/MozillaSecurity/funfuzz (accessed on 26/03/2023)
[26] V. Ganesh, T. Leek and M. C. Rinard, "Taint-based directed whitebox fuzzing," in *Proc. of ICSE*, Vancouver, BC, Canada, pp. 474–484, 2009.
[27] Z. Yu, A. Sohail, M. Jamil, O. A. Beg and J. M. R. Tavares, "Hybrid algorithm for the classification of fractal designs and images," *Fractals*, 2022. https://doi.org/10.1142/S0218348X23400030
[28] M. Böhme, V. T. Pham and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2019.
[29] M. Böhme, V. T. Pham, M. D. Nguyen and A. Roychoudhury, "Directed greybox fuzzing," in *Proc. of CCS*, Dallas, TX, USA, pp. 2329–2344, 2017.
[30] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A. Sadeghi *et al.,* "NAUTILUS: Fishing for deep bugs with grammars," in *Proc. of NDSS*, San Diego, CA, USA, pp. 1–15, 2019.
[31] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang *et al.,* "Driller: Augmenting fuzzing through selective symbolic execution," in *Proc. of NDSS*, San Diego, CA, USA, pp. 1–16, 2016.
[32] K. Böttinger and C. Eckert, "Deepfuzz: Triggering vulnerabilities deeply hidden in binaries," in *Proc. of the Int. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment*, San Sebastian, Spain, pp. 25–34, 2016.
[33] D. Molnar, X. C. Li and D. A. Wagner, "Dynamic test generation to find integer bugs in x86 binary linux programs," in *18th USENIX Security Symp.*, Montreal, Canada, pp. 67–82, 2009.
[34] S. Nagy and M. Hicks, "Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing," in *2019 IEEE Symp. on Security and Privacy (SP)*, San Francisco, CA, USA, pp. 787–802, 2019.
[35] Y. Wang, X. Jia, Y. Liu, K. Zeng, T. Bao *et al.,* "Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization," in *Proc. of NDSS*, San Diego, CA, USA, pp. 1–17, 2020.
[36] T. Wang, T. Wei, G. Gu and W. Zou, "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *Proc. of S&P*, Oakland, CA, USA, pp. 497–512, 2010.
[37] I. Haller, A. Slowinska, M. Neugschwandtner and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," in *Proc. of USENIX Security*, Washington DC, USA, pp. 49–64, 2013.
[38] P. Godefroid, A. Kiezun and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proc. of PLDI*, New York, USA, pp. 206–215, 2008.
[39] C. Holler, K. Herzig and A. Zeller, "Fuzzing with code fragments," in *Proc. of USENIX Security*, Bellevue, WA, USA, pp. 445–458, 2012.
[40] J. Wang, B. Chen, L. Wei and Y. Liu, "Superion: Grammar-aware greybox fuzzing," in *Proc. of the 2019 IEEE/ACM 41st Int. Conf. on Software Engineering (ICSE)*, Montreal, QC, Canada, IEEE, pp. 724–735, 2019.
[41] Y. Hsu, G. Shu and D. Lee, "A model-based approach to security flaw detection of network protocol implementations," in *Proc. of the 2008 IEEE Int. Conf. on Network Protocols*, Orlando, FL, USA, pp. 114–123, 2008.

[42] D. Yang, Y. Zhang and Q. Liu, "Blendfuzz: A model-based framework for fuzz testing programs with grammatical inputs," in *Proc. of the 2012 IEEE Int. Conf. on Trust, Security and Privacy in Computing and Communications*, Liverpool, UK, pp. 1070–1076, 2012.
[43] C. Ş. Gebizli, D. Metin and H. Sözer, "Combining model-based and risk-based testing for effective test case generation," in *Proc. of the IEEE 8th Int. Conf. on Software Testing, Verification and Validation Workshops*, Graz, Austria, pp. 1–4, 2015.
[44] C. Lyu, S. Ji, C. Zhang, Y. Li, W. H. Lee *et al.,* "MOPT: Optimized mutation scheduling for fuzzers," in *Proc. of the USENIX Security*, Santa Clara, CA, USA, pp. 1949–1966, 2019.
[45] T. Yue, P. Wang, Y. Tang, E. Wang, B. Yu *et al.,* "EcoFuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit," in *Proc. of the USENIX Security*, Boston, MA, USA, pp. 2307–2324, 2020.
