Fuzzing: A Survey for Roadmap

XIAOGANG ZHU and SHENG WEN, Swinburne University of Technology, Australia
SEYIT CAMTEPE, CSIRO Data61, Australia
YANG XIANG, Swinburne University of Technology, Australia

Fuzz testing (fuzzing) has witnessed its prosperity in detecting security flaws recently. It generates a large number of test cases and monitors the executions for defects. Fuzzing has detected thousands of bugs and vulnerabilities in various applications. Although effective, there lacks systematic analysis of gaps faced by fuzzing. As a technique of defect detection, fuzzing is required to narrow down the gaps between the entire input space and the defect space. Without limitation on the generated inputs, the input space is infinite. However, defects are sparse in an application, which indicates that the defect space is much smaller than the entire input space. Besides, because fuzzing generates numerous test cases to repeatedly examine targets, it requires fuzzing to perform in an automatic manner. Due to the complexity of applications and defects, it is challenging to automatize the execution of diverse applications. In this article, we systematically review and analyze the gaps as well as their solutions, considering both breadth and depth. This survey can be a roadmap for both beginners and advanced developers to better understand fuzzing.

CCS Concepts: • **Security and privacy** → *Software security engineering*

Additional Key Words and Phrases: Fuzz testing, security, fuzzing theory, input space, automation

ACM Reference format:
Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: A Survey for Roadmap. *ACM Comput. Surv.* 54, 11s, Article 230 (September 2022), 36 pages.
https://doi.org/10.1145/3512345

1 INTRODUCTION

Software security is a severe problem of computer systems, which affects people's daily life and even causes severe financial problems [109, 169]. Fuzz testing has become one of the most successful techniques to detect security flaws in programs. Fuzzing generates numerous test cases to test target programs repeatedly and monitors the exceptions of the program. The exceptions are the indicators of potential security flaws. Generally, fuzzing has a queue of seeds, which are the interesting inputs, and new inputs are generated via mutating seeds in an infinite loop. By steering computing resources for fuzzing in different ways [22, 23, 37, 40, 65, 188, 202], researchers can discover vulnerabilities more effectively and efficiently. Until now, fuzzing has discovered thousands

Authors. addresses: X. Zhu, S. Wen (corresponding author), and Y. Xiang, Swinburne University of Technology, John St, Hawthorn VIC 3122, Australia; emails: {xiaogangzhu, swen, yxiang}@swin.edu.au; S. Camtepe, CSIRO Data 61, Corner Vimiera and Pembroke Rd, Marsfield NSW 2122, Australia; email: Seyit.Camtepe@data61.csiro.au.

0360-0300/2022/09-ART230 $15.00
https://doi.org/10.1145/3512345

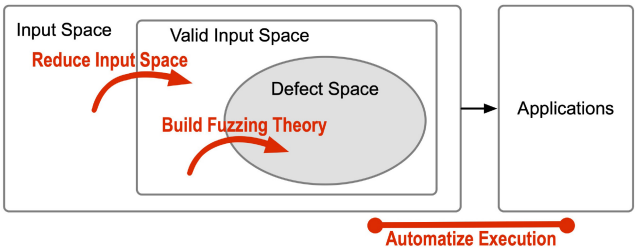


Fig. 1. Illustration of knowledge gaps in the domain of fuzzing. Fuzzing theories are utilized to improve the efficiency of defect detection (Section 3). The reduction of input space restricts the generated inputs to valid input space (Section 4). The automatic execution is the basis to apply fuzzing on various applications (Section 5).

of bugs in real-world programs, such as bugs discovered in general applications [30], **Internet of Things (IoT)** devices [36], firmware [211], kernels [194], and database systems [212].

Although fuzzing has achieved great success in detecting security flaws, it still has knowledge gaps for developing efficient defect detection solutions. As depicted in Figure 1, the three main gaps are the sparse defect space of inputs, the strict valid input space, and the automatic execution of various targets. The following paragraphs explain the details of the gaps.

Gap 1: Sparse defect space of inputs. Defects are sparse in applications, and only some specific inputs can trigger the defects. Because the essential purpose of fuzzing is to detect defects in target applications, fuzzing theories are required to generate inputs that belong to the defect space. Some security flaws are shallow so that they can be discovered in a short time of fuzzing campaigns. However, many security flaws require fuzzing to examine complex execution paths and solve tight path constraints. Therefore, a highly efficient fuzzing algorithm requires a sophisticated understanding of both **programs under test (PUTs)** and security flaws. Because defects are usually unknown before fuzzing campaigns, fuzzing theories based on the understanding of PUTs and/or security flaws steer computing resources toward code regions that are more likely to have defects.

Gap 2: Strict valid input space. Fuzzing has been utilized in various applications, and each application requires its own specific inputs. Modern applications are increasingly larger, resulting in more complex specifications of inputs. Therefore, it is challenging to generate valid inputs that target applications accept. Moreover, to improve the efficiency of fuzzing, it is better that the generated inputs exercise different execution states (e.g., code coverage). This requires fuzzing to develop more advanced schemes for the generation of valid inputs. Without a systematic analysis of PUTs, it is almost impossible to restrict the input space precisely. For instance, a random mutation of PDF files may breach the specifications of PDF. Fuzzing needs to mutate PDF files carefully so that the generated inputs belong to the valid input space.

Gap 3: Various targets. Because fuzzing repeatedly tests PUTs for numerous trials, fuzzing processes are required to perform automatically for high efficiency. Due to the diversity of PUTs and defects, the execution environments vary. It is straightforward for some applications to be fuzzed in an automatic manner, such as command-line programs. However, many other applications require extra effort to be tested automatically, such as hardware. Moreover, the security flaws also require automatic indicators to record potential real flaws. Program execution crash is a widely used indicator as it automatically catches exceptions from operating systems. However, many other security flaws may not manifest themselves as crashes, such as data races. These flaws require careful designs so that they are automatically recorded during fuzzing campaigns.

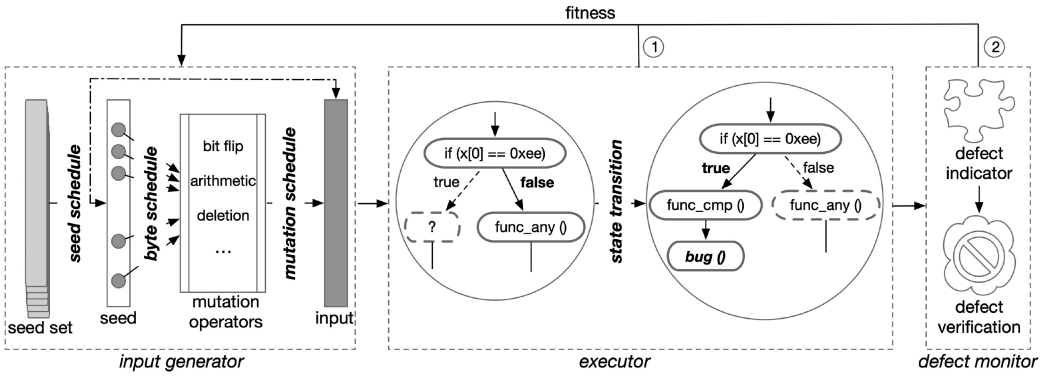


Fig. 2. General workflow of fuzzing. Essentially, fuzzing consists of three components: input generator, executor, and defect monitor.

The research community has put in many efforts to narrow down these gaps. Many theories have been proposed to formulate fuzzing processes partially. Various approaches have been designed to reduce the input space. Moreover, different execution scenarios have been successfully automatized. A few surveys have shed light on fuzzing, but none of them systematically review and analyze the gaps of fuzzing as well as their solutions [102, 107, 119]. Therefore, it is still unclear about questions such as what the gaps are, what the potential solutions are, and how to bridge the gaps. While other surveys focus on what fuzzing is, this article also explains how and why the existing solutions solve problems. With such information, this article builds a roadmap to bridge the gaps and pave roads for future research. For beginners who have limited knowledge of fuzzing, this article provides them with the conception of fuzzing and the existing solutions to improve fuzzing. For the advanced developers, this article also provides them with three main roads (i.e., three gaps) so that they can make a breakthrough by following one or more roads.

In this article, we systematically review and analyze the gaps and solutions of fuzzing, considering both breadth and depth. Since fuzzing is primarily a technique for security issues, we mainly collect papers from security and software conferences, including but not limited to four cyber security conferences and three software engineering conferences, from January 2008 to May 2021. The four cyber security conferences are *the ACM Conference on Computer and Communications Security (CCS)*, *the Network and Distributed System Security Symposium (NDSS)*, *the IEEE Symposium on Security and Privacy (S&P)*, and *the Usenix Security Symposium (USENIX)*. The three software engineering conferences are the *International Conference on Automated Software Engineering (ASE)*, *International Conference on Software Engineering (ICSE)*, and *ACM SIGSOFT Symposium on the Foundation of Software Engineering/European Software Engineering Conference (FSE/ESEC)*.

The article is organized as follows. Section 2 introduces the overview of fuzzing. Section 3 depicts fuzzing processes and various fuzzing theories to formulate the processes. Section 4 analyzes diverse solutions to reduce the search space of inputs. Section 5 analyzes how to automatize the execution of various PUTs and the detection of different bugs. Section 6 offers some directions for future research.

2 OVERVIEW OF FUZZING

Terminologies. We first introduce some terminologies for the convenience of reading. As depicted in Figure 2, an input is retained as a *seed* when the input achieves better fitness (e.g., new coverage). The *fitness* measures the quality of a seed or input. Later, fuzzing campaigns will select seeds in the

seed set to perform mutations. The mutation operators are called *mutators*. When a seed is selected from the seed set, a *power schedule* determines the energy assigned to the seed. The *energy* is the number of mutations assigned for the current fuzzing round. The implementation of a fuzzing algorithm is called a *fuzzer*.

In 1990, B. P. Miller et al. tested 90 programs by running them on random input strings and found that more than 24% of them crashed [129]. The program that generated random input strings was named *fuzz* by Miller. Since then, fuzz testing (or fuzzing) has become the name of the technique that identifies bugs via numerous test cases. Nowadays, because programs are increasingly more complex and fuzzing is an inexpensive approach, fuzzing is one of the major tools for bug discovery. As shown in Figure 2, fuzzing consists of three basic components, namely *input generator*, *executor*, and *defect monitor*. The *input generator* provides the *executor* with numerous inputs, and the *executor* runs target programs on the inputs. Then, fuzzing monitors the execution to check if it discovers new execution states or defects (e.g., crashes).

Fuzzing can be divided into generation-based and mutation-based fuzzing from the perspective of input generation. The generation-based fuzzing generates inputs from scratch based on grammars [3, 56, 57] or valid corpus [74, 79, 111, 112, 185]. As shown in Figure 2, generation-based fuzzing gets inputs directly from a seed set. On the other hand, the mutation-based fuzzing mutates existing inputs, which are called seeds, to get new inputs [22, 23, 66, 206, 216]. Given a seed set, mutation-based fuzzing performs the seed schedule, byte schedule, and mutation schedule to obtain inputs. Note that it is not necessary for fuzzing to go through all the steps in Figure 2. For example, generation-based fuzzing does not conduct the byte schedule or mutation schedule but focuses on selecting the optimal seed set from the initial input files.

Based on the amount of information observed during execution, fuzzing can be classified into blackbox, greybox, and whitebox fuzzing. Blackbox fuzzing does not have any knowledge about the internal states of each execution [3, 29, 36, 78, 99]. These fuzzers usually optimize fuzzing processes via utilizing input formats [59, 78, 99] or different output states [55, 61]. Whitebox fuzzing obtains all the internal knowledge of each execution, enabling it to systematically explore the state space of target programs. Whitebox fuzzing usually utilizes concolic execution (i.e., dynamic symbolic execution) to analyze target programs [71, 81, 150, 177, 204]. Greybox fuzzing obtains the knowledge of execution states between blackbox and whitebox fuzzing; e.g., many fuzzers use edge coverage as the internal execution states [7, 23, 66, 133, 216].

The most common execution states are the information of code coverage (e.g., basic blocks [133] or edges [206] in **control flow graphs (CFGs)**). The basic assumption for the usage of coverage is that the discovery of more execution states (e.g., new coverage) increases the possibility of finding bugs. For example, Miller [130] reports that the increase of 1% in code coverage results in an increase of 0.92% in bug discovery. Therefore, coverage-guided fuzzing aims to explore more code coverage [23, 66, 101, 203, 206]. Yet, due to the diverse usage of fuzzing, the execution states are not limited to code coverage. The states can be legality of executions for object-oriented programs [141], state machine for protocol implementations [3, 11, 55, 61, 64, 69, 154], alias coverage for concurrency implementations [194], neuron coverage for deep learning models [148], or execution logs for Android SmartTVs [1].

Fuzzers often use crashes as the indicator of security bugs because crashes offer straightforward automatic records [206]. The operating systems will automatically incur signals to inform program crashes. However, some defects do not manifest themselves as crashes; thus, fuzzers utilize other defect indicators, such as physical safety violations [41, 82]. Note that the indicators only show potential security issues, which are further verified by security tools or manual efforts to confirm a vulnerability [85, 168, 175].

3 FUZZING THEORY

The essential objective of fuzzing is to search for the test cases that can trigger defects (e.g., bugs). The challenge is that the search space is infinite due to the complexity of PUTs. Therefore, searching defect-triggering inputs with blind mutation is as hard as finding a needle in a haystack. To improve the possibility of exposing defects, fuzzers use feedback from executions, such as execution states or results, as the fitness. A typical fitness is based on code coverage (e.g., basic blocks or edges) [23, 39, 116, 157, 203, 210], which is used to determine the reward for the generated inputs. However, code coverage is not always available, and even when it is available, it may not be sensitive enough to expose defects (Section 3.5). Moreover, with only code coverage as feedback, generating exponentially more inputs per minute (i.e., using exponentially more machines) exposes linearly more vulnerabilities only [19]. Therefore, a common improvement is to optimize fuzzing processes or enrich the information for fitness.

Fuzzing theories intend to optimize fuzzing processes so that fuzzing can expose defects more effectively and efficiently; i.e., fuzzing can discover defects in a shorter time budget or with fewer test cases. The existing fuzzers use various solutions of optimization problems for fuzzing processes. As shown in Figure 2, fuzzing can optimize the *seed set*, *seed schedule*, *byte schedule*, and *mutation schedule* with the fitness based on execution states or defect discovery. Table 1 shows the optimization solutions for different fuzzing processes. The column *Fitness By* indicates how the corresponding solutions formulate fuzzing processes. The columns *Gene.*- and *Muta.-based* are generation-based and mutation-based fuzzing, respectively. The fuzzers in Table 1 are selected because their main contributions are the development of fuzzing theories.

3.1 Seed Set Selection

The optimization of seed sets focuses on minimizing the size of a set; i.e., it selects the fewest number of seeds that can cover all the discovered code coverage [2, 147, 206]. The reason for the minimization is that the redundancy of seeds wastes computing resources on examining the well-explored code regions. COVERSET [158] formulates the problem of minimizing a seed set as a ***minimal set cover problem*** **(MSCP)**, which minimizes the subsets that contain all elements. Since MSCP is an NP-hard problem, COVERSET uses a greedy polynomial approximation algorithm to obtain the minimal set.

