The Art, Science, and Engineering of Fuzzing: A Survey

Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo

Abstract—Among the many software testing techniques available today, *fuzzing* has remained highly popular due to its conceptual simplicity, its low barrier to deployment, and its vast amount of empirical evidence in discovering real-world software vulnerabilities. At a high level, fuzzing refers to a process of repeatedly running a program with generated inputs that may be syntactically or semantically malformed. While researchers and practitioners alike have invested a large and diverse effort towards improving fuzzing in recent years, this surge of work has also made it difficult to gain a comprehensive and coherent view of fuzzing. To help preserve and bring coherence to the vast literature of fuzzing, this paper presents a unified, general-purpose model of fuzzing together with a taxonomy of the current fuzzing literature. We methodically explore the design decisions at every stage of our model fuzzer by surveying the related literature and innovations in the art, science, and engineering that make modern-day fuzzers effective.

Index Terms—software security, automated software testing, fuzzing, fuzz testing.

✦

1 INTRODUCTION

Ever since its introduction in the early 1990s [157], *fuzzing* has remained one of the most widely-deployed techniques to test software correctness and reliability. At a high level, fuzzing refers to a process of repeatedly running a program with generated inputs that may be syntactically or semantically malformed. In practice, malicious attackers routinely deploy fuzzing in scenarios such as exploit generation and penetration testing [24], [113]; several teams in the 2016 DARPA Cyber Grand Challenge (CGC) also employed fuzzing in their cyber reasoning systems [40], [209], [11], [96]. Fueled by these activities, defenders have started to use fuzzing in an attempt to discover vulnerabilities before attackers do. For example, prominent vendors such as Adobe [1], Cisco [2], Google [64], [7], [18], and Microsoft [41], [10] all employ fuzzing as part of their secure development practices. More recently, security auditors [247] and open-source developers [6] have also started to use fuzzing to gauge the security of commodity software packages and provide some suitable forms of assurance to end-users.

The fuzzing community is extremely vibrant. As of this writing, GitHub alone hosts over a thousand public repositories related to fuzzing [89]. And as we will demonstrate, the literature also contains a large number of fuzzers (see

Figure 1 on p. 5) and an increasing number of fuzzing studies appear at major security conferences (e.g., [235], [55], [40], [183], [86], [249]). In addition, the blogosphere is filled with many success stories of fuzzing, some of which also contain what we consider to be gems that warrant a permanent place in the literature.

Unfortunately, this surge of work in fuzzing by researchers and practitioners alike also bears a warning sign of impeded progress. For example, the description of some fuzzers do not go much beyond their source code and manual page. As such, it is easy to lose track of the design decisions and potentially important tweaks in these fuzzers over time. Furthermore, there has been an observable fragmentation in the terminology used by various fuzzers. For example, whereas AFL [241] uses the term "test case minimization" to refer to a technique that reduces the size of a crashing input, the same technique is called "test case reduction" in funfuzz [194]. At the same time, while BFF [52] includes a similar-sounding technique called "crash minimization", which is a technique that seeks to minimize the number of bits that differ between a crashing input and its original seed file and is not related to reducing input size. We believe such fragmentation makes it difficult to discover and disseminate fuzzing knowledge and this may severely hinder the progress in fuzzing research in the long run.

Due to the above reasons, we believe it is prime time to consolidate and distill the large amount of progress in fuzzing, many of which happened after the three trade-books on the subject were published in 2007–2008 [82], [212], [214].

As we attempt to unify the field, we will start by using §2 to present our fuzzing terminology and a unified model of fuzzing. Staying true to the purpose of this paper, our terminology is chosen to closely reflect the current predominant usages, and our model fuzzer (Algorithm 1, p. 3) is designed to suit a large number of fuzzing tasks as classified in a taxonomy of the current fuzzing literature (Figure 1, p. 5). With this setup, we will then explore every

- *V. J. M. Manès is with KAIST Cyber Security Research Center, Korea.*
  E-mail: valentin.manes@kaist.ac.kr.
- *H. Han and S. K. Cha are with KAIST, Korea.*
  E-mail: hyungseok.han@kaist.ac.kr and sangkilc@kaist.ac.kr.
- *C. Han is with Naver Corp., Korea.*
  E-mail: cwhan.tunz@navercorp.com.
- *M. Egele is with Boston University.*
  E-mail: megele@bu.edu.
- *E. J. Schwartz is with SEI, Carnegie Mellon University.*
  E-mail: edmcman@cmu.edu.
- *M. Woo is with Carnegie Mellon University.*
  E-mail: pooh@cmu.edu.

Corresponding author: Sang Kil Cha.
Manuscript submitted on April 8, 2019; revised on September 18, 2019.

stage of our model fuzzer in §3–§7 and present a detailed overview of major fuzzers in Table 1 (p. 6). At each stage, we will survey the relevant literature to explain the design choices, discuss important trade-offs, and highlight many marvelous engineering efforts that help make modern-day fuzzers effective at their task.