[46] X. Zhu and M. Böhme, "Regression greybox fuzzing," in *Proc. of the 2021 ACM SIGSAC Conf. on Computer and Communications Security*, Virtual Event, Korea, pp. 2169–2182, 2021.
[47] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.
[48] V. Chipounov, V. Georgescu, C. Zamfir and G. Candea, "Selective symbolic execution," in *Proc. of the 5th Workshop on Hot Topics in System Dependability*, Lisbon, Portugal, pp. 1–6, 2009.
[49] S. Wen, C. Feng, Q. Meng, B. Zhang, L. Wu *et al.,* "Testing network protocol binary software with selective symbolic execution," in *Proc. of the CIS*, Wuxi, China, pp. 318–322, 2016.
[50] C. Cadar, D. Dunbar and D. R. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. of the 8th USENIX Symp. on Operating Systems Design and Implementation*, San Diego, California, USA, pp. 209–224, 2008.
[51] P. Godefroid, N. Klarlund and K. Sen, "DART: Directed automated random testing," in *Proc. of the 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Chicago, Illinois, USA, pp. 213–223, 2005.
[52] V. Chipounov, V. Kuznetsov and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," in *Proc. of the ASPLOS*, Newport Beach, CA, USA, pp. 265–278, 2011.
[53] P. Wang, X. Zhou, K. Lu, T. Yue and Y. Liu, "SoK: The progress, challenges, and perspectives of directed greybox fuzzing," arXiv:2005.11907, 2020.
[54] LibFuzzer, 2023. [Online]. Available: http://llvm.org/docs/LibFuzzer.html (accessed on 26/03/2023)
[55] K. Serebryany, "Oss-fuzz," 2023. [Online]. Available: https://github.com/google/oss-fuzz (accessed on 26/03/2023)
[56] Google, "Honggfuzz," 2021. [Online]. Available: https://github.com/google/honggfuzz (accessed on 26/03/2023)
[57] D. E. Denning, "A lattice model of secure information flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236–243, 1976.
[58] J. S. Foster, M. Fähndrich and A. Aiken, "A theory of type qualifiers," *ACM Sigplan Notices*, vol. 34, no. 5, pp. 192–203, 1999.
[59] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proc. of the 12th Annual Network and Distributed System Security Symp.*, San Diego, California, USA, pp. 3–4, 2005.
[60] M. G. Kang, "DTA++: Dynamic taint analysis with targeted controlflow propagation," in *Proc. of the 18th Annual Network and Distributed System Security Symp.*, San Diego, California, USA, pp. 1–14, 2011.
[61] E. J. Schwartz, T. Avgerinos and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proc. of the S&P*, Oakland, CA, USA, pp. 317–331, 2010.
[62] C. K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser *et al.,* "Pin: Building customized program analysis tools with dynamic instrumentation," *ACM Sigplan Notices*, vol. 40, no. 6, pp. 190–200, 2015.
[63] D. Bruening, Q. Zhao and R. Kleckner, "DynamoRIO," 2023. [Online]. Available: https://github.com/DynamoRIO/dynamorio (accessed on 26/03/2023)

[64] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *ACM Sigplan Notices*, vol. 42, no. 6, pp. 89–100, 2007.
[65] S. Jero, M. L. Pacheco, D. Goldwasser and C. NitaRotaru, "Leveraging textual specifications for grammar-based fuzzing of network protocols," in *Proc. of the AAAI*, Honolulu, Hawaii, USA, pp. 9478–9483, 2019.
[66] I. Schieferdecker, J. Grossmann and M. Schneider, "Model-based security testing," arXiv:1202.6118, 2012.
[67] C. Chen, B. Cui, J. Ma, R. Wu, J. Guo *et al.,* "A systematic review of fuzzing techniques," *Computers & Security*, vol. 75, pp. 118–137, 2018.
[68] S. Kirkpatrick, C. D. Gelatt and M. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.
[69] S. Forrest, "Genetic algorithms," *ACM Computing Surveys*, vol. 28, no. 1, pp. 77–80, 1996.
[70] P. Ghamisi and J. A. Benediktsson, "Feature selection based on hybridization of genetic algorithm and particle swarm optimization," *IEEE Geoscience and Remote Sensing Letters*, vol. 12, no. 2, pp. 309–313, 2014.
[71] A. Sharif, I. Sharif, M. A. Saleem, M. A. Khan, M. Alhaisoni *et al.,* "Traffic management in internet of vehicles using improved ant colony optimization," *Computers, Materials & Continua*, vol. 75, no. 3, pp. 5379–5393, 2023.