3.2 Seed Schedule

With the seed set, the seed schedule aims to solve the problems of (1) which seed to be selected for the next round and (2) the time budget for the selected seed. In practice, instead of time budget, most fuzzers optimize the number of times of mutating the selected seeds [23, 203, 206]. Although these two problems are pretty clear, the existing solutions for optimizing them vary due to the complexity of PUTs or bugs. The most critical challenge is the agnostic of undiscovered code coverage or bugs. Before verifying bugs, one cannot know whether an input can trigger a bug; thus, the optimization problem does not have an effective fitness. Similarly, before examining code lines, one cannot obtain the probability distribution of program behaviors (e.g., branch behaviors); thus, it is (almost) impossible to find the global optimal solution mathematically. As a result, researchers approximately or partially formulate fuzzing processes based on diverse optimization problems. As shown in Figure 2, fuzzing theories can be designed based on the properties of defects (e.g., bug arrival) or execution states (e.g., edge transition).

3.2.1 Fitness by #Bugs. Generally, fuzzing utilizes two types of fitness for optimization problems, namely fitness based on bugs or execution states (e.g., code coverage). The fitness is a scalar that measures the quality of a seed or input. Since fuzzing is a technique for bug discovery, an

Table 1. Fuzzers and Their Optimization Solutions

Year	Fuzzer	Solution(Process)	Fitness By	Target App/Bug	Input		Runtime Info.
					Muta. -based	Gene. -based	
2006	Sidewinder [62]	MC(seed.) + GA(rete.)	block transition	general	✓		◐
2007	RANDOOP [141]	GA(rete.)	legality	object-oriented	✓		●
2013	FuzzSim [192]	WCCP(seed.)	#bugs	general	✓		●
2014	COVERSET [158]	MSCP(set.)	code coverage	general		✓	◐
		ILP(seed.)	#bugs		✓		●+◐+○
2015	Ruiter and Poll [55]	GA(rete.)	state machine	protocol	✓		●
2016	AFLFast [23]	MC(seed.) + GA(rete.)	path transition	general	✓		◐
2016	classfuzz [42]	MH(mutation.) + GA(rete.)	code coverage	JVM	✓		◐
2017	VUzzer [157]	MC(seed.) + GA(rete.)	block transition	general	✓		◐
2017	AFLGo [22]	SA(seed.) + GA(rete.)	path transition	general	✓		◐
2017	NEZHA [151]	GA(rete.)	asymmetry	semantic bugs	✓		●+◐
2017	DeepXplore [148]	GA(rete.)	neuron coverage	deep learning	✓		◐
2018	STADS [17]	Species(seed.)∗	state discovery	general	✓		◐
2018	CollAFL [66]	GA(rete.)	△code coverage	general	✓		◐
2018	Angora [37]	GD(byte.) + GA(rete.)	△code coverage	general	✓		◐
2019	DigFuzz [210]	MC(seed.) + GA(rete.)	block transition	general	✓		○
2019	MOPT [116]	PSO(mutation.) + GA(rete.)	code coverage	general	✓		◐
2019	NEUZZ [172]	NN(byte.) + GA(rete.)	branch behavior	general	✓		◐
2019	Cerebro [105]	MOO(seed.) + GA(rete.)	△code coverage	general	✓		◐
2019	DiffFuzz [138]	GA(rete.)	asymmetry	side-channel	✓		●
2020	AFLNET [154]	GA(rete.)	state machine	protocol	✓		◐
2020	EcoFuzz [203]	VAMAB(seed.) + GA(rete.)	path transition	general	✓		◐
2020	Entropic [21]	Shannon(seed.) + GA(rete.)	state discovery	general	✓		◐
2020	MTFuzz [171]	MTNN(byte.) + GA(rete.)	△branch behavior	general	✓		◐
2020	Ankou [118]	GA(rete.)	△code coverage	general	✓		◐
2020	FIFUZZ [87]	GA(rete.)	△code coverage	error-handling	✓		◐
2020	IJON [6]	GA(rete.)	△code coverage	general	✓		◐
2020	Krace [194]	GA(rete.)	alias coverage	data race	✓		◐
2021	AFL-HIER [88]	UCB1(seed.) + GA(rete.)	△path transition	general	✓		◐
2021	PGFUZZ [82]	GA(rete.)	safety policy	robotic vehicle	✓		●
2021	Aafer et al. [1]	GA(rete.)	validation log	SmartTV	✓		●
2021	AFLChurn [214]	SA(seed.) + ACO(byte.) + GA(rete.)	path transition + commit history	general	✓		◐

Based on different understanding of fuzzing processes, optimization solutions are utilized to formulate different processes. Different types of fitness are designed based on features of different applications or bugs.

MC: Markov Chain; **MSCP**: Minimal Set Cover Problem; **ILP**: Integer Linear Programming Problem; **WCCP**: Weighted Coupon Collector's Problem; **VAMAB**: Variant of Adversarial Multi-Armed Bandit; **UCB1**: Upper Confidence Bound, version one; **MH**: Metropolis-Hastings; **PSO**: Particle Swarm Optimization; **Shannon**: Shannon's entropy; **Species**∗: Models of Species Discovery; **ACO**: Ant Colony Optimization; **SA**: Simulated Annealing; **NN**: Neural Network; **MTNN**: Multi-task Neural Networks; **GA**: Genetic Algorithm; **GD**: Gradient Descent; **MOO**: Multi-objective Optimization; **R**: Random.

set.: Seed Set Selection; seed.: Seed Schedule; byte.: Byte Schedule; mutation.: Mutation Schedule; rete.: Seed Retention; ○: Whitebox Fuzzing; ◐: Greybox Fuzzing; ●: Blackbox Fuzzing.

△: More sensitive code coverage.

∗: The paper does not assign energy based on the model but in fact, the model can achieve energy assignment.

intuitive fitness is the number of bugs. In order to maximize the number of bugs, one approach is to schedule the time budget of each seed while randomly or sequentially selecting seeds. Without considering the execution states, the maximization problem can be simplified as an **Integer Linear Programming (ILP)** problem [158]. That is, ILP intends to maximize the number of bugs with the linear constraints, such as the upper bound of the time budget for each seed. By solving the ILP problem, one can automatically compute the time budget for each seed. Another insight is to regard the bug arrival process as the **weighted Coupon Collector's Problem (WCCP)** [13].

The WCCP estimates how much time is required before a new bug arrives in the process, which is similar to the problem about the expected purchases required before a consumer obtains a new coupon [192]. Each unique bug obtained by fuzzing is a type of coupon, and WCCP aims to predict the number of purchases (time budget) required for the arrival of new coupons (unique bugs). In order to predict the bug arrival time, WCCP requires the distribution of all bugs, which is estimated by observing the discovered bugs. Based on the distribution, fuzzing will gradually assign minimum time, which is predicted by WCCP, for seeds to expose new bugs. Both ILP and WCCP intend to assign more time budget to seeds that have more potential to expose bugs.

3.2.2 Fitness by State Transition (Markov Chain). Because bugs are sparse in PUTs, the optimization process will quickly converge to local optima when using the number of bugs as fitness. Thus, fuzzing will focus on the code regions that are related to discovered bugs. As a result, it may miss the opportunities to explore more code coverage. In this case, deep bugs, which are guarded by complex conditions, can evade fuzzing. To mitigate this problem, fuzzers calculate fitness based on execution states (e.g., code coverage) because execution states can provide fuzzing with more information. Most existing fuzzers compute fitness based on code coverage; i.e., they intend to discover more code coverage [21, 23, 62, 88, 157, 203]. Another reason for using code coverage is that larger code coverage indicates a higher possibility of bug discovery [130].

As depicted in Figure 2, if fuzzing succeeds in formulating state transitions, it can efficiently guide fuzzing to explore undiscovered states. A popular theory for modeling state transition is the Markov chain [93], which is a stochastic process that transitions from one state to another. In a nutshell, the Markov chain maintains a probability table, in which an element p_{ij} is the transition probability that the chain transitions from a state i to another state j. In order to formulate fuzzing, one solution is to define a basic block in CFGs as a state, and the state transition is the jump from one basic block to another. Because the transition probabilities of blocks cannot be obtained without running the program, fuzzers calculate the probabilities by recording the frequencies of block transitions during fuzzing [62, 157, 210]. For instance, if an edge *AB* has been examined 30 times while its neighbor edge *AC* has been examined 70 times, the transition probabilities of edges *AB* and *AC* are 0.3 and 0.7, respectively (Figure 3(a)). The transition probabilities in a path *ABDEG* can be obtained in the same way. Suppose that the probabilities for edges *AB*, *BD*, *DE*, and *EG* are 0.3, 0.25, 0.6, and 0.8, respectively. Then, the fitness of a seed exercising the path *ABDEG* can be calculated as $0.3 \times 0.25 \times 0.6 \times 0.8 = 0.036$. At the beginning of fuzzing, the transition probabilities can be obtained by the Monte Carlo method [125], which examines basic blocks randomly. With the fitness calculated based on the Markov chain, fuzzers can guide the calculation of energy assignment [62, 157] or select the hardest path (lowest transition probability) to be solved by concolic execution [210]. Basically, the lower the transition probability, the better the fitness. The motivation is that the hard-to-reach code regions (low transition probabilities) require more energy to fuzz because the easy-to-reach regions can be easily well explored.

The mutation-based fuzzing generates new inputs by mutating seeds, and each input exercises an execution path. This provides another perspective to model a fuzzing process based on the Markov chain. Specifically, the state is defined as an execution path exercised by an input; meanwhile, the state transition is the mutation of an input t_i, which generates a new input t_j. Correspondingly, the state transition is also the transition from a path i, which is exercised by the input t_i, to another path j, which is exercised by the input t_j. Similar to block transitions, the probabilities of path transitions are calculated based on previous executions during fuzzing. As deduced by AFLFast [23], the minimum energy required for a state i to discover a new state j is $1/p_{ij}$, where p_{ij} is the transition probability. Therefore, AFLFast assigns more energy to less frequent paths,

i.e., the paths with low path transition probabilities. Variants of this model are investigated in **directed greybox fuzzing (DGF)** [22, 217] and **regression greybox fuzzing (RGF)** [214].

3.2.3 Fitness by State Transition (Multi-armed Bandit). Although the Markov chain has achieved great success in modeling state transitions, it is not profound enough to model the seed schedule of fuzzing. The Markov chain requires transition probabilities among all states so that fuzzing makes a proper decision. However, it is common that many states have not been examined during fuzzing, which indicates that the decisions based on the Markov chain are not optimal. For block transitions, one solution is to use the *rule of three* in statistics for the transitions from a discovered block to an undiscovered block [210]. For path transitions, a naive solution to obtain all transition probabilities is the Round-Robin schedule [156], which splits the time budget between seeds equally [158]. However, this solution cannot determine when to switch from Round-Robin to Markov chain. The balance between traversing all seeds and focusing on a specific seed is a classic *"exploration vs. exploitation"* problem. A better solution for solving the *"exploration vs. exploitation"* problem is to formulate the path transitions as a **Multi-armed Bandit (MAB)** problem [14].

For a Multi-armed Bandit problem, a player intends to maximize the total rewards by observing the rewards of playing some trials on the arms of a slot machine. The *exploration* is the process when the player plays all arms to obtain their reward expectations. When reward expectations of all arms are known, the *exploitation* is the process when the player only selects arms with the highest reward expectations. In order to formulate the path transitions, a seed t_i is defined as an arm. Meanwhile, the reward is the discovery of a new path exercised by an input, which is generated from the seed t_i.

Considering that the total number of seeds is increasing and the reward expectation of a seed is decreasing during fuzzing, EcoFuzz [203] proposes the **Variant of the Adversarial Multi-armed Bandit Model (VAMAB)** to solve the problem. Specifically, EcoFuzz adaptively assigns energy for unfuzzed seeds, i.e., the exploration process. With the rewards of all seeds, EcoFuzz estimates the expectation of a seed t_i as $(1 - p_{ii}/\sqrt{i})$, where p_{ii} is the self-transition probability (i.e., mutation of seed t_i results in exercising the same path i). The logic is that a low self-transition probability indicates that mutations of the seed can discover other paths (new paths) with high probability. Therefore, EcoFuzz prefers seeds with low self-transition probabilities. Similarly, AFL-HIER [88] also formulates the path transitions as a MAB problem. While EcoFuzz uses single metrics (i.e., edge coverage) to retain new seeds, AFL-HIER proposes to utilize multi-level coverage metrics, such as functions, edges, and basic blocks, to add new seeds. AFL-HIER chooses UCB1 [9], one of the MAB algorithms, to solve the MAB problem with multi-level coverage metrics.

3.2.4 Fitness by State Discovery. Both Markov chain and MAB formulate the state transitions of programs. However, the essential objective of fuzzing is to discover new states, e.g., new code coverage, new crashes, or new bugs. This motivates Böhme et al. [17, 20, 21] to formulate fuzzing processes as a species discovery problem [32, 33, 52]. In a nutshell, ecologists collect numerous samples from the wild, and the species in the samples may be abundant or rare. Ecologists extrapolate the properties of a complete assemblage, including undiscovered species, based on the samples. Similarly, inputs generated by fuzzers are the collected samples, and the input space of a program is the assemblage. Fuzzing categorizes inputs into different species based on specific metrics. For example, an execution path can be a species, and all inputs exercising the path belong to this species. In this case, a rare species is an execution path that a few inputs exercise. An important hypothesis in species discovery is that the properties of undiscovered species can almost be explained only by discovered rare species [31]. This hypothesis implies that fuzzing can assign more energy to the rare species (e.g., rare paths) to discover new states.

Based on the problem of species discovery, Entropic [21] understands fuzzing as a learning process; i.e., a fuzzer gradually learns more information about the program behaviors (species). Entropic proposes to use Shannon's entropy [170] to measure the efficiency of species discovery. The original Shannon's entropy H measures the average information of species and is calculated as $H = -\sum_i p_i log(p_i)$, where p_i is the probability of selecting the species $\mathcal{S}_i$. The entropy H is large (more information) if the collected samples contain many species; otherwise, the entropy H is small (less information) if the collected samples contain a few species. Meanwhile, Entropic argues that the rate of species discovery quantifies the efficiency of fuzzing processes. Deduced from Shannon's theory, Entropic measures the efficiency of species discovery for a seed. Specifically, p_i^t is the probability of mutating a seed t and generating an input that belongs to species $\mathcal{S}_i$. The learning rate of the seed t is calculated based on the probability p_i^t and an improved entropy estimator. Entropic concludes that more energy is assigned to seeds with a larger learning rate; i.e., seeds that discover new species more efficiently are assigned more energy.

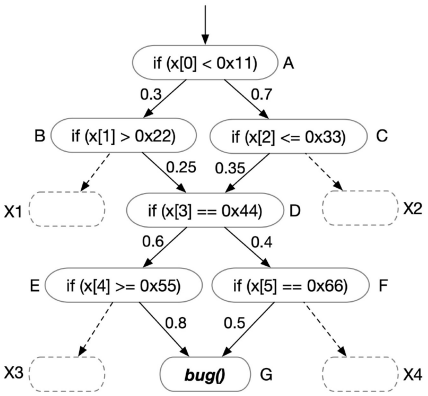
3.3 Byte Schedule

The byte schedule determines the frequencies of selecting a byte in a seed to be mutated. Most fuzzers select bytes heuristically based on execution information [7, 37, 38, 67, 101, 103, 157, 160, 187] or randomly [22, 23, 118, 203, 206]. The byte schedule requires a more sophisticated understanding of program behaviors, such as path constraints or dataflow, than the seed schedule. Therefore, fuzzers focus on a less complex problem called the *importance* of bytes, which implies how the bytes influence fuzzing processes. Because most greybox fuzzers use edge coverage to test PUTs, the first approach is to define the importance as how the bytes influence branch behaviors. NEUZZ [172] and MTFuzz [171] model the relationships between input bytes and branch behaviors based on **deep learning (DL)** models. The gradients of DL models quantify the importance of bytes because the large gradient of a byte indicates that a small perturbation of the byte results in a significant difference in branch behavior. For later mutations, fuzzing will prioritize the bytes with higher importance to be mutated.