2 SYSTEMIZATION, TAXONOMY, AND TEST PROGRAMS

The term "fuzz" was originally coined by Miller et al. in 1990 to refer to a program that "generates a stream of random characters to be consumed by a target program" [157, p. 4]. Since then, the concept of fuzz as well as its action—"fuzzing"—has appeared in a wide variety of contexts, including dynamic symbolic execution [93], [236], grammar-based test case generation [91], [109], [223], permission testing [27], [83], behavioral testing [126], [182], [234], complexity testing [140], [232], kernel testing [226], [204], [193], representation dependence testing [125], function detection [237], robustness evaluation [233], exploit development [115], GUI testing [206], signature generation [75], penetration testing [84], [162], embedded devices [161], and neural network testing [176], [98]. To systematize the knowledge from the vast literature of fuzzing, let us first present a terminology of fuzzing extracted from modern uses.

2.1 Fuzzing & Fuzz Testing

Intuitively, fuzzing is the action of running a Program Under Test (PUT) with "fuzz inputs". Honoring Miller et al., we consider a fuzz input to be an input that the PUT *may not* be expecting, i.e., an input that the PUT may process incorrectly and trigger a behavior that was unintended by the developer of the PUT. To capture this idea, we define the term *fuzzing* as follows.

Definition 1 (*Fuzzing*). Fuzzing is the execution of the PUT using input(s) sampled from an input space (the "fuzz input space") that *protrudes* the expected input space of the PUT.

Three remarks are in order. First, although it may be common to see the fuzz input space to contain the expected input space, this is *not* necessary—it suffices for the former to contain an input *not in* the latter. Second, in practice fuzzing almost surely runs for *many* iterations; thus writing "repeated executions" above would still be largely accurate. Third, the sampling process is *not* necessarily randomized, as we will see in §5.

Fuzz testing is a form of software testing technique that utilizes fuzzing. Historically, fuzz testing has been used primarily for finding security-related bugs; however, nowadays it is also widely used in many non-security applications. In addition, we also define *fuzzer* and *fuzz campaign*, both of which are common terms in fuzz testing:

Definition 2 (*Fuzz Testing*). Fuzz testing is the use of fuzzing to test if a PUT violates a correctness policy.

Definition 3 (*Fuzzer*). A fuzzer is a program that performs fuzz testing on a PUT.

Definition 4 (*Fuzz Campaign*). A fuzz campaign is a specific execution of a fuzzer on a PUT with a specific correctness policy.

The goal of running a PUT through a fuzz campaign is to find bugs [29] that violate the specified correctness policy. For example, a correctness policy employed by early fuzzers tested only whether a generated input—the *test case*—crashed the PUT. However, fuzz testing can actually be used to test any policy observable from an execution, i.e., EM-enforceable [190]. The specific mechanism that decides whether an execution violates the policy is called the *bug oracle*.

Definition 5 (*Bug Oracle*). A bug oracle is a program, perhaps as part of a fuzzer, that determines whether a given execution of the PUT violates a specific correctness policy.

Although fuzz testing is focused on finding policy violations, the techniques it is based on can be diverted towards other usages. Indeed, fuzzing-inspired approaches have been employed in a broad range of applications. For instance, PerfFuzz [140] looks for inputs that reveal performance problems.

We refer to the algorithm implemented by a fuzzer simply as its "fuzz algorithm". Almost all fuzz algorithms depend on some parameters beyond (the path to) the PUT. Each concrete setting of the parameters is a *fuzz configuration*:

Definition 6 (*Fuzz Configuration*). A fuzz configuration of a fuzz algorithm comprises the parameter value(s) that control(s) the fuzz algorithm.

Our definition of a fuzz configuration is intended to be broad. Note that the type of values in a fuzz configuration depend on the type of the fuzz algorithm. For example, a fuzz algorithm that sends streams of random bytes to the PUT [157] has a simple configuration space $\{(PUT)\}$. On the other hand, sophisticated fuzzers contain algorithms that accept a set of configurations and evolve the set over time—this includes adding and removing configurations. For example, CERT BFF [52] varies both the mutation ratio and the seed over the course of a campaign, and thus its configuration space is of the form $\{(PUT, s_1, r_1), (PUT, s_2, r_2), \ldots\}$. A seed is a (commonly well-structured) input to the PUT, used to generate test cases by modifying it. Fuzzers typically maintain a collection of seeds known as the *seed pool* and some fuzzers evolve the pool as the fuzz campaign progresses. Finally, a fuzzer is able to store some data within each configuration. For instance, coverage-guided fuzzers may store the attained coverage in each configuration.