[72] Y. Wang, Z. Wu, Q. Wei and Q. Wang, "NeuFuzz: Efficient fuzzing with deep neural network," *IEEE Access*, vol. 7, no. 1, pp. 36340–36352, 2019.
[73] H. Zhao, Z. Li, H. Wei, J. Shi and Y. Huang, "SeqFuzzer: An industrial protocol fuzzing framework from a deep learning perspective," in *Proc. of the 12th IEEE Conf. on Software Testing, Validation and Verification*, Xi'an, China, pp. 59–67, 2019.
[74] Z. Li, H. Zhao, J. Shi, Y. Huang and J. Xiong, "An intelligent fuzzing data generation method based on deep adversarial learning," *IEEE Access*, vol. 7, pp. 49327–49340, 2019.
[75] P. Zong, T. Lv, D. Wang, Z. Deng, R. Liang *et al.,* "FuzzGuard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning," in *Proc. of the USENIX Security*, Boston, MA, USA, pp. 2255–2269, 2020.
[76] K. Böttinger, P. Godefroid and R. Singh, "Deep reinforcement fuzzing," in *Proc. of the 2018 IEEE Security and Privacy Workshops (SPW)*, San Francisco, CA, USA, pp. 116–122, 2018.
[77] S. Becker, H. Abdelnur and T. Engel, "An autonomic testing framework for IPv6 configuration protocols," in *Proc. of the 4th Int. Conf. on Autonomous Infrastructure, Management and Security*, Zurich, Switzerland, pp. 65–76, 2010.
[78] K. Fang and G. Yan, "Emulation-instrumented fuzz testing of 4G/LTE android mobile devices guided by reinforcement learning," in *Proc. of the 23rd European Symp. on Research in Computer Security*, Barcelona, Spain, pp. 20–40, 2018.
[79] W. Drozd and M. D. Wagner, "FuzzerGym: A competitive framework for fuzzing and learning," arXiv:1807.07490, 2018.
[80] L. Deng and D. Yu, "Deep learning: Methods and applications," *Foundations and Trends in Signal Processing*, vol. 7, no. 3, pp. 197–387, 2014.
[81] Y. LeCun, Y. Bengio and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.
[82] L. Medsker and L. Jain, "Recurrent neural networks," *Design and Applications*, vol. 5, no. 1, pp. 64–67, 2001.
[83] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
[84] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu D. Warde-Farley *et al.,* "Generative adversarial networks," *Communications of the ACM*, vol. 63, no. 11, pp. 139–144, 2020.
[85] J. Gu, Z. Wang, J. Kuen, L. Ma, A. Shahroudy *et al.,* "Recent advances in convolutional neural networks," *Pattern Recognition*, vol. 77, no. 1, pp. 354–377, 2018.
[86] M. Zakeri Nasrabadi, S. Parsa and A. Kalaee, "Format-aware learn&fuzz: Deep test data generation for efficient fuzzing," *Neural Computing and Applications*, vol. 33, no. 1, pp. 1497–1513, 2021.

[87] M. Rajpal, W. Blum and R. Singh, "Not all bytes are equal: Neural byte sieve for fuzzing," arXiv:1711.04596, 2017.
[88] A. Radford, L. Metz and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," arXiv:1511. 06434, 2015.
[89] Y. Zhang, Z. Gan and L. Carin, "Generating text via adversarial training," in *Proc. of the 2016 NIPS Workshop on Adversarial Training*, Barcelona, Spain, pp. 21–32, 2016.
[90] L. Yu, W. Zhang, J. Wang and Y. Yu, "SeqGAN: Sequence generative adversarial nets with policy gradient," in *Proc. of the AAAI*, San Francisco, California, 2017.
[91] N. Nichols, M. Raugas, R. Jasper and N. Hilliard, "Faster fuzzing: Reinitialization with deep neural models," arXiv:1711.02807, 2017.
[92] Z. Hu, J. Shi, Y. Huang, J. Xiong and X. Bu, "GANFuzz: A GAN-based industrial network protocol fuzzing framework," in *Proc. of the 15th ACM Int. Conf. on Computing Frontiers*, Ischia, Italy, pp. 138–145, 2018.
[93] R. S. Sutton and A. G. Barto, "Reinforcement learning: An introduction," *IEEE Transactions on Neural Networks*, vol. 16, no. 1, pp. 285–286, 2005.