Another approach to quantify the importance of bytes is to define it based on the fitness of seeds. As analyzed in Section 3.2, the fitness of a seed reflects the quality of the seed. Therefore, fuzzing can focus on bytes that can improve the quality. AFLChurn [214] utilizes **Ant Colony Optimization (ACO)** [60] to learn how bytes influence the fitness of seeds. Similar to the process that an ant colony searches for food, if the change of a byte improves the fitness of a seed, fuzzing increases the score of the byte. When fuzzing continues the test, the scores of all bytes decrease over time, which reflects the pheromone evaporation of ACO. As a result, fuzzing prefers to select the bytes with higher scores.

3.4 Mutation Operator Schedule

As depicted in Figure 2, the final step of the input generator is to choose a mutation operator (mutator) to mutate the selected byte(s). The mutation schedule decides which mutator to be selected for the next mutation of bytes. The motivation for the mutation schedule is based on the observation that the efficiency of mutators varies [42, 116]. Classfuzz [42] argues that the mutators that have explored more new states have a higher probability of being selected. Thus, classfuzz assumes that **Markov Chain Monte Carlo (MCMC)** can model the process of the mutation schedule. Classfuzz adopts the Metropolis-Hastings algorithm [45], one of the MCMC methods, to solve the problem of the mutation schedule. Specifically, each mutator has a success rate that quantifies the new states explored by the mutator. Classfuzz first randomly selects the next mutator, and then it accepts or rejects the selection based on the success rates of both the current mutators and the selected ones.



(a) Coverage sensitivity. The bug can only be triggered by path *ACDEG*.

```
while (1){
  pa=a; pb=b;
  switch ( x[i] ) {
    case 'q': b--; break;
    case 'w': b++; break;
    case 'e': a--; break;
    case 'r': a++; break;
  }

  if(labyr[b][a] == '#')
     {bug ();}

  if(labyr[b][a] != ' ')
     {a = pa; b=pb;}
}
```

(b) Limitation of code coverage. Edge coverage can hardly trigger the bug (adapted from [6]).

Fig. 3. Sensitivity and limitation of code coverage. Edge coverage is limited to discover more execution states, including program defects.

MOPT [116] utilizes **Particle Swarm Optimization (PSO)** [94] to model the process of mutation selection. PSO uses several particles and gradually moves them toward their best positions. As to the problem of the mutation schedule, a mutator is a particle, and the position is the probability of selecting the mutator. A particle finds its best position when the particle yields the most number of new states at a position among all the other positions. Therefore, all particles (mutators) asymptotically find their own best positions (probabilities), which constructs the probability distribution for selecting mutators. Later fuzzing campaigns will select mutators based on the probability distribution.

3.5 Diverse Information for Fitness

Besides the schedule of seeds, bytes, or mutators, fitness can also be used to guide seed retention. Fuzzers usually utilize a **genetic algorithm (GA)** to formulate the process of seed retention. Specifically, a fuzzer generates an input by mutating a seed, and if the input explores new execution states (i.e., better fitness), the input is retained as a new seed. When selecting a seed for the next round of test (based on seed schedule), fuzzing may choose the new seed. Most coverage-guided fuzzers [7, 23, 116, 177, 204, 206] retain seeds based on edge coverage. In order to improve the ability of defect discovery, more sensitive code coverage is necessary to reveal more information of execution states. On the other hand, new types of fitness are designed for some specific scenarios, such as deep learning models [148] or robotic vehicles [82]. Note that the diversity of information is utilized for both seed retention and the schedule problems mentioned before.

3.5.1 Sensitive Code Coverage. The sensitivity of fitness indicates the ability to differentiate execution states. Many coverage-guided fuzzers [7, 23, 116, 177, 204, 206] implement a bitmap to provide fuzzing with the knowledge of edge coverage. Essentially, the bitmap is a compact vector, where the index of each element indicates an edge identifier. They calculate hash values $hash(b_i, b_j)$ for the edge identifiers, where b_i and b_j are block identifiers that are randomly assigned during instrumentation. Although this implementation is fast during execution, it sacrifices the precision of edge coverage. Specifically, this implementation of obtaining edge coverage incurs the problem of edge collision; i.e., two different edges are assigned with the same identifier [66, 216]. As shown in Figure 3(a), if edge identifiers $id_{AB} = id_{AC}$ and $id_{BD} = id_{CD}$, then the paths *ABD* and *ACD* are

regarded as the same one. Since fuzzing does not discover new coverage in this scenario, it will neglect the bug path *ACDEG*. Therefore, in order to assign unique edge identifiers, fuzzing assigns block identifiers carefully and proposes more precise hash functions [66].

With the bitmap, fuzzing can determine whether an input exercises new edges and retain the input as a new seed if it does. Specifically, fuzzers maintain an overall bitmap, which is a union of bitmaps of individual executions. When determining new edges, fuzzing compares an individual bitmap with the overall bitmap to check if new edges exist in the individual bitmap. However, the union of bitmaps loses information of executions [118]. For example, if paths *ABDEG* and *ACDFG* in Figure 3(a) have been exercised, the input that exercises the new path *ACDEG* will not be retained as a seed because all edges have already existed in the overall bitmap. Therefore, one solution is to research combining individual bitmaps [118]. Because the combination of bitmaps will introduce too many seeds, a critical challenge is to balance the efficiency of fuzzing and the sensitive coverage. One potential solution is to use dynamic Principal Component Analysis [191] to reduce the dimensionality of the dataset [118]. Other solutions to improve the sensitivity of edge coverage include path hash [198], calling context [37, 87, 171], multilevel coverage [88], and code complexity [105], which add extra information for the edge coverage.

The improvement of the bitmap focuses on searching for more code coverage. However, such improvement may fail to explore complex execution states. For instance, Figure 3(b) is the code snippet of a labyrinth, where the pair *(a, b)* indicates the position in the labyrinth. In order to trigger the *bug()*, the pair *(a, b)* has to be the one with specific values. However, the *switch* snippet has only four edges and can be quickly explored. After that, fuzzing has no guidance for reaching the bug location. Therefore, an effective solution is to introduce the human-in-the-loop approach for the guidance of exploring complex execution states [6]. In the example of Figure 3(b), an analyst can add an annotation in the code so that fuzzing will explore different values of the pair *(a, b)*. Besides, the value of a comparison condition is more sensitive than the binary results, i.e., "examined" or "not examined" [37, 38, 65, 98, 137]. For instance, if the value of *(x[3] - 0x44)* in Figure 3(a) can be known, a fuzzer can select seeds that are more likely to satisfy the condition *if (x[3] == 0x44)*.

3.5.2 Diverse Fitness. Fitness is not limited to code coverage. In fact, code coverage is not always practical or the best feedback for fuzzing campaigns. Therefore, researchers utilize different types of feedback for different applications or defects. If code coverage is not available, an intuitive solution is to retain seeds based on execution outputs, such as the legality of execution results [141] or state machine of protocol implementations [154]. Because fuzzing can be utilized to detect defects of various applications, different types of fitness are designed for specific applications [1, 82, 148] or defects [194]. The following summarizes the diverse kinds of fitness.

- *Legality of execution result.* An object-oriented program (e.g., Java) consists of a sequence of method calls, and the execution result either is legal or throws exceptions. Fuzzing generates and obtains new method call sequences that can explore more new and legal object states [141].
- *State machine of protocol implementations.* A state machine consists of states and inputs that transform its states [161]. Due to the complexity of protocols, fuzzers usually infer the state machine by gradually adding new states into the state machine [55, 61, 64, 69, 154]. The state machine starts from a seed (i.e., an initial state machine), and fuzzers mutate the current states of state machine to explore new states. The vulnerabilities are analyzed based on the state machine, which intends to search for vulnerable transitions [55].
- *Safety policy of robotic vehicles.* The safety policies are the requirements for a robotic vehicle's physical or functional safety, such as the maximum temperature of a vehicle's engine [82]. When an input is closer to violations of policies, the input is retained for later mutation.

- *Fitness for deep learning systems.* The fuzzing of **deep learning systems (DLSs)** aims to improve the robustness and reliability of them [68, 115, 148]. To achieve this, fuzzers design different types of fitness, such as neuron coverage for discovering corner cases [148], loss function for augmenting the training data [68], or operator-level coverage for exploring deep learning inference engines (i.e., frameworks and libraries) [115].
- *Validation log of Android SmartTVs.* The validation logs are the messages of executing Android SmartTVs [1]. The validation logs are utilized to infer valid inputs and extract input boundaries. The valid inputs provide fuzzing with effective seeds, and the input boundaries reduce the search space of inputs.
- *Behavioral asymmetry of differential testing.* For differential testing, bugs are discovered via observing the discrepancies in the behaviors of different implementations, which have the same functionality, on the same input. The behavioral asymmetries indicate how discrepant the various implementations are. Fuzzing aims to generate test cases that can discover more discrepancies [138, 151].
- *Alias coverage for data race.* The alias coverage is designed to detect data races in kernel file systems [194]. A data race is a concurrent bug in which two threads access a shared memory location without proper synchronization. Therefore, the alias coverage tracks pairs of memory accesses that may interleave with each other.
- *Dangerous locations for bugs.* The dangerous locations are the code regions that have higher probability to trigger a bug. Therefore, fuzzers can steer fuzzing resource toward those locations to improve the effectiveness and efficiency of fuzzing. For concurrency bugs, the dangerous locations usually refer to code regions that result in atomicity violations [97], data races [84, 167], or suspicious interleavings [35]. For non-concurrency bugs, dangerous locations may be obtained by patch testing [122], crash reproduction [155], static analysis report [48], or information flow detection [123]. Moreover, the dangerous locations may be the memory accesses [84, 182, 188], sanitizer checks [40, 140], or commit history [214].

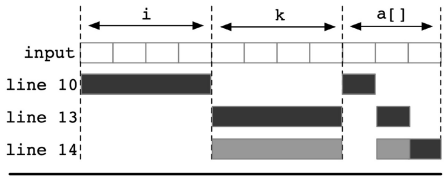
3.6 Evaluation Theory

The evaluation of fuzzing is usually conducted separately from the detection stage. However, we consider the evaluation as a part of the fuzzing processes because a proper evaluation can help improve the performance of fuzzing [215]. A proper evaluation includes an effective experimental corpus [215], fair evaluation environment/platform [30, 104, 126], reasonable fuzzing time [17, 20], and comprehensive comparison metrics [96, 104]. Although these research works have made efforts on proper evaluations, it is still an open question about how to evaluate techniques (i.e., the fuzzing algorithms) instead of implementations (i.e., the code that implements the algorithms) [18]. A widely used solution is to evaluate fuzzers based on a statistical test, which offers a likelihood that reflects the differences among fuzzing techniques [96].

Gap 1: The fuzzing theories narrow down the gaps between input space and defect space. Fuzzing theories formulate fuzzing processes based on program behaviors, such as bug arrival, state transition, and state discovery. Most theories formulate the process of seed schedule. Almost all fuzzers formulate seed retention based on genetic algorithm.

4 SEARCH SPACE OF INPUTS

As discussed in Section 3, fuzzers utilize optimization solutions to solve the search problem of input generation, which optimizes the search process in the input space. If a fuzzer can reduce the input space, it will also improve the performance of fuzzing. To achieve that, fuzzers group the




```
1  int main(int argc, char *argv[]){
2      int i, k;
3      char a[3];
4      i = get_int();
5      k = get_int();
6
7      for(int j = 0; j < 3; j++){
8        a[j] = get_char();}
9
10     if (11 < a[0] + i < 22)
11     {…}
12
13     if (a[1] < 33 + k) {
14       if (a[2] == 33) {
15         bug();}}
16 }
```

(a) Byte relation

```
1  char *parser (cJSON *item, char *s) {
2    if(!s) return 0; // fail
3    /* type: cJSON_NULL */
4    if(!strncmp(s, "null", 4)){…}
5    /* type: cJON_False */
6    if(!strncmp(s, "false", 5)){…}
7    /* type: cJSON_True */
8    if(!strncmp(s, "true", 4)){…}
9    /* type: cJSON_String */
10   if(*s == '\"')
11     return parse_string(item, s);
12   /* type: cJSON_Number */
13   if(*s == '-' ||
14         (*s >= '0' && *s <= '9'))
15     return parse_number(item, s);
16   /* type: cJSON_Array */
17   if(*s == '[')
18     return parse_array(item, s);
19   /* type: cJSON_Object */
20   if(*s == '{')
21     return parse_object(item, s);
22   return 0; // fail
23 }
```

(b) Syntax parser (cJSON)

Fig. 4. Search space of input. The input space can be reduced by grouping related bytes. The relationships between bytes can be obtained based on path constraints or input specifications.

related bytes in an input and apply specific mutators to each group. Suppose that an input includes $a \times b$ bytes and is equally divided into a parts; then instead of $256^{a \times b}$, the search space of fuzzing is $a \times 256^b$ when resolving a specific path constraint. The related bytes can be the ones constructing the same data structure [16, 201], influencing the same path constraint [37, 38, 65, 67, 157, 160, 187], or conforming to the same part of a grammar [78, 115, 120, 136, 181, 197, 212]. The mutators include byte mutation (e.g., bitflip, byte deletion, and byte insertion) [23, 206] and chunk mutation (e.g., chunk replacement, chunk deletion, and chunk insertion) [74, 77, 78, 181, 197].

As shown in Figure 4(a), the whole input space can be divided into three parts, each of which is related to the variables i, k, and array $a[]$, respectively. On the other hand, when solving a path constraint, a method focusing on the related bytes also reduces the search space. For example, when solving the path constraint of line 14 in Figure 4(a), the search space is only 1 byte if the constraint of line 13 is satisfied. A special kind of input is the highly structured input, which is utilized for applications such as protocol implementations, **Document Object Model (DOM)** engines, and compilers. As shown in Figure 4(b), the cJSON parser requires that the segments of an input start with some specific characters. If an input violates the requirement, the input is not allowed to examine the functionalities guarded by the parser. Table 2 describes the approaches that reduce the search space and the relations utilized for grouping input bytes. The fuzzers in Table 2 are selected because their main contributions are the reduction of input space.

4.1 Byte-constraint Relation

For most path constraints, they are influenced by only a small part of an input. If a fuzzer only mutates the related bytes, the performance of fuzzing can be significantly improved by reducing the search space of inputs. For instance, fuzzing may generate 256^{11} inputs to satisfy the condition *if (a[2] == 33)* in Figure 4(a) with a random mutation. However, if $pos[10]$ is known to be the only byte that influences the value of $a[2]$, we can only mutate the byte $pos[10]$ to pass the condition. As a result, only 256 inputs at most are required to resolve the path constraint.

Table 2. Input Space

Year	Fuzzer	Input Space Reduced By	Relation	Target App	Input		Runtime Info.
					Muta. -based	Gene. -based	
2009	BuzzFuzz [67]	dynamic taint analysis	byte-constraint	general	✓		○
2010	TaintScope [187]	dynamic taint analysis	byte-constraint	general	✓		○
2010	FLAX [160]	dynamic taint analysis	input-web operation	web apps	✓		●
2012	LangFuzz [80]	fragment	chunk-chunk	JavaScript engine		✓	●
2015	MutaGen [92]	encoding function	byte-instruction	file processor	✓		●
2016	Driller [177]	concolic execution	input-path	general	✓		○
2016	MoWF [153]	input model	chunk-chunk	file processor	✓		●+○
2016	TLS-Attacker [173]	framework integration	chunk-chunk	protocol	✓		●
2017	Steelix [103]	relation inference	byte-constraint	general	✓		◐
2017	Skyfire [185]	fragment	chunk-chunk	file processor		✓	●
2017	Learn&Fuzz [74]	machine learning	chunk-chunk	file processor		✓	●
2017	GLADE [12]	grammar synthesis	chunk-chunk	file processor		✓	●
2017	DIFUZE [53]	dependency inference	chunk-chunk	kernel driver		✓	●
2017	IMF [77]	dependency inference	chunk-chunk	kernel		✓	●
2018	IoTFuzzer [36]	encoding function	byte-function	IoT	✓		●
2018	FairFuzz [101]	relation inference	byte-coverage	general	✓		◐
2018	QSYM [204]	concolic execution	input-path	general	✓		○
2018	T-Fuzz [150]	program transformation	input-bug path	general	✓		◐+○
2018	ContractFuzzer [86]	dependency inference	chunk-chunk	smart contract	✓		◐
2019	RESTler [8]	dependency inference	chunk-chunk	cloud service	✓		●
2019	NEUZZ [172]	neural network	byte-coverage	general	✓		◐
2019	SLF [200]	relation inference	byte-constraint	general	✓		◐
2019	NAUTILUS [5]	input model	chunk-chunk	file processor	✓		◐
2019	CodeAlchemist [78]	fragment	chunk-chunk	JavaScript engine	✓		●
2019	ILF [79]	machine learning	chunk-chunk	smart contract		✓	◐+○
2019	GRIMOIRE [16]	format inference	byte-coverage	file processor	✓		◐
2019	ProFuzzer [201]	format inference	byte-coverage	file processor	✓		◐
2020	GREYONE [65]	relation inference	byte-variable	general	✓		◐
2020	Pangolin [81]	concolic execution	input-path(s)	general	✓		○
2020	SQUIRREL [212]	IR	chunk-chunk	DBMS	✓		◐
2020	FreeDOM [197]	IR	chunk-chunk	DOM	✓		●
2020	Montage [99]	fragment	chunk-chunk	JavaScript engine	✓		●
2020	HFL [95]	dependency inference	chunk-chunk	kernel	✓		○
2020	FANS [110]	dependency inference	chunk-chunk	Android		✓	●
2021	POLYGLOT [43]	IR	chunk-chunk	language processor	✓		◐
2021	DIANE [159]	encoding function	byte-function	IoT	✓		●
2021	Facovado [59]	dependency inference	chunk-chunk	JavaScript engine		✓	●

Fuzzers reduce the input space by grouping the related bytes. The groups of bytes are obtained based on certain relations. ○: Whitebox Fuzzing; ◐: Greybox Fuzzing; ●: Blackbox Fuzzing.

After obtaining the byte-constraint relation, a naive mutation scheme is to mutate the related bytes randomly [67, 157, 187]. A more uniform approach is to set the values of a byte from 0 to 255, respectively [172]. However, these two solutions are ineffective because they have no knowledge about the quality of inputs. If the inferring process of the byte relation can obtain values of comparison instructions in a program, fuzzing can mutate related bytes and select inputs that make progress in passing path constraints. An input makes progress if it matches more bytes of a path constraint [65, 103]. Moreover, fuzzing can utilize a gradient descent algorithm to mutate related bytes and gradually solve path constraints [37, 38].

4.1.1 Dynamic Taint Analysis. **Dynamic taint analysis (DTA)** [51, 135] is a common technique for building the relationships between input bytes and path constraints. DTA marks certain data in inputs and propagates the labels during executions. If a variable in the program obtains a label, the variable is connected to the data with the label [51]. Fuzzers [37, 38, 67, 157, 160, 187] utilize

DTA to build relationships between input bytes and security-sensitive points (e.g., system/library calls or conditional jumps).

4.1.2 Relation Inference. DTA requires heavy manual effort and can also result in inaccurate relations due to the implicit dataflows [65]. Because fuzzing examines target programs with numerous test cases, a lightweight solution is to infer the byte relation at runtime. One solution is to observe if the mutation of a byte changes the values of a variable [65], the results of a comparison instruction [7, 103, 200], or the hits of a branch [101]. If it does, the byte is linked to the variable, the comparison instruction, or the branch, respectively. Another approach for inference is to approximately build connections between input bytes and branch behaviors based on deep learning [172].

4.2 Concolic Execution

Concolic execution (also known as dynamic symbolic execution) regards the program variables as symbolic variables, tracks path constraints, and uses constraint solvers to generate a concrete input for a specific path [165]. In other words, concolic execution reduces the search space via directly solving the path constraints. Techniques utilizing both symbolic execution and fuzzing are called hybrid fuzzing or whitebox fuzzing. Hybrid fuzzing [72, 73, 153] utilizes fuzzing to exercise execution paths in target programs and uses concolic execution to solve constraints in those execution paths. Fuzzing suffers from solving path constraints due to its random nature. Although concolic execution is effective in solving path constraints, it is time-consuming to apply concolic execution for every execution path. Therefore, when fuzzing can no longer explore more states, concolic execution is utilized to solve constraints that fuzzing cannot satisfy [177].

One improvement of the hybrid fuzzing is to prioritize the most difficult path for concolic execution to solve [210]. In addition to the path selection, the performance of hybrid fuzzing can be improved via developing approximate constraint solvers. Generally, the **satisfiability modulo theory (SMT)** solvers (e.g., MathSAT5 [49] or Z3 [54]) are utilized to solve the path constraints. However, the SMT solvers have challenges in solving constraints due to the complex constraints or path explosion [177]. To mitigate the challenge, the constraint solver only symbolizes the path constraints that are influenced by inputs [46, 204]. Another improvement is based on the observation that many path constraints tend to be linear or monotonic [47]. Thus, the constraint solver performs in a greybox manner; e.g., it utilizes linear functions to approximate constraint behaviors. Interestingly, researchers have paid attention to solving path constraints via fuzzing [24, 108]. For example, JFS [108] translates SMT formulas to a program and utilizes coverage-guided fuzzing to explore the program. The SMT formulas are solved when a fuzzing-generated input reaches specific locations of the corresponding program.

Constraint solvers can also be improved based on features of targets. In terms of nested conditions (e.g., lines 13–15 in Figure 4(a)), Pangolin [81] proposes polyhedral path abstraction to resolve nested path constraints. The polyhedral path abstraction retains the solution space of historical constraints and reuses the solution space to satisfy the reachability for the current path constraints. For example, in order to solve the constraint in line *14* of Figure 4(a), the input has to satisfy conditions in line *13* first. In order to utilize hybrid fuzzing in programs that require highly structured inputs, Godefroid et al. [71] first symbolize tokens in grammar as symbolic variables. Then, they use a context-free constraint solver to generate new inputs.

4.3 Program Transformation

For fuzzing, the program transformation aims to remove sanity checks that prevent fuzzing from discovering more execution states. By removing those checks, fuzzing can explore deep code in

target programs and expose potential bugs [150]. The removal introduces many false positives of bug locations, which can be further verified by symbolic execution. Thus, program transformation reduces the search space by focusing on inputs that may potentially trigger bugs.

4.4 Input Model

Many applications require highly structured inputs, such as protocol implementations [3], DOM engines [197], JavaScript engines [78], PDF readers [74], system calls [77], and compilers [43]. An input model specifies the rules of constructing a highly structured input, including the structure, format, and data constraints of inputs. In order to generate inputs that satisfy the specifications, the generation process is restricted to specific operations. If an input violates the syntax or semantics of target programs, the input will be rejected by the program at an early stage. In other words, the input space is subjected to the input model.

4.4.1 Accessible Models or Tools. Many fuzzers generate valid input files based on accessible input models [3, 56, 57, 100, 115, 136, 147, 181] or existing tools [75, 153]. Input generation based on bare specifications requires heavy engineering effort. Moreover, it is error-prone because the specifications are complex to parse. The cJSON parser of Figure 4(b) is an implementation of the JSON specification, which is error-prone, although simple, due to the complex parse of different data types. Therefore, the research community has open-sourced some tools for highly structured inputs, e.g., QuickCheck [50] and ANTLR [146]. For example, NAUTILUS [5] and Superion [186] generate new inputs based on ANTLR. Then, both NAUTILUS and Superion use code coverage to optimize the mutation process. In some scenarios, the input model can simply be the type that the generated data will conform to (e.g., types of API arguments or physical signals) [1, 41, 70, 179]. For instance, the data for actuators of **cyber-physical systems (CPSs)** could be the binary values *on* or *off* [41].

4.4.2 Integration of Implementations. Another promising approach is to integrate fuzzing with the implementations of target applications [64, 89, 173]. Such integration allows fuzzing to examine the desired properties of target applications by customizing the process of input generation. For example, TLS-Attacker [173] creates a framework that can mutate inputs based on the type of each segment and manipulate the order of protocol messages. This framework includes a complete **Transport Layer Security (TLS)** protocol implementation.

4.4.3 Intermediate Representation. A more complex approach is to convert the input model into an **intermediate representation (IR)**. For highly structured inputs, mutations on the original input files are too complex to maintain the syntax and semantics. Therefore, researchers translate the original files into IR, which is simpler and more unified. Fuzzers mutate the IR and then translate the mutated IR back to the original input formats. This mutation strategy can maintain syntactic or semantic correctness and generate more diverse inputs. For instance, IR is utilized to test **database management systems (DBMSs)** [212], examine DOM engines [197], or fuzz different language processors (e.g., compilers or interpreters) [43].

4.5 Fragment Recombination

Based on input specifications, another solution for input generation is to generate new input files via fragment recombination. The basic idea of fragment recombination is to separate input files into many small chunks (i.e., fragments) and then generate a new input file via combining small chunks from different input files. Each fragment conforms to the specifications of inputs so that the recombined input files are syntactically correct. Ideally, the reassembled input file will exercise a new execution path or expose a new bug.

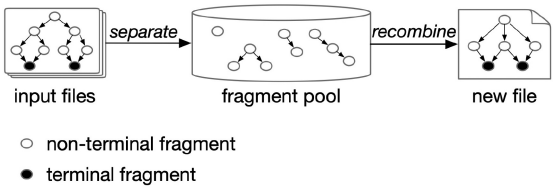


Fig. 5. Fragment recombination. Input files are often parsed into trees (e.g., ASTs), and the fragments in the trees will be reco mbined for generating a new input file. Each fragment conforms to the specifications of inputs.

```
snippet 1
1-1 var f = function()
1-2       {return 30;};
1-3 var a = 20;
1-4 var b = f();

snippet 2
2-1 var f = function()
2-2       {return 30;};
2-3 var a = 20;
2-4 //error
2-5 var b = errf();
```

Fig. 6. Semantic error (JavaScript). Line 2–5 introduces semantic error because the method errf() is not defined.

As depicted in Figure 5, fuzzers first parse an input file into a tree (e.g., **abstract syntax tree (AST)**), which remains the syntactic correctness. In order to properly parse inputs, the input corpus is required to be valid [78, 180, 185, 199]. One way to gather a valid input corpus is to download files from the Internet [26]. Besides validity, fuzzers also collect problematic inputs, which have caused invalid behaviors before, for the input corpus [80, 99, 145]. The basic assumption is that a new bug may still exist in or near the locations where an input has already discovered a bug [80]. The problematic inputs have already exercised complex execution paths that cause invalid behaviors. Therefore, the recombination of fragments may exercise the same or neighbor complex paths, which helps fuzzing explore deep code lines. In the second stage, the input files are separated into many fragments, which are stored in a fragment pool. Because fuzzers parse inputs into ASTs, the fragments can be sub-trees that contain non-terminals. When recombining fragments, a newly generated input file is required to be syntactically correct. Therefore, fuzzers recombine syntactically compatible fragments based on random selection [80, 120, 199], genetic algorithm [180], or machine learning [185]. In addition to syntactic correctness, semantic correctness also has a significant influence on the effectiveness of fuzzing. For instance, in order to generate syntactically and semantically correct JavaScript inputs, CodeAlchemist [78] tags fragments with assembly constraints. That is, different fragments are combined only when the constraints are satisfied.

4.6 Format Inference

If input models are not accessible, inferring the format of inputs is a promising solution to generate valid inputs. Moreover, one input model can only generate inputs with a specific format. In order to support more formats of inputs, developers have to utilize new input models and select the corresponding input model when generating inputs. Therefore, format inference is more scalable than model-based approaches.

4.6.1 Corpus-based. In order to infer the formats, a straightforward approach is to learn from a valid input corpus. Due to the lack of input models, researchers build end-to-end deep learning models to surrogate the input models. **Recurrent neural network (RNN)** [127] is a preferable deep learning model for fuzzers to generate structured inputs [74, 79, 111, 112]. However, the surrogate solution may suffer from generating invalid inputs. For example, the highest rate for DeepFuzz [112] to generate valid syntax inputs is only 82.63%. To improve the rate of generating valid inputs, the training data needs to be refined accordingly. For example, when generating PDF files, the training data is composed of sequences of PDF objects instead of textual data [74]. As for

smart contracts, the training data is about the sequences of transactions [79]. Similarly, LipFuzzer [208] trains adversarial linguistic models to generate voice commands, where the training data is presented through a linguistic structure. Besides, fuzzing can synthesize a context-free grammar (e.g., regular properties such as repetitions and alternations) based on a valid input corpus [12]. The synthesized grammar is then utilized for generating highly structured inputs.

4.6.2 Coverage-based. The corpus-based solution requires the training data to have comprehensive coverage of input specification, which may not be practical [200]. Besides, it does not use knowledge from internal execution states (e.g., code coverage), which may cause low code coverage. Essentially, the format of inputs indicates the relationships among different bytes in inputs. Therefore, based on the code coverage, fuzzers infer the byte-to-byte relations to boost fuzzing [16, 201]. For instance, GRIMOIRE [16] uses code coverage to infer the format required by target programs. It aims to identify the format boundary of an input. Specifically, it changes some bytes in an input and checks whether the changes result in different code coverage. If the code coverage remains the same, the positions where the bytes are mutated can be randomly mutated. Otherwise, the positions are required to be carefully mutated. ProFuzzer [201] first defines six data types that cover most input content. Then, based on the distribution of edge coverage, it infers the types of each byte and merges the consecutive bytes that belong to the same type.

4.6.3 Encoding Function. Different from all the aforementioned approaches that focus on inputs, some fuzzers search for the code regions that encode the formats of inputs [36, 92, 159]. Because such code regions correlate to the generation of well-structured inputs, fuzzers perform mutation before encoding the formats. Although the source code of PUTs may not be accessible, their corresponding implementations that generate well-structured inputs are often accessible [92]. For example, the community open-sources some tools for generating highly structured inputs [50, 146] (Section 4.4.1). As for the IoT devices, most of them are controlled through companion applications, which form messages communicating with target devices [36]. By locating the code regions related to encoding formats, mutations can be operated on the arguments of functions [36, 159] or the instructions calculating the formats [92]. For instance, IoTFuzzer [36] hooks such functions and mutates the data for arguments of those functions.

4.7 Dependency Inference

The format inference mainly intends to satisfy the syntactic requirements, which may still generate inputs with incorrect data dependency. For instance, in the code *snippet 2* of Figure 6, an error occurs in line *2-5* because the method *errf()* is not defined. Many applications require correct data dependency in inputs, which usually consist of sequences of statements. The sequences include system calls for kernel code [77, 95, 142], objects for processors of object-oriented programs [59, 117], **application programming interfaces (APIs)** for services/libraries [8, 83, 110], or **application binary interfaces (ABIs)** for smart contracts [86]. On the one hand, most of these applications require definition/declaration before using the data in inputs, as depicted in Figure 6. On the other hand, the outputs of executing some statements are the data of arguments for some other statements.