[94] Z. Yu, H. Wang, D. Wang, Z. Li and H. Song, "CGFuzzer: A fuzzing approach based on coverage-guided generative adversarial networks for industrial IoT protocols," *IEEE Internet of Things Journal*, vol. 9, no. 21, pp. 21607–21619, 2022.
[95] Z. Luo, F. Zuo, Y. Shen, X. Jiao, W. Chang *et al.,* "ICS protocol fuzzing: Coverage guided packet crack and generation," in *Proc. of the 57th ACM/IEEE Design Automation Conf. (DAC)*, San Francisco, CA, USA, pp. 1–6, 2020.
[96] R. Ma, W. Ji, C. Hu, C. Shan and W. Peng, "Fuzz testing data generation for network protocol using classification tree," in *Proc. of the 2014 Communications Security Conf.*, Beijing, China, pp. 1–5, 2014.
[97] W. Lv, J. Xiong, J. Shi, Y. Huang and S. Qin, "A deep convolution generative adversarial networks based fuzzing framework for industry control protocols," *Journal of Intelligent Manufacturing*, vol. 32, no. 1, pp. 441–457, 2021.
[98] Web Application Security Consortium Glossary, 2023. [Online]. Available: http://www.webappsec.org/projects/ (accessed on 26/03/2023)
[99] D. M. Inamdar and S. Gupta, "A survey on web application security," *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, vol. 6, pp. 223–228, 2020.
[100] A. Doupé, L. Cavedon, C. Krügel and G. Vigna, "Enemy of the state: A state-aware black-box web vulnerability scanner," in *Proc. of USENIX Security*, Bellevue, WA, USA, pp. 523–538, 2012.
[101] P. Saxena, S. Hanna, P. Poosankam and D. X. Song, "FLAX: Systematic discovery of client-side validation vulnerabilities in rich web applications," in *Proc. of the NDSS*, San Diego, California, USA, pp. 1–17, 2010.
[102] F. Duchene, S. Rawat, J. Richier and R. Groz, "KameleonFuzz: Evolutionary fuzzing for black-box XSS detection," in *Proc. of the 4th ACM Conf. on Data and Application Security and Privacy*, San Antonio, Texas, USA, pp. 37–48, 2014.
[103] CVE Details, 2023. [Online]. Available: https://www.cvedetails.com (accessed on 26/03/2023)
[104] D. Vyukov, "Syzkaller," 2023. [Online]. Available: https://github.com/google/syzkaller (accessed on 26/03/2023)
[105] M. Cho, H. Jin, D. An and T. Kwon, "Evaluating code coverage for kernel fuzzers via function call graph," *IEEE Access*, vol. 9, pp. 157267–157277, 2021.
[106] Y. Shen, H. Sun, Y. Jiang, H. Shi, Y. Yang *et al.,* "Rtkaller: State-aware task generation for RTOS fuzzing," *ACM Transactions on Embedded Computing Systems*, vol. 20, no. 5, pp. 1–22, 2021.
[107] H. Sun, Y. Shen, C. Wang, J. Liu, Y. Jiang *et al.,* "HEALER: Relation learning guided kernel fuzzing," in *Proc. of the ACM SIGOPS 28th Symp. on Operating Systems Principles*, Virtual Event, Germany, pp. 344–358, 2021.
[108] S. Schumilo, C. Aschermann and R. Gawlik, "kAFL: Hardware-assisted feedback fuzzing for OS kernels," in *Proc. of the USENIX Security*, Vancouver, BC, Canada, pp. 167–182, 2017.

[109] Project Triforce, 2023. [Online]. Available: https://research.nccgroup.com/2022/09/27/whitepaper-project-triforce-run-afl-on-everything-2017/ (accessed on 26/03/2023)
[110] H. Han and S. K. Cha, "IMF: Inferred model-based fuzzer," in *Proc. of the 2017 ACM SIGSAC Conf. on Computer and Communications Security*, Dallas, TX, USA, pp. 2345–2358, 2017.
[111] S. Pailoor, A. Aday and S. Jana, "MoonShine: Optimizing OS fuzzer seed selection with trace distillation," in *Proc. of the USENIX Security*, Baltimore, MD, USA, pp. 729–743, 2018.
[112] B. S. Pak, "Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution," Master's thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, 2012.
[113] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin *et al.,* "HFL: Hybrid fuzzing on the linux kernel," in *Proc. of the NDSS*, San Diego, CA, USA, pp. 1–17, 2020.