4.7.1 Documents or Source Code. The data dependency of sequences is usually inferred via static analysis. Because many applications have documents or source code describing their interfaces, researchers infer the data dependency based on those resources [8, 53, 59, 86, 110, 117]. The resources contain information about how to use an interface and the prerequisites for the interface. When fuzzing generates inputs including an interface, fuzzing is also required to generate the prerequisites for the interface. Otherwise, the generated input will be refused at an early stage.

However, the static analysis introduces high false positives and misses dependencies of interfaces. Therefore, when having access to the source code of PUTs, a better solution is to combine static analysis and dynamic analysis [95].

4.7.2 Real-world Programs. Many real-world programs implement code lines to call interfaces, which have already considered the data dependency of interfaces. Therefore, fuzzing can synthesize new programs that call the interfaces based on program slicing of the real-world programs [10]. Besides, the data dependency can be inferred by analyzing execution logs of those real-world programs [77, 83, 142]. The execution logs explicitly contain the ordering information of interfaces, i.e., the information about which interface is executed first. Moreover, the execution logs implicitly contain the information of argument dependency between interfaces. In order to obtain explicit and implicit information, fuzzing hooks each interface when executing a real-world program and records desired information.

Gap 2: The reduction of input space relies on grouping the input bytes that are syntactically and/or semantically related. The advantage of grouping bytes is the improvement of efficiency in exploring more execution states. That is, fuzzing is more likely to satisfy path constraints so that it explores deep code regions guarded by these constraints.

5 AUTOMATION

Automatic execution is the basis of applying fuzzing theory and input space reduction approaches. Fuzzing repeatedly executes PUTs and monitors executions for exceptions, and the exceptions are further verified about whether they are bugs or not. Therefore, in order to succeed in fuzzing, the first requirement is to run PUTs automatically and repeatedly. Most fuzzers have already succeeded in testing command-line software, but they cannot be directly utilized for other targets, such as hardware or polyglot software [18]. The second requirement is the automatic indicator of potential bugs. Currently, fuzzing uses crashes as a sign of potential bugs. Yet, many bugs are not manifested as crashes, such as data races. Because the nature of fuzzing is to test targets repeatedly, the third requirement is the high execution speed of fuzzing. A higher execution speed indicates that more test cases are examined in the same time budget, which offers higher opportunity to find defects.

5.1 Automatic Execution of PUTs

Fuzzing is utilized for diverse applications, which require different engineering efforts to automatize the fuzzing processes. This section introduces several kinds of applications where fuzzing has successfully automatized the testing.

5.1.1 Command-line Programs. Fuzzing has achieved great success in testing command-line programs (e.g., the "general" applications in Tables 1 and 2). It runs PUTs in a sub-process and feeds the PUTs with the required options and inputs [23, 206, 214]. In order to improve the execution speed, fuzzing does not replay all steps for executing a PUT. Instead, it clones a child process so that it skips pre-processing steps, such as loading the program file into memory [206, 216]. Usually, fuzzing takes only one command option for all fuzzing campaigns; i.e., all generated inputs are executed based on one option. Because different options indicate different code coverage, a thorough test needs to enumerate all command options during fuzzing. An efficient scheme is that, if an input is invalid for one option, fuzzing skips testing all the rest of the options [176]. An important observation for this scheme is that if an input is invalid for one option, the input will fail all the other options.

5.1.2 Deep Learning Systems. The fuzzing of DLSs is similar to the test of command-line programs. DLSs are tested by fuzzing-generated inputs, and fuzzing intends to generate inputs for better fitness [68, 115, 148]. The inputs are training data, testing data, or even deep learning models based on different targets. On the other hand, the fitness can be neuron coverage, loss function, or operator-level coverage, as described in Section 3.5.2. As for testing DLSs, fuzzing will not only detect defects [115] but also examine the robustness of models [68, 148].

5.1.3 Operating System Kernels. The **Operating System (OS)** kernels are complex compared to general command-line programs. Kernels include many interrupts and kernel threads, resulting in non-deterministic execution states. In order to fuzz the kernels in the way as command-line programs, fuzzing utilizes a hypervisor (e.g., QEMU or KVM) to run a target kernel [143, 164]. Meanwhile, the code coverage is obtained via Intel's **Processor Trace (PT)** technology. Although this approach can fuzz various target kernels with feedback, it still requires manually constructing syntactically and semantically correct inputs. Because the inputs for the kernels include file system images or a sequence of syscalls, fuzzers can test kernels in a more lightweight manner. That is, after the data dependency of syscalls is analyzed or inferred (Section 4.7), fuzzers generate sequences of syscalls to run on target kernels [53, 77, 95, 142, 183, 194, 196]. Then, fuzzers monitor whether sequences of syscalls incur system panics that indicate potential bugs in the target kernel. Another way to fuzz OS kernels is to emulate the external devices. Because the kernel communicates with the emulated devices, fuzzers can generate inputs to test the drivers in kernels [149].

5.1.4 Cyber-Physical Systems. CPSs include two major components that are tightly integrated, i.e., the computational elements and the physical processes [41]. A widely used computational element is the **Programmable Logic Controller (PLC)**, which controls actuators to manage the physical processes and receives inputs from sensors. Thus, when fuzzing CPSs, fuzzers can replace PLCs and directly send numerous commands to actuators through the network [41]. Another way to fuzz CPSs is to examine the control applications and the runtime of PLCs [179]. However, the PLC binaries cannot be fuzzed in the same way as fuzzing command-line programs. Because PLC applications have various binary formats and complex communication with physical components, the automation of these applications varies. Based on the analysis of PLC binaries and their development platforms (e.g., Codesys), it is possible to automatically fuzz PLC binaries when running them on PLC devices [179].

5.1.5 Internet of Things. The automation of fuzzing IoT devices includes emulation [34, 205, 211] and network-level test [36, 63, 159]. The emulators [34, 205] can execute programs, which originally run on IoT firmware, without the corresponding hardware. With the help of emulators, fuzzers run target programs in a greybox manner [211]. On the other hand, the network-level fuzzing examines IoT devices in a blackbox manner. Because IoT devices can communicate with the outside world through networks, fuzzers automatically send messages (requests) to the IoT devices and wait for the execution results from IoT devices (responses) [36, 63, 159]. By categorizing the responses, the fitness is the number of categories; i.e., the purpose is to explore more categories [63].

5.1.6 Applications with Graphical User Interface. The execution speed of applications with **Graphical User Interface (GUI)** is much slower than command-line programs [91]. Since the execution speed is one of the keys for the success of fuzzing, the automation of GUI applications often replaces the GUI with a faster approach and executes targets in the command-line manner [91, 106, 121]. For instance, fuzzers can model the interactions of user interfaces so that they generate event sequences for Android applications [106, 121]. Moreover, fuzzers can also utilize a harness, which prepares the execution context, to directly invoke target functions in GUIs [91].

```
1 char str[10];
2 char buf[12];
3 // buffer overflow
4 memcpy(str, buf, sizeof(buf));
```

(a) Spatial safety violation.

```
1 struct A {void (*func)(void); };
2 struct A *p = (struct A *)malloc(sizeof(struct A));
3 free(p);
4 …
5 p->func(); // Use-after-free
```

(b) Temporal safety violation.

Fig. 7. Memory violation bugs (adapted from [175]).

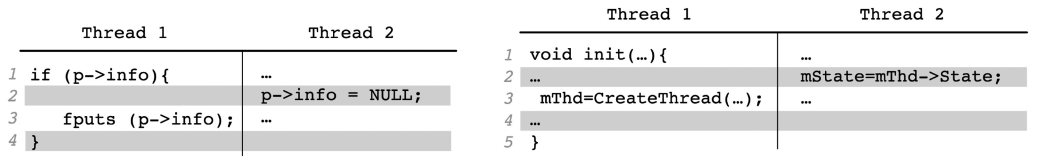
5.1.7 Applications with Network. Some applications receive inputs (messages) through the network, such as smart contracts [79, 86, 137, 193], protocol implementations [55, 61, 64, 69, 154], cloud services [8], Android Native System Services [1, 110], or robotic vehicles [82]. Therefore, an input can be generated locally, and the execution of the target applications can be performed remotely. The efficiency of the automatic testing relies on the quality of generated inputs as well as the fitness that reflects the execution states. For instance, the inputs for smart contracts are sequences of contract transactions, i.e., messages between different accounts. When receiving the transactions, functions in smart contracts are executed on their infrastructure of the blockchain [79, 86, 137, 193].

5.2 Automatic Detection of Bugs

Many security bugs can be exploited to control over systems, leak private data, or break down servers [131]. The challenge to detect bugs is that bugs are unpredictable. Specifically, a detector does not know the bug locations, and even does not know whether a bug exists in a target program before testing it. Therefore, it is critical to record a potential bug during fuzzing automatically. Usually, the indicators are crashes of program execution, but some other indicators are designed based on bug patterns. Although bug patterns can only be utilized to expose specific types of bugs, it is highly effective if such types of bugs exist in target programs [143]. This section mainly introduces six types of bugs that are successfully detected by fuzzing. They are memory-violation bugs, concurrency bugs, algorithm complexity, Spectre-type bugs, side channels, and integer bugs.

5.2.1 Memory-violation Bugs. Memory-violation bugs are among the oldest and most severe security bugs [131, 175, 178]. A program is memory safe if pointers are restricted to accessing intended referents. Memory safety violations can be classified into two categories, i.e., spatial safety violations and temporal safety violations [175]. A spatial safety violation gets access to memory that is out of bounds, while a temporal safety violation accesses an invalid referent. For instance, Figure 7(a) is a spatial violation because the size of *buf* is larger than *str* (buffer overflow). On the other hand, Figure 7(b) is a temporal violation because the pointer *p* is used after it has been freed (use-after-free). Although many approaches have been proposed to mitigate the influence of memory violations, most of them are not used in practice due to the disadvantages such as performance overhead, low compatibility, and low robustness [178].

Buffer overflow is one kind of memory-violation bug and writes bytes out of bounds (e.g., example of Figure 7(a)). Dowser [76] argues that typical buffer overflows are caused mainly by accessing an array in a loop. In order to detect buffer overflows in loops, Dowser ranks instructions that access buffers in loops and prioritizes inputs that exercise higher-ranking accesses. It then uses taint analysis and concolic execution to solve path constraints for the selected inputs. Since Dowser focuses on arrays in loops, only a small number of instructions are required to be instrumented. Such focus improves the execution speed of both taint analysis and concolic execution. The **use-after-free (UaF)** bug is another type of memory violation, i.e., temporal safety violation. As shown in Figure 7(b), an UaF bug consists of at least three statements: (1) allocating memory to a pointer,



(a) Atomicity-violation bug. (b) Order-violation bug.

Fig. 8. Non-deadlock concurrency bugs (adapted from [114]).

(2) releasing the corresponding memory, and (3) reusing the pointer. This bug pattern motivates UAFL [184] to generate inputs that can gradually cover an entire sequence of a potential UaF. The potential UaF sequences are obtained by static typestate analysis based on the bug pattern.

5.2.2 Concurrency Bugs. Another severe security bug is the concurrency bug, which occurs when concurrent programs run without proper synchronization or order. Generally, concurrency bugs are categorized into deadlock bugs and non-deadlock bugs [114]. A deadlock bug occurs when operations in a program wait for each other to release resources (e.g., locks). The non-deadlock concurrency bugs mainly include atomicity-violation bugs and order-violation bugs [114]. An atomicity-violation bug violates the desired serializability of a certain code region. For example, in Thread 1 of Figure 8(a), line *3* is required to be executed after line *1* so that *fputs()* is called with a valid argument. However, line *2* in Thread 2 is executed before line *3* in Thread 1. Thus, *p->info* is set to *NULL* before *fputs()* is called, which incurs an error. On the other hand, an order-violation bug occurs when two (or more) memory locations are accessed with an incorrect order. For instance, in Figure 8(b), *mThd->State* in Thread 2 is executed before the initialization of *mThd* in Thread 1, which incurs an error called the use of uninitialized variables. Concurrency bugs can also result in memory violations, such as use-after-free and double-free [28].

One solution to discover deadlocks is to detect cycles on a lock order graph, of which each node represents a lock [4]. If a cycle exists in the graph, a potential deadlock is detected [90]. In order to improve the efficiency and scalability of the cycle detection, MagicFuzzer [27] iteratively removes locks in the graph if those locks are not included in any cycle. MagicFuzzer then checks the remaining cycles based on a random scheduler. As for atomicity violations, ATOMFUZZER [144] observes a typical bug pattern that a lock inside an atomic block is repeatedly acquired and released by two threads. Specifically, if a thread t inside an atomic block is about to acquire a lock L that has been acquired and released by t before, ATOMFUZZER delays the execution of thread t and waits for another thread t' to acquire the lock L. It is an atomicity violation when the other thread t' acquires the lock L during the delay of thread t.

More generally, concurrency bugs occur due to the incorrect interleavings of threads. The challenge is that concurrent programs may have too many interleavings to examine each one (i.e., the problem of *state-explosion*). CalFuzzer [166] mitigates the state-explosion based on the fact that some interleavings are equivalent because they are from different execution orders of non-interacting instructions. The equivalence indicates that executions of them will result in the same state. CalFuzzer randomly selects a set of threads whose following instructions do not interact with each other and executes those instructions simultaneously. Therefore, CalFuzzer [166] can examine different interleavings more efficiently.

5.2.3 Algorithmic Complexity. **Algorithm complexity (AC)** vulnerabilities occur when the worst-case complexity of an algorithm significantly reduces the performance, which could result in **Denial-of-Service (DoS)** attacks. Figure 9 shows an example that when given different inputs to the argument *array*, the algorithm has different complexities. For example, if the value of *array*

```
function quicksort(array):
  smaller, equal, greater = [], [], []
  if len(array) <= 1:
    return
  pivot = array[0]
  for x in array:
    if x > pivot:
      greater.append(x)
    else if x == piviot:
      equal.append(x)
    else if x < pivot:
      smaller.append(x)
  quicksort(greater)
  quicksort(smaller)
  array = concat(smaller, equal, greater)
```

Fig. 9. Algorithm complexity (adapted from [152]).

```
1 i = input [0];
2 if (i < size) {
3       secret = foo[i];
4       baz = bar[secret];
5 }
```

Fig. 10. Spectre-type bug (adapted from [139]).

is [8, 5, 3, 7, 9], the algorithm will execute 37 **lines of code (LOC)**. On the other hand, if *array* is [1, 5, 6, 7, 9], it will cause the execution of 67 LOC. The increase of LOC requires more computing resources. Thus, the worst-case behavior could be exploited by attackers to launch a DoS attack. Therefore, SlowFuzz [152] detects AC bugs via guiding fuzzing toward executions that increase the number of executed instructions. Similarly, HotFuzz [15] detects AC bugs in Java methods via maximizing the resource consumption of individual methods. MemLock [44] detects AC bugs based on both metrics of edge coverage and memory consumption. It steers fuzzing toward inputs that can either discover more edges or consume more memories. The aforementioned fuzzers directly generate the **worst performance inputs (WPIs)** for AC bug discovery. On the contrary, Singularity [190] synthesizes programs for input generation based on the observation that these WPIs always follow a specific pattern.

5.2.4 Spectre-type Bugs. A Spectre-type bug is a microarchitectural attack that exploits mispredicted branch speculations to control memory accesses [139]. For instance, in Figure 10, an attacker can send several in-bound values for the variable *input*, which will train the branch predictor to speculate if the check in line *2* is always true. When the attacker sends an out-of-bound value for *input*, the predictor will incorrectly predict the branch behavior, and lines *3–4* are speculatively executed (i.e., they are executed without the check in line *2*). Since the *input* actually does not satisfy the check in line *2*, the execution of lines *3–4* results in buffer overread. Therefore, SpecFuzz [139] instruments target programs to simulate the speculative execution, which can forcefully execute the mispredicted code paths. Then, any invalid memory access in the mispredicted paths can be triggered.

5.2.5 Side Channels. Side-channel bugs leak secret information via observing non-functional behaviors of a system (e.g., execution time). For example, if a secret is the variable a in the statement "$if(a > 0)\{...\}else\{...\}$," one can observe the execution time of the then-branch and else-branch to tell whether the value of a is larger than zero. A special kind of side channels is called JIT-induced side channels, which is caused by **Just-In-Time (JIT)** optimization [25]. Similar to the aforementioned Spectre-type bugs, one can repeatedly run programs to train the JIT compiler to optimize the execution time of either the then-branch or the else-branch. Then, the execution time of the trained branch (e.g., the then-branch) and the untrained branch (e.g., the else-branch) will be biased enough to be observable. As a result, the secret value of the variable a is leaked.

5.2.6 Integer Bugs. Integer overflow/underflow bugs occur when the value of an arithmetic expression is out of the range that is determined by a machine type. On the other hand, integer conversion bugs occur when incorrectly converting one integer type to another integer type. In

order to detect integer bugs, SmartFuzz [132] adds specific constraints according to different integer bugs into the symbolic emulation. Then, the symbolic solver intends to generate concrete inputs that may trigger integer bugs.

5.3 Improvement of Execution Speed

Execution speed is critical to fuzzing as fuzzing runs numerous test cases in a limited time budget. A higher execution speed means that fuzzing can examine more test cases, which offers a higher opportunity to find defects. Therefore, researchers put many efforts into improving the execution speed of fuzzing, including binary analysis [58, 134], optimized execution processes [46, 133, 204, 216], and application-specified techniques [91, 162, 163, 174, 195, 196, 211].

5.3.1 Binary Analysis. As a pre-process, fuzzing mainly utilizes static instrumentation to obtain execution states because static instrumentation provides fuzzing with high execution speed [58, 134]. A widely used static analysis tool is LLVM [113], which instruments programs during compilation. When it comes to closed-source applications, fuzzers are restricted to binary analysis [124] because source code is not available. The problem is that binary instrumentation tools (e.g., Dyninst [128]), which succeed in many domains (e.g., emulation), suffer from runtime overhead when applied to fuzzing. To improve execution speed and provide fuzzing with compiler-level performance, RetroWrite [58] proposes to use static binary rewriting techniques based on reassembleable assembly. It focuses on instrumenting binaries of 64-bit **position independent code (PIC)** via utilizing PIC's relocation information to instrument assembler files. The performance overhead is reduced because RetroWrite can instrument inlined code snippets. Although fast, RetroWrite only supports 64-bit PIC binaries. In order to uphold both low runtime overhead and scalability, FIBRE [134] streamlines instrumentation through four IR-modifying phases. The four phases instrument programs via static rewriting, inlining, tracking register liveness, and considering various binary formats. The aforementioned rewriting techniques only rewrite binaries once, which may result in unsound binary rewriting, especially for stripped binaries [189]. To solve this problem, STOCHFUZZ [209] proposes the incremental and stochastic rewriting techniques based on the fact that fuzzing executes target programs repetitively. Specifically, STOCHFUZZ rewrites target binaries multiple times, and it gradually fixes problems introduced by previous rewriting results.

5.3.2 Execution Process. Execution speed can also be improved during the fuzzing campaign. UnTracer [133] observes that most test cases generated during fuzzing do not discover new coverage. This indicates that tracing all test cases, which AFL uses, incurs significant runtime overhead. Therefore, UnTracer only traces the coverage-increasing test cases to improve execution speed. This is accomplished by inserting interrupts at the beginning of basic blocks. When a block is examined, UnTracer removes the instrumentation at the block so that the execution will not be interrupted at the block in the future. Since block coverage loses information of execution states, CSI-Fuzz [216] utilizes edge coverage to improve UnTracer. Besides, Zeror [213] improves UnTracer via adaptively switching between Untracer-instrumented binary and AFL-instrumented binary. For hybrid fuzzing, concolic execution is utilized to resolve path constraints. Yet, the symbolic emulation in concolic execution is slow in formulating path constraints, which is the main factor that hybrid fuzzing suffers from scaling to real-world applications. QSYM [204] mitigates the performance bottleneck via removing some time-consuming components, such as IR translation and snapshot. Moreover, it collects and solves only a portion of path constraints. Although the concrete inputs generated by QSYM may not be the exact solution for path constraints, QSYM uses fuzzing to search for valid inputs via mutating those concrete inputs. Intriguer [46] observes that QSYM still suffers from performance bottleneck because QSYM solves many unnecessary

constraints. Intriguer then performs symbolic emulation for more relevant instructions, which is determined by dynamic taint analysis. In addition to instrumentation and hybrid fuzzing, another optimization is to improve execution speed in the parallel mode. Xu et al. [195] observe that AFL [206] significantly slows down when it runs on 120 cores in parallel. This motivates them to design new operating primitives to improve the execution speed.

5.3.3 Various Applications. Besides general applications, fuzzing is also used to detect defects in targets from different fields, such as IoT, OS kernels, and **virtual machine monitors (VMMs)**. Since these targets usually have special features, fuzzing is customized for the targets to perform testing in an efficient manner.

Although emulation is a promising approach to fuzz IoT firmware, full-system emulation suffers from low throughput. The runtime overhead of full-system emulation mainly comes from translating virtual addresses for memory accesses and emulating system calls. FIRM-AFL [211] mitigates the overhead via combining user-mode emulation and full-system emulation, and it mainly runs programs in the user-mode emulation. In order to fuzz VMMs (i.e., hypervisors), Schumilo et al. [162, 163] design a customized OS and a fast snapshot restoration mechanism to conduct fuzzing efficiently. As to the file systems, mutating a whole disk image degrades the fuzzing throughput significantly because an image is too large. To solve this challenge, JANUS [196] only mutates the metadata of a seed image; i.e., it exploits the properties of structured data. This solution reduces the search space of inputs, resulting in improvement of throughput. OS kernels can also be compromised through a peripheral device; i.e., vulnerabilities occur along the hardware-OS boundary. In order to detect the defects in device-driver communication, PeriScope [174] proposes to fuzz based on the kernel's page fault handling mechanism. Windows applications are different from Linux's because they heavily use graphical interfaces, and Windows lacks approaches to clone a process quickly. WINNIE [91] synthesizes a harness to run applications without graphical interfaces. Moreover, it implements *fork()* for Windows to clone processes efficiently. BigFuzz [207] transforms **data-intensive scalable computing (DISC)** applications to a semantically equivalent program, which is independent from the DISC framework. Because the DISC framework introduces long latency, the framework-independent execution significantly improves the execution speed.

Gap 3: The automatic execution of applications is based on the sophisticated understanding of those applications. When designing indicators for automatic records of security flaws, the properties of those flaws have to be investigated first.

6 DIRECTIONS OF FUTURE RESEARCH

Fuzzing has attracted much attention from the research community. It has discovered thousands of real-world bugs and severe vulnerabilities in recent decades. We summarize some directions for future research on fuzzing.

More sensitive fitness. Researchers have made many efforts in improving the efficiency and effectiveness of code coverage, especially the sensitivity of code coverage (Section 3.5.1). Recently, researchers realize that code coverage has its limitation in discovering complex bugs. Therefore, they extend code coverage via introducing information (e.g., dangerous code regions) that is obtained by analyzing bugs. Future works can analyze bugs and detect them based on features of bugs, especially analyzing the bugs that evade current fuzz testing.

More sophisticated fuzzing theories. Current fuzzing theories partially formulate fuzzing processes (Section 3). Most of the existing works intend to formulate the seed schedule, while much fewer works pay attention to other fuzzing processes. Due to the complication of fuzzing

processes, few of the existing works formulate the entire fuzzing process. It is non-trivial to mathematically formulate the entire fuzzing process. However, it is possible to formulate more than one fuzzing process, such as Game Theory, considering both seed schedule and byte schedule. A bigger picture is about the theoretical limitation of fuzzing (e.g., the limitation of greybox fuzzing). On the other hand, formulating fuzzing processes with multiple types of fitness is another way to build more sophisticated fuzzing theories. For example, future works may formulate fuzzing processes considering both the arrival of bugs and the state transitions.

Sound evaluation. A few works focus on the soundness of evaluation but no firm conclusion has been made (Section 3.6). These works only provide suggestions for a sound evaluation, such as time budget or evaluation metrics. More questions remain open to be answered. Should we use synthesized bugs or real-world bugs for the evaluation corpora? Is the statistical test the final answer to differentiate two fuzzing techniques? What is a reasonable time budget to terminate the fuzzing processes? How do we evaluate special target applications, such as hardware, when no comparison fuzzer exists?

Scalable input inference. The efficiency of fuzzing can be significantly improved when the format or data dependency is used during fuzzing (Sections 4.6 and 4.7). Static analysis is widely used for both format inference and data dependency inference. However, static analysis is application-specific; i.e., the implementations of inference approaches are required to consider the features of different applications. Currently, dynamic analysis focuses on format inference and few works make efforts in data dependency inference. Inference approaches with dynamic analysis are possible to be utilized for multiple applications; i.e., dynamic analysis is more scalable than static analysis. More research may focus on the inference of data dependency based on dynamic analysis.

Efficient mutation operators. Almost all fuzzers utilize fixed mutators during fuzzing. That is, fuzzers design some mutators in advance based on features of target applications and the mutators are not changed during fuzzing (Section 4). A few works intend to optimize the mutator schedule, but no one focuses on changeable mutators (Section 3.4). Is it possible to design evolved mutators that are changed during fuzzing to improve performance? Because the mutator schedule is tightly interacted with the byte schedule, it may be promising to design mutators considering the byte schedule. Moreover, mutators for highly structured inputs may have different attributions to general applications. Thus, the mutator schedule for highly structured inputs may also be worthy of being studied.

More types of applications. Fuzzing has achieved great success in detecting bugs in command-line programs. Researchers also make many efforts in fuzzing more types of applications (Section 5.1). Due to the complexity of different applications, fuzzing has its limitation in detecting more types of applications in practice. For example, a few works explore the possibility to fuzz cyber-physical systems, but the fuzzing capability is limited [41, 179]. Because the execution speed is critical for fuzzing, a potential direction for the hard-fuzzing applications is to improve their execution speed.

More types of bugs. Fuzzing has already successfully detected bugs such as memory-violation bugs, concurrency bugs, or algorithmic complexity bugs (Section 5.2). However, it has difficulty in detecting many other types of bugs, such as privilege escalation or logical bugs. The challenge is to design proper indicators for these bugs so that they are automatically recorded during fuzzing. Because such indicators reflect the features of corresponding bugs, the design of the indicators requires researchers to have a good understanding of both fuzzing and target bugs. For instance, programs will run without exceptions even when they trigger logical bugs. In order to design automatic indicators for logical bugs, it requires a profound understanding of the functional requirements that are used to develop the code.

REFERENCES

[1] Yousra Aafer, Wei You, Yi Sun, Yu Shi, Xiangyu Zhang, and Heng Yin. 2021. Android SmartTVs vulnerability discovery via log-guided fuzzing. In *30th USENIX Security Symposium (USENIX Security'21)*. 2759–2776.

[2] Humberto Abdelnur, Radu State, Obes Jorge Lucangeli, and Olivier Festor. 2010. *Spectral Fuzzing: Evaluation & Feedback*. Technical Report. https://hal.inria.fr/inria-00452015.

[3] Humberto J. Abdelnur, Radu State, and Olivier Festor. 2007. KiF: A stateful SIP fuzzer. In *Proceedings of the 1st International Conference on Principles, Systems and Applications of IP Telecommunications (Iptcomm'07)*. 47–56.

[4] Rahul Agarwal, Liqiang Wang, and Scott D. Stoller. 2005. Detecting potential deadlocks with static analysis and run-time monitoring. In *Haifa Verification Conference*. Springer, 191–207.

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

[6] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz. 2020. IJON: Exploring deep state spaces via fuzzing. In *IEEE Symposium on Security and Privacy (S&P'20)*. IEEE Computer Society, Los Alamitos, CA, 874–889.

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

[8] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. RESTler: Stateful REST API fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE'19)*. IEEE, 748–758.

[9] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. 2002. Finite-time analysis of the multiarmed bandit problem. *Machine Learning* 47, 2 (2002), 235–256.

[10] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. 2019. Fudge: Fuzz driver generation at scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'19)*. 975–985.

[11] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. 2006. SNOOZE: Toward a stateful network protocol fuzzer. In *International Conference on Information Security*. Springer, 343–358.

[12] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing program input grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'17)*. 95–110.

[13] Petra Berenbrink and Thomas Sauerwald. 2009. The weighted coupon collector's problem and applications. In *International Computing and Combinatorics Conference*. Springer, 449–458.

[14] Donald A. Berry and Bert Fristedt. 1985. *Bandit Problems: Sequential Allocation of Experiments (Monographs on Statistics and Applied Probability)*. London: Chapman and Hall, 5, 7 (1985), 71–87.

[15] William Blair, Andrea Mambretti, Sajjad Arshad, Michael Weissbacher, William Robertson, Engin Kirda, and Manuel Egele. 2020. HotFuzz: Discovering algorithmic denial-of-service vulnerabilities through guided micro-fuzzing. In *The Network and Distributed System Security Symposium (NDSS'20)*. 1–19.

[16] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. 2019. GRIMOIRE: Synthesizing structure while fuzzing. In *28th USENIX Security Symposium (USENIX Security'19)*. 1985–2002.

[17] Marcel Böhme. 2018. STADS: Software testing as species discovery. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 27, 2 (2018), 1–52.

[18] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. 2020. Fuzzing: Challenges and reflections. *IEEE Software* 38, 3 (2020), 79–86.

[19] Marcel Böhme and Brandon Falk. 2020. Fuzzing: On the exponential cost of vulnerability discovery. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'20)*. 713–724.

[20] Marcel Böhme, Danushka Liyanage, and Valentin Wüstholz. 2021. Estimating residual risk in greybox fuzzing. In *ACM European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'21)*. 230–241.

[21] Marcel Böhme, Valentin J. M. Manès, and Sang Kil Cha. 2020. Boosting fuzzer efficiency: An information theoretic perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'20)*. 678–689.

[22] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *The ACM Conference on Computer and Communications Security (CCS'17)*. ACM, 2329–2344.

[23] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as Markov chain. In *The ACM Conference on Computer and Communications Security (CCS'16)*. ACM, 1032–1043.

[24] Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. 2021. Fuzzing symbolic expressions. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE'21)*. IEEE, 711–722.

[25] Tegan Brennan, Seemanta Saha, and Tevfik Bultan. 2020. JVM fuzzing for JIT-induced side-channel detection. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE'20)*. 1011–1023.

[26] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. 2014. Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *IEEE Symposium on Security and Privacy (S&P'14)*. IEEE, 114–129.

[27] Yan Cai and W. K. Chan. 2012. MagicFuzzer: Scalable deadlock detection for large-scale applications. In *2012 34th International Conference on Software Engineering (ICSE'12)*. IEEE, 606–616.