[114] A. Odena, C. Olsson, D. Andersen and I. Goodfellow, "Tensorfuzz: Debugging neural networks with coverage-guided fuzzing," in *Proc. of the Int. Conf. on Machine Learning*, California, USA, pp. 4901–4911, 2019.
[115] X. Xie, L. Ma, F. Juefei-Xu, M. Xue, H. Chen *et al.,* "Deephunter: A coverage-guided fuzz testing framework for deep neural networks," in *Proc. of the 28th ACM SIGSOFT Int. Symp. on Software Testing and Analysis*, Beijing, China, pp. 146–157, 2019.
[116] L. Ma, J. F. Xu, F. Zhang, J. Sun, M. Xue *et al.,* "Deepgauge: Multi-granularity testing criteria for deep learning systems," in *Proc. of the 33rd ACM/IEEE Int. Conf. on Automated Software Engineering*, Montpellier, France, pp. 120–131, 2018.
[117] J. Guo, Y. Jiang, Y. Zhao, Q. Chen and J. Sun, "Dlfuzz: Differential fuzzing testing of deep learning systems," in *Proc. of the 2018 26th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering*, Lake Buena Vista, FL, USA, pp. 739–743, 2018.
[118] M. Wei, Y. Huang, J. Yang, J. Wang and S. Wang, "CoCoFuzzing: Testing neural code models with coverage-guided fuzzing," arXiv:2106.09242, 2021.
[119] X. Zhang, J. Liu, N. Sun, C. Fang, J. Liu *et al.,* "Duo: Differential fuzzing for deep learning operators," *IEEE Transactions on Reliability*, vol. 70, no. 4, pp. 1671–1685, 2021.
[120] L. H. Park, J. Kim, J. Park and T. Kwon, "Mixed and constrained input mutation for effective fuzzing of deep learning systems," *Information Sciences*, vol. 614, no. 1, pp. 497–517, 2022.
[121] C. D. Clack, V. A. Bakshi and L. Braine, "Smart contract templates: Foundations, design landscape and research directions," arXiv:1608.00771, 2016.
[122] S. Sayeed, H. Marco-Gisbert and T. Caira, "Smart contract: Attacks and protections," *IEEE Access*, vol. 8, no. 1, pp. 24416–24427, 2020.
[123] B. Jiang, Y. Liu and W. K. Chan, "ContractFuzzer: Fuzzing smart contracts for vulnerability detection," in *Proc. of the 33rd IEEE/ACM Int. Conf. on Automated Software Engineering (ASE)*, Montpellier, France, pp. 259–269, 2018.
[124] Z. Yu, H. Gao, X. Cong, N. Wu and H. H. Song, "A survey on cyber-physical systems security," *IEEE Internet of Things Journal*, 2023. https://doi.org/10.1109/JIOT.2023.3289625
[125] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, no. 2014, pp. 1–32, 2014.
[126] J. He, M. Balunović, N. Ambroladze, P. Tsankov and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *Proc. of the 2019 ACM SIGSAC Conf. on Computer and Communications Security*, London, UK, pp. 531–548, 2019.
[127] S. Ross, G. Gordon and D. Bagnell, "A reduction of imitation learning and structured prediction to no-regret online learning," in *Proc. of the 14th Int. Conf. on Artificial Intelligence and Statistics*, Fort Lauderdale, USA, pp. 627–635, 2011.
[128] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin and Q. T. Minh, "sFuzz: An efficient adaptive fuzzer for solidity smart contracts," in *Proc. of the ACM/IEEE 42nd Int. Conf. on Software Engineering*, Seoul, Korea, pp. 778–788, 2020.
[129] Aleth, 2023. [Online]. Available: https://github.com/ethereum/aleth/ (accessed on 26/03/2023)

[130] Q. Zhang, Y. Wang, J. Li and S. Ma, "ETHPLOIT: From fuzzing to efficient exploit generation against smart contracts," in *Proc. of the IEEE 27th Int. Conf. on Software Analysis, Evolution and Reengineering (SANER)*, London, ON, Canada, pp. 116–126, 2020.

[131] Z. Yu, Z. Wang, J. Yu, D. Liu, H. H. Song *et al.,* "Cybersecurity of unmanned aerial vehicles: A survey," *IEEE Aerospace and Electronic Systems Magazine*, pp. 1–25, 2023. https://doi.org/10.1109/MAES.2023.3318226