[28] Yan Cai, Biyun Zhu, Ruijie Meng, Hao Yun, Liang He, Purui Su, and Bin Liang. 2019. Detecting concurrency memory corruption vulnerabilities. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'19)*. 706–717.

[29] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-adaptive mutational fuzzing. In *IEEE Symposium on Security and Privacy (S&P'15)*. IEEE, 725–741.

[30] Oliver Chang, Jonathan Metzman, Max Moroz, Martin Barbella, and Abhishek Arya. 2016. OSS-Fuzz: Continuous Fuzzing for Open Source Software. Retrieved January 19, 2021, from https://github.com/google/oss-fuzz.

[31] Anne Chao and Chun-Huo Chiu. 2016. Species richness: Estimation and comparison. *Wiley StatsRef: Statistics Reference Online*, 26.

[32] Anne Chao and Robert K. Colwell. 2017. Thirty years of progeny from Chao's inequality: Estimating and comparing richness with incidence data and incomplete sampling. *SORT-Statistics and Operations Research Transactions* Vol. 1 (2017), 3–54.

[33] Anne Chao and Lou Jost. 2012. Coverage-based rarefaction and extrapolation: Standardizing samples by completeness rather than size. *Ecology* 93, 12 (2012), 2533–2547.

[34] Daming D. Chen, Maverick Woo, David Brumley, and Manuel Egele. 2016. Towards automated dynamic analysis for linux-based embedded firmware. In *The Network and Distributed System Security Symposium (NDSS'16)*, Vol. 1. 1–16.

[35] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. 2020. MUZZ: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In *29th USENIX Security Symposium (USENIX Security'20)*. 2325–2342.

[36] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, Xiao Feng Wang, Wing Cheong Lau, et al. 2018. IoTFUZZER: Discovering memory corruptions in IoT through app-based fuzzing. In *The Network and Distributed System Security Symposium (NDSS'18)*. 1–15.

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

[38] Peng Chen, Jianzhong Liu, and Hao Chen. 2019. Matryoshka: Fuzzing deeply nested branches. In *The ACM Conference on Computer and Communications Security (CCS'19)*. 499–513.

[39] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. 2019. EnFuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. In *28th USENIX Security Symposium (USENIX Security'19)*. 1967–1983.

[40] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, et al. 2020. SAVIOR: Towards bug-driven hybrid testing. In *IEEE Symposium on Security and Privacy (S&P'20)*. IEEE Computer Society, Los Alamitos, CA, 1580–1596.

[41] Yuqi Chen, Christopher M. Poskitt, Jun Sun, Sridhar Adepu, and Fan Zhang. 2019. Learning-guided network fuzzing for testing cyber-physical system defences. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE'19)*. IEEE, 962–973.

[42] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed differential testing of JVM implementations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'16)*. 85–99.

[43] Yongheng Chen, Rui Zhong, Hong Hu, Hangfan Zhang, Yupeng Yang, Dinghao Wu, and Wenke Lee. 2021. One engine to Fuzz'em all: Generic language processor testing with semantic validation. In *IEEE Symposium on Security and Privacy (S&P'21)*. 1–17.

[44] Wen Cheng, Wang Haijun, Li Yuekang, Qin Shengchao, Liu Yang, Xu Zhiwu, Chen Hongxu, et al. 2020. MemLock: Memory usage guided fuzzing. In *IEEE/ACM 42nd International Conference on Software Engineering (ICSE'20)*. 765–777.

[45] Siddhartha Chib and Edward Greenberg. 1995. Understanding the metropolis-hastings algorithm. *American Statistician* 49, 4 (1995), 327–335.

[46] Mingi Cho, Seoyoung Kim, and Taekyoung Kwon. 2019. Intriguer: Field-level constraint solving for hybrid fuzzing. In *The ACM Conference on Computer and Communications Security (CCS'19)*. 515–530.

[47] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. 2019. Grey-box concolic testing on binary code. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE'19)*. IEEE, 736–747.
[48] Maria Christakis, Peter Müller, and Valentin Wüstholz. 2016. Guiding dynamic symbolic execution toward unverified program executions. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*. 144–155.
[49] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. 2013. The mathsat5 SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'13)*. Springer, 93–107.
[50] Koen Claessen and John Hughes. 2000. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*. 268–279.
[51] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA'07)*. 196–206.
[52] Robert K. Colwell, Anne Chao, Nicholas J. Gotelli, Shang-Yi Lin, Chang Xuan Mao, Robin L. Chazdon, and John T. Longino. 2012. Models and estimators linking individual-based and sample-based rarefaction, extrapolation and comparison of assemblages. *Journal of Plant Ecology* 5, 1 (2012), 3–21.
[53] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. 2017. Difuze: Interface aware fuzzing for kernel drivers. In *The ACM Conference on Computer and Communications Security (CCS'17)*. ACM, 2123–2138.
[54] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*. Springer, 337–340.
[55] Joeri De Ruiter and Erik Poll. 2015. Protocol state fuzzing of TLS implementations. In *24th USENIX Security Symposium (USENIX Security'15)*. 193–206.
[56] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2014. Language fuzzing using constraint logic programming. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE'14)*. 725–730.
[57] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2015. Fuzzing the Rust typechecker using CLP. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE'15)*. IEEE, 482–493.
[58] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (S&P'20)*. IEEE, 1497–1511.
[59] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, et al. 2021. Favocado: Fuzzing the binding code of Javascript engines using semantically correct test cases. In *The Network and Distributed System Security Symposium (NDSS'21)*. 1–15.
[60] Marco Dorigo, Mauro Birattari, and Thomas Stutzle. 2006. Ant colony optimization. *IEEE Computational Intelligence Magazine* 1, 4 (2006), 28–39.
[61] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. 2012. Enemy of the state: A state-aware black-box web vulnerability scanner. In *21st USENIX Security Symposium (USENIX Security'12)*. 523–538.
[62] Shawn Embleton, Sherri Sparks, and Ryan Cunningham. 2006. Sidewinder: An evolutionary guidance system for malicious input crafting. Black Hat USA.
[63] Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu, Minhui Xue, Sheng Wen, Dongxi Liu, Surya Nepal, and Yang Xiang. 2021. Snipuzz: Black-box fuzzing of IoT firmware via message snippet inference. In*The ACM Conference on Computer and Communications Security (CCS'21)*, 337–350.
[64] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. 2020. Analysis of DTLS implementations using protocol state fuzzing. In *29th USENIX Security Symposium (USENIX Security'20)*. 2523–2540.
[65] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. GREYONE: Data flow sensitive fuzzing. In *29th USENIX Security Symposium (USENIX Security'20)*. 2577–2594.
[66] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path sensitive fuzzing. In *IEEE Symposium on Security and Privacy (S&P'18)*. IEEE, 679–696.
[67] Vijay Ganesh, Tim Leek, and Martin Rinard. 2009. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 474–484.
[68] Xiang Gao, Ripon K. Saha, Mukul R. Prasad, and Abhik Roychoudhury. 2020. Fuzz testing based data augmentation to improve robustness of deep neural networks. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE'20)*. IEEE, 1147–1158.
[69] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. 2015. Pulsar: Stateful black-box fuzzing of proprietary network protocols. In *International Conference on Security and Privacy in Communication Systems (SecureComm'15)*. Springer, 330–347.
[70] Patrice Godefroid, Bo-Yuan Huang, and Marina Polishchuk. 2020. Intelligent REST API data fuzzing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'20)*. 725–736.

[71] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*. ACM, 206–215.
[72] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*. 213–223.
[73] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated whitebox fuzz testing. In *The Network and Distributed System Security Symposium (NDSS'08)*, Vol. 8. 151–166.
[74] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17)*. IEEE Press, 50–59.
[75] Gustavo Grieco, Martín Ceresa, and Pablo Buiras. 2016. QuickFuzz: An automatic random fuzzer for common file formats. In *Proceedings of the 9th International Symposium on Haskell (Haskell'16)*. 13–20.
[76] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. 2013. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *USENIX Security Symposium*. 49–64.
[77] HyungSeok Han and Sang Kil Cha. 2017. IMF: Inferred model-based fuzzer. In *The ACM Conference on Computer and Communications Security (CCS'17)*. ACM, 2345–2358.
[78] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. 2019. CodeAlchemist: Semantics-aware code generation to find vulnerabilities in Javascript engines. In *The Network and Distributed System Security Symposium (NDSS'19)*. 1–15.
[79] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. 2019. Learning to fuzz from symbolic execution with application to smart contracts. In *The ACM Conference on Computer and Communications Security (CCS'19)*. 531–548.
[80] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security'12)*. 445–458.
[81] H. Huang, P. Yao, R. Wu, Q. Shi, and C. Zhang. 2020. Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction. In *IEEE Symposium on Security and Privacy (S&P'20)*. IEEE Computer Society, Los Alamitos, CA, 1144–1158.
[82] Kim Hyungsub, Ozmen Muslum Ozgur, Bianchi Antonio, Celik Z. Berkay, and Xu Dongyan. 2021. PGFUZZ: Policy-guided fuzzing for robotic vehicles. In *The Network and Distributed System Security Symposium (NDSS'21)*. 1–18.
[83] Kyriakos K. Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. FuzzGen: Automatic fuzzer generation. In *29th USENIX Security Symposium (USENIX Security'20)*. 2271–2287.
[84] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. 2019. Razzer: Finding kernel race bugs through fuzzing. In *IEEE Symposium on Security and Privacy (S&P'19)*. IEEE, 754–768.
[85] jfoote. 2020. The exploitable GDB plugin. Retrieved February 7, 2020, from https://github.com/jfoote/exploitable.
[86] Bo Jiang, Ye Liu, and W. K. Chan. 2018. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE'18)*. IEEE, 259–269.
[87] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. 2020. Fuzzing error handling code using context-sensitive software fault injection. In *29th USENIX Security Symposium (USENIX Security'20)*. USENIX Association, 2595–2612.
[88] Wang Jinghan, Song Chengyu, and Heng Yin. 2021. Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing. In *The Network and Distributed System Security Symposium (NDSS'21)*. 1–17.
[89] William Johansson, Martin Svensson, Ulf E. Larson, Magnus Almgren, and Vincenzo Gulisano. 2014. T-Fuzz: Model-based fuzzing for robustness testing of telecommunication protocols. In *2014 IEEE 7th International Conference on Software Testing, Verification and Validation (ICST'14)*. IEEE, 323–332.
[90] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. 2009. A randomized dynamic program analysis technique for detecting real deadlocks. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*. 110–120.
[91] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. 2021. WINNIE: Fuzzing windows applications with harness synthesis and fast cloning. In *The Network and Distributed System Security Symposium (NDSS'21)*. 1–17.
[92] Ulf Kargén and Nahid Shahmehri. 2015. Turning programs against each other: High coverage fuzz-testing using binary-code mutation and dynamic slicing. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'15)*. ACM, 782–792.
[93] John G. Kemeny and J. Laurie Snell. 1976. *Markov Chains*, Vol. 6. Springer-Verlag, New York.
[94] James Kennedy and Russell Eberhart. 1995. Particle swarm optimization. In *Proceedings of International Conference on Neural Networks (ICNN'95*), Vol. 4. IEEE, 1942–1948.
[95] Kyungtae Kim, Dae R. Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid fuzzing on the Linux kernel. In *The Network and Distributed System Security Symposium (NDSS'20)*. 1–17.
[96] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*. ACM, 2123–2138.

[97] Zhifeng Lai, Shing-Chi Cheung, and Wing Kwong Chan. 2010. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)*. 235–244.

[98] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. 2021. Constraint-guided directed greybox fuzzing. In *30th USENIX Security Symposium (USENIX Security'21)*. 1–18.

[99] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. 2020. Montage: A neural network language model-guided Javascript engine fuzzer. In *29th USENIX Security Symposium (USENIX Security'20)*. USENIX Association, 1–18.

[100] Seungsoo Lee, Changhoon Yoon, Chanhee Lee, Seungwon Shin, Vinod Yegneswaran, and Phillip A. Porras. 2017. DELTA: A security assessment framework for software-defined networks. In *The Network and Distributed System Security Symposium (NDSS'17)*. 1–15.

[101] Caroline Lemieux and Koushik Sen. 2018. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18)*. 475–485.

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

[103] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: Program-state based binary fuzzing. In *Proceedings of the 2017 12th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17)*. ACM, 627–637.

[104] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, et al. 2021. UniFuzz: A holistic and pragmatic metrics-driven platform for evaluating fuzzers. In *30th USENIX Security Symposium (USENIX Security'21)*. 1–18.

[105] Yuekang Li, Yinxing Xue, Hongxu Chen, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang Liu. 2019. Cerebro: Context-aware adaptive fuzzing for effective vulnerability detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'19)*. 533–544.

[106] Chieh-Jan Mike Liang, Nicholas D. Lane, Niels Brouwers, Li Zhang, Börje F. Karlsson, Hao Liu, Yan Liu, Jun Tang, Xiang Shan, Ranveer Chandra, et al. 2014. Caiipa: Automated large-scale mobile app testing through contextual fuzzing. In *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking (MobiCom'14)*. 519–530.

[107] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. 2018. Fuzzing: State of the art. *IEEE Transactions on Reliability* 67, 3 (2018), 1199–1218.

[108] Daniel Liew, Cristian Cadar, Alastair F. Donaldson, and J. Ryan Stinnett. 2019. Just fuzz it: Solving floating-point constraints using coverage-guided fuzzing. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'19)*. 521–532.

[109] Guanjun Lin, Sheng Wen, Qing-Long Han, Jun Zhang, and Yang Xiang. 2020. Software vulnerability detection using deep neural networks: A survey. *Proceedings of the IEEE* 108, 10 (2020), 1825–1848.

[110] Baozheng Liu, Chao Zhang, Guang Gong, Yishun Zeng, Haifeng Ruan, and Jianwei Zhuge. 2020. FANS: Fuzzing Android native system services via automated interface analysis. In *29th USENIX Security Symposium (USENIX Security'20)*. USENIX Association, 307–323.

[111] Peng Liu, Xiangyu Zhang, Marco Pistoia, Yunhui Zheng, Manoel Marques, and Lingfei Zeng. 2017. Automatic text input generation for mobile testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE'17)*. IEEE, 643–653.

[112] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. 2019. Deepfuzz: Automatic generation of syntax valid C programs for fuzz testing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 1044–1051.

[113] LLVM. 2021. The LLVM Compiler Infrastructure. Retrieved March 2021 from https://llvm.org/.

[114] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)*. 329–339.

[115] Weisi Luo, Dong Chai, Xiaoyue Run, Jiang Wang, Chunrong Fang, and Zhenyu Chen. 2021. Graph-based fuzz testing for deep learning inference engines. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE'21)*. IEEE, 288–299.

[116] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, and Yu Song. 2019. MOPT: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security'19)*. USENIX Association, 1949–1966.

[117] Lei Ma, Cyrille Artho, Cheng Zhang, Hiroyuki Sato, Johannes Gmeiner, and Rudolf Ramler. 2015. GRT: Program-analysis-guided random testing (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE'15)*. IEEE, 212–223.

[118] Valentin J. M. Manès, Soomin Kim, and Sang Kil Cha. 2020. Ankou: Guiding grey-box fuzzing towards combinatorial difference. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE'20)*. 1024–1036.

[119] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* 47 (2019), 2312–2331.

[120] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. 2020. Detecting critical bugs in SMT solvers using blackbox mutational fuzzing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'20)*. 701–712.

[121] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA'16)*. 94–105.

[122] Paul Dan Marinescu and Cristian Cadar. 2013. KATCH: High-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'13)*. 235–245.

[123] Björn Mathis, Vitalii Avdiienko, Ezekiel O. Soremekun, Marcel Böhme, and Andreas Zeller. 2017. Detecting information flow by mutating input data. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17)*. IEEE, 263–273.

[124] Xiaozhu Meng and Barton P. Miller. 2016. Binary code is not easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA'16)*. ACM, 24–35.

[125] Nicholas Metropolis and Stanislaw Ulam. 1949. The Monte Carlo method. *Journal of the American Statistical Association* 44, 247 (1949), 335–341.

[126] Jonathan Metzman, Abhishek Arya, and Laszlo Szekeres. 2020. FuzzBench: Fuzzer Benchmarking as a Service. https://security.googleblog.com/2020/03/fuzzbench-fuzzer-benchmarking-as-service.html.

[127] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. In *11th Annual Conference of the International Speech Communication Association (INTERSPEECH'10)*. 1045–1048.

[128] Barton Miller and Jeff Hollingsworth. 2019. Dyninst: An API for program binary analysis and instrumentation. Retrieved June 2019 from https://dyninst.org/dyninst.

[129] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Communications of the ACM* 33, 12 (1990), 32–44.

[130] Charlie Miller. 2008. Fuzz by number - More data about fuzzing than you ever wanted to know. In *CanSecWest*.

[131] MITRE. 2020. 2020 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html.

[132] David Molnar, Xue Cong Li, and David A. Wagner. 2009. Dynamic test generation to find integer bugs in x86 binary Linux programs. In *18th USENIX Security Symposium (USENIX Security'09)*, Vol. 9. 67–82.

[133] Stefan Nagy and Matthew Hicks. 2019. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *IEEE Symposium on Security and Privacy (S&P'19)*. 787–802.

[134] Stefan Nagy, Anh Nguyen-Tuong, Jason D. Hiser, Jack W. Davidson, and Matthew Hicks. 2021. Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing. In *30th USENIX Security Symposium (USENIX Security'21)*. 1–19.

[135] James Newsome and Dawn Xiaodong Song. 2005. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *The Network and Distributed System Security Symposium (NDSS'05)*, Vol. 5. Citeseer, 3–4.

[136] Hoang Lam Nguyen, Nebras Nassar, Timo Kehrer, and Lars Grunske. 2020. MoFuzz: A fuzzer suite for testing model-driven software engineering tools. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE'20)*. IEEE, 1103–1115.

[137] Tai D. Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. sFuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE'20)*. 778–788.

[138] Shirin Nilizadeh, Yannic Noller, and Corina S. Pasareanu. 2019. DiffFuzz: Differential fuzzing for side-channel analysis. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE'19)*. IEEE, 176–187.

[139] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. 2020. SpecFuzz: Bringing spectre-type vulnerabilities to the surface. In *29th USENIX Security Symposium (USENIX Security'20)*. USENIX Association, 1481–1498.

[140] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. ParmeSan: Sanitizer-guided greybox fuzzing. In *29th USENIX Security Symposium (USENIX Security'20)*. USENIX Association, 2289–2306.

[141] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 75–84.

[142] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. Moonshine: Optimizing OS fuzzer seed selection with trace distillation. In *27th USENIX Security Symposium (USENIX Security'18)*. 729–743.
[143] Jianfeng Pan, Guanglu Yan, and Xiaocao Fan. 2017. Digtool: A virtualization-based framework for detecting kernel vulnerabilities. In *26th USENIX Security Symposium (USENIX Security'17)*. 149–165.
[144] Chang-Seo Park and Koushik Sen. 2008. Randomized active atomicity violation detection in concurrent programs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT'08/FSE-16)*. 135–145.
[145] S. Park, W. Xu, I. Yun, D. Jang, and T. Kim. 2020. Fuzzing JavaScript engines with aspect-preserving mutation. In *IEEE Symposium on Security and Privacy (S&P'20)*. IEEE Computer Society, Los Alamitos, CA, 1211–1225.
[146] Terence Parr. [n.d.]. ANTLR: ANother Tool for Language Recognition. Retrieved January 2021 from https://www.antlr.org/.
[147] Peachtech. 2021. Peach: The Peach Fuzzer Platform. Retrieved January 2021 from https://www.peach.tech/products/peach-fuzzer/.
[148] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*. 1–18.
[149] Hui Peng and Mathias Payer. 2020. USBFuzz: A framework for fuzzing USB drivers by device emulation. In *29th USENIX Security Symposium (USENIX Security'20)*. USENIX Association, 2559–2575.
[150] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: Fuzzing by program transformation. In *IEEE Symposium on Security and Privacy (S&P'18)*. IEEE, 697–710.
[151] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D. Keromytis, and Suman Jana. 2017. NEZHA: Efficient domain-independent differential testing. In *IEEE Symposium on Security and Privacy (S&P'17)*. IEEE, 615–632.
[152] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. 2017. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *The ACM Conference on Computer and Communications Security (CCS'17)*. ACM, 2155–2168.
[153] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2016. Model-based whitebox fuzzing for program binaries. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE'16)*. IEEE, 543–553.
[154] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNET: A greybox fuzzer for network protocols. In *IEEE International Conference on Software Testing, Verification and Validation (ICST'20)*. 1–6.
[155] Van-Thuan Pham, Wei Boon Ng, Konstantin Rubinov, and Abhik Roychoudhury. 2015. Hercules: Reproducing crashes in real-world application binaries. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE'15)*. IEEE, 891–901.
[156] Rasmus V. Rasmussen and Michael A. Trick. 2008. Round robin scheduling–a survey. *European Journal of Operational Research* 188, 3 (2008), 617–636.
[157] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware evolutionary fuzzing. In *The Network and Distributed System Security Symposium (NDSS'17)*. 1–15.
[158] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing seed selection for fuzzing. In *23rd USENIX Security Symposium (USENIX Security'14)*. 861–875.
[159] Nilo Redini, Andrea Continella, Dipanjan Das, Giulio De Pasquale, Noah Spahn, Aravind Machiry, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. 2021. DIANE: Identifying fuzzing triggers in apps to generate under-constrained inputs for IoT devices. In *IEEE Symposium on Security and Privacy (S&P'21)*. 1–17.
[160] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. 2010. FLAX: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *The Network and Distributed System Security Symposium (NDSS'10)*. 1–17.
[161] Fred B. Schneider. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)* 22, 4 (1990), 299–319.
[162] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. 2020. HYPER-CUBE: High-dimensional hypervisor fuzzing. In *The Network and Distributed System Security Symposium (NDSS'20)*. 1–16.
[163] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. 2021. NYX: Greybox hypervisor fuzzing using fast snapshots and affine types. In *30th USENIX Security Symposium (USENIX Security'21)*. 1–18.
[164] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-assisted feedback fuzzing for OS kernels. In *26th USENIX Security Symposium (USENIX Security'17)*. 167–182.
[165] Koushik Sen. 2007. Concolic testing. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*. 571–572.

[166] Koushik Sen. 2007. Effective random testing of concurrent programs. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*. 323–332.
[167] Koushik Sen. 2008. Race directed random testing of concurrent programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*. 11–21.
[168] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC'12)*. 309–318.
[169] Hossain Shahriar and Mohammad Zulkernine. 2012. Mitigating program security vulnerabilities: Approaches and challenges. *ACM Computing Surveys (CSUR)* 44, 3 (2012), 1–46.
[170] Claude E. Shannon. 1948. A mathematical theory of communication. *Bell System Technical Journal* 27, 3 (1948), 379–423.
[171] Dongdong She, Rahul Krishna, Lu Yan, Suman Jana, and Baishakhi Ray. 2020. MTFuzz: Fuzzing with a multi-task neural network. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'20)*. 737–749.
[172] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2019. NEUZZ: Efficient fuzzing with neural program smoothing. In *IEEE Symposium on Security and Privacy (S&P'19)*. 803–817.
[173] Juraj Somorovsky. 2016. Systematic fuzzing and testing of TLS libraries. In *The ACM Conference on Computer and Communications Security (CCS'16)*. 1492–1504.
[174] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. 2019. Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In *The Network and Distributed System Security Symposium (NDSS'19)*. 1–15.
[175] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2019. SoK: Sanitizing for security. In *IEEE Symposium on Security and Privacy (S&P'19)*. 1275–1295.
[176] Suhwan Song, Chengyu Song, Yeongjin Jang, and Byoungyoung Lee. 2020. CrFuzz: Fuzzing multi-purpose programs through input validation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'20)*. 690–700.
[177] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution. In *The Network and Distributed System Security Symposium (NDSS'16)*. 1–16.
[178] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. Sok: Eternal war in memory. In *IEEE Symposium on Security and Privacy (S&P'13)*. 48–62.
[179] Dimitrios Tychalas, Hadjer Benkraouda, and Michail Maniatakos. 2021. ICSFuzz: Manipulating I/Os and repurposing binary code to enable instrumented fuzzing in ICS control applications. In *30th USENIX Security Symposium (USENIX Security'21)*. 1–16.
[180] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. 2016. IFuzzer: An evolutionary interpreter fuzzer using genetic programming. In *European Symposium on Research in Computer Security (ESORICS'16)*. Springer, 581–601.
[181] Vasudev Vikram, Rohan Padhye, and Koushik Sen. 2021. Growing a test corpus with bonsai fuzzing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE'21)*. IEEE, 723–735.
[182] Martin Vuagnoux. 2005. Autodafé: An act of software torture. In *Proceedings of the 22th Chaos Communication Congress*. Chaos Computer Club, 47–58.
[183] Dmitry Vyukov. 2021. syzkaller. Retrieved May 2021 from https://github.com/google/syzkaller.
[184] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. 2020. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE'20)*. IEEE, 999–1010.
[185] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-driven seed generation for fuzzing. In *IEEE Symposium on Security and Privacy (S&P'17)*. IEEE, 579–594.
[186] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE'19)*. IEEE, 724–735.
[187] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *IEEE Symposium on Security and Privacy (S&P'10)*. IEEE, 497–512.
[188] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. 2020. Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization. In *The Network and Distributed System Security Symposium (NDSS'20)*. 1–17.
[189] Richard Wartell, Yan Zhou, Kevin W. Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. 2011. Differentiating code from data in x86 binaries. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 522–536.

[190] Jiayi Wei, Jia Chen, Yu Feng, Kostas Ferles, and Isil Dillig. 2018. Singularity: Pattern fuzzing for worst case complexity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'18)*. 213–223.

[191] Svante Wold, Kim Esbensen, and Paul Geladi. 1987. Principal component analysis. *Chemometrics and Intelligent Laboratory Systems* 2, 1–3 (1987), 37–52.

[192] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. 2013. Scheduling black-box mutational fuzzing. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS'13)*. 511–522.

[193] Valentin Wüstholz and Maria Christakis. 2020. Harvey: A greybox fuzzer for smart contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'20)*. 1398–1409.

[194] M. Xu, S. Kashyap, H. Zhao, and T. Kim. 2020. Krace: Data race fuzzing for kernel file systems. In *IEEE Symposium on Security and Privacy (S&P'20)*. IEEE Computer Society, Los Alamitos, CA, 1396–1413.

[195] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Designing new operating primitives to improve fuzzing performance. In *The ACM Conference on Computer and Communications Security (CCS'17)*. ACM, 2313–2328.

[196] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. 2019. Fuzzing file systems via two-dimensional input space exploration. In *IEEE Symposium on Security and Privacy (S&P'19)*. 818–834.

[197] Wen Xu, Soyeon Park, and Taesoo Kim. 2020. FREEDOM: Engineering a State-of-the-Art DOM Fuzzer. In *The ACM Conference on Computer and Communications Security (CCS'20)*. 971–986.

[198] Shengbo Yan, Chenlu Wu, Hang Li, Wei Shao, and Chunfu Jia. 2020. PathAFL: Path-coverage assisted fuzzing. In *Proceedings of the 15th ACM ASIA Conference on Computer and Communications Security (ASIACCS'20)*. 598–609.

[199] Dingning Yang, Yuqing Zhang, and Qixu Liu. 2012. Blendfuzz: A model-based framework for fuzz testing programs with grammatical inputs. In *IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom'12)*. IEEE, 1070–1076.

[200] Wei You, Xuwei Liu, Shiqing Ma, David Perry, Xiangyu Zhang, and Bin Liang. 2019. SLF: Fuzzing without valid seed inputs. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE'19)*. IEEE, 712–723.

[201] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. 2019. ProFuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In *IEEE Symposium on Security and Privacy (S&P'19)*. IEEE, 769–786.

[202] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. 2017. SemFuzz: Semantics-based automatic generation of proof-of-concept exploits. In *The ACM Conference on Computer and Communications Security (CCS'17)*. ACM, 2139–2154.

[203] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. 2020. EcoFuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In *29th USENIX Security Symposium (USENIX Security'20)*. USENIX Association, 2307–2324.

[204] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security'18)*. 745–761.

[205] Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. 2014. AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares. In *The Network and Distributed System Security Symposium (NDSS'14)*, Vol. 23. 1–16.

[206] Michał Zalewski. 2021. AFL (American fuzzy lop). Retrieved January 21, 2021, from https://github.com/google/AFL.

[207] Qian Zhang, Jiyuan Wang, Muhammad Ali Gulzar, Rohan Padhye, and Miryung Kim. 2020. BigFuzz: Efficient fuzz testing for data analytics using framework abstraction. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE'20)*. IEEE, 722–733.

[208] Yangyong Zhang, Lei Xu, Abner Mendoza, Guangliang Yang, Phakpoom Chinprutthiwong, and Guofei Gu. 2019. Life after speech recognition: Fuzzing semantic misinterpretation for voice assistant applications. In *The Network and Distributed System Security Symposium (NDSS'19)*. 1–15.

[209] Zhuo Zhang, Wei You, Guanhong Tao, Yousra Aafer, Xuwei Liu, and Xiangyu Zhang. 2021. STOCHFUZZ: Sound and cost-effective fuzzing of stripped binaries by incremental and stochastic rewriting. In *IEEE Symposium on Security and Privacy (S&P'21)*. 1–18.

[210] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. 2019. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In *The Network and Distributed System Security Symposium (NDSS'19)*. 1–15.

[211] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. 2019. FIRM-AFL: High-Throughput greybox fuzzing of IoT firmware via augmented process emulation. In *28th USENIX Security Symposium (USENIX Security'19)*. USENIX Association, 1099–1114.

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

[213] Chijin Zhou, Mingzhe Wang, Jie Liang, Zhe Liu, and Yu Jiang. 2020. Zeror: Speed up fuzzing with coverage-sensitive tracing and scheduling. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE'20)*. IEEE, 858–870.

[214] Xiaogang Zhu and Marcel Böhme. 2021. Regression greybox fuzzing. In *The ACM Conference on Computer and Communications Security (CCS'21)*. 2169–2182.

[215] Xiaogang Zhu, Xiaotao Feng, Tengyun Jiao, Sheng Wen, Yang Xiang, Seyit Camtepe, and Jingling Xue. 2019. A feature-oriented corpus for understanding, evaluating and improving fuzz testing. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security (Asia CCS'19)*. 658–663.

[216] Xiaogang Zhu, Xiaotao Feng, Xiaozhu Meng, Sheng Wen, Seyit Camtepe, Yang Xiang, and Kui Ren. 2020. CSI-Fuzz: Full-speed edge tracing using coverage sensitive instrumentation. *IEEE Transactions on Dependable and Secure Computing* 19 (2020), 912–923.

[217] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. 2020. FuzzGuard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In *29th USENIX Security Symposium (USENIX Security'20)*. 2255–2269.

Received July 2021; revised December 2021; accepted January 2022