2.2 Paper Selection Criteria

To achieve a well-defined scope, we have chosen to include all publications on fuzzing in the most-recent proceedings of 4 major security conferences and 3 major software engineering conferences from January 2008 to February 2019. Alphabetically, the former includes (i) ACM Conference on Computer and Communications Security (CCS), (ii) IEEE Symposium on Security and Privacy (S&P), (iii) Network and Distributed System Security Symposium (NDSS), and (iv) USENIX Security Symposium (USEC); and the latter includes (i) ACM International Symposium on the Foundations

---

ALGORITHM 1: Fuzz Testing

Input: $\mathbb{C}$, t_{limit}
Output: $\mathbb{B}$ // a finite set of bugs

1   $\mathbb{B} \leftarrow \varnothing$
2   $\mathbb{C} \leftarrow$ **PREPROCESS**($\mathbb{C}$)
3   **while** $t_{\text{elapsed}} < t_{\text{limit}} \wedge$ **CONTINUE**($\mathbb{C}$) **do**
4   conf $\leftarrow$ **SCHEDULE**($\mathbb{C}$, t_{elapsed}, t_{limit})
5   tcs $\leftarrow$ **INPUTGEN**(*conf*)
      // O_{bug} is embedded in a fuzzer
6      $\mathbb{B}'$, execinfos $\leftarrow$ **INPUTEVAL**(*conf*, tcs, O_{bug})
7      $\mathbb{C} \leftarrow$ **CONFUPDATE**($\mathbb{C}$, *conf*, execinfos)
8      $\mathbb{B} \leftarrow \mathbb{B} \cup \mathbb{B}'$

9   **return** $\mathbb{B}$

---

of Software Engineering (FSE), (ii) IEEE/ACM International Conference on Automated Software Engineering (ASE), and (iii) International Conference on Software Engineering (ICSE). For writings that appear in other venues or mediums, we include them based on our own judgment on their relevance.

As mentioned in §2.1, *fuzz testing* has been mainly focused on finding security-related bugs. In theory, focusing on security bugs does not imply a difference in the testing process beyond the selection of a bug oracle. The techniques used often vary in practice, however. When designing a testing tool, access to source code and some knowledge about the PUT are often assumed. Such assumptions often drive the development of testing tools to have different characteristics compared to those of fuzzers, which are more likely to be employed by parties other than the developer of the PUT. Nevertheless, the two fields are still closely related to one another. Therefore, when we are unsure whether to classify a publication as relating to "fuzz testing" and include it in this survey, we follow a simple rule of thumb: we include the publication if the word *fuzz* appears in it.

2.3 Fuzz Testing Algorithm

In Algorithm 1, we present a generic algorithm for fuzz testing, which we imagine to have been implemented in a *model fuzzer*. It is general enough to accommodate existing fuzzing techniques, including black-, grey-, and white-box fuzzing as defined in §2.4. Algorithm 1 takes a set of fuzz configurations $\mathbb{C}$ and a timeout t_{limit} as input, and outputs a set of discovered bugs $\mathbb{B}$. It consists of two parts. The first part is the PREPROCESS function, which is executed at the beginning of a fuzz campaign. The second part is a series of five functions inside a loop: SCHEDULE, INPUTGEN, INPUTEVAL, CONFUPDATE, and CONTINUE. Each execution of this loop is called a *fuzz iteration* and each time INPUTEVAL executes the PUT on a test case is called a *fuzz run*. Note that some fuzzers do *not* implement all five functions. For example, to model Radamsa [106], which never updates the set of fuzz configurations, CONFUPDATE always returns the current set of configurations unchanged.

PREPROCESS ($\mathbb{C}$) $\rightarrow$ $\mathbb{C}$
     A user supplies PREPROCESS with a set of fuzz configurations as input, and it returns a potentially-modified set of fuzz configurations. Depending on the fuzz algorithm, PREPROCESS may perform a variety of actions such as inserting instrumentation code to PUTs, or measuring the execution speed of seed files. See §3.

SCHEDULE ($\mathbb{C}$, t_{elapsed}, t_{limit}) $\rightarrow$ conf
     SCHEDULE takes in the current set of fuzz configurations, the current time t_{elapsed}, and a timeout t_{limit} as input, and selects a fuzz configuration to be used for the current fuzz iteration. See §4.

INPUTGEN (conf) $\rightarrow$ tcs
     INPUTGEN takes a fuzz configuration as input and returns a set of concrete test cases tcs as output. When generating test cases, INPUTGEN uses specific parameter(s) in conf. Some fuzzers use a seed in conf for generating test cases, while others use a model or grammar as a parameter. See §5.

INPUTEVAL (conf, tcs, O_{bug}) $\rightarrow$ $\mathbb{B}'$, execinfos
     INPUTEVAL takes a fuzz configuration conf, a set of test cases tcs, and a bug oracle O_{bug} as input. It executes the PUT on tcs and checks if the executions violate the correctness policy using the bug oracle O_{bug}. It then outputs the set of bugs found $\mathbb{B}'$ and information about each of the fuzz runs execinfos, which may be used to update the fuzz configurations. We assume O_{bug} is embedded in our model fuzzer. See §6.

CONFUPDATE ($\mathbb{C}$, conf, execinfos) $\rightarrow$ $\mathbb{C}$
     CONFUPDATE takes a set of fuzz configurations $\mathbb{C}$, the current configuration conf, and the information about each of the fuzz runs execinfos, as input. It may update the set of fuzz configurations $\mathbb{C}$. For example, many grey-box fuzzers reduce the number of fuzz configurations in $\mathbb{C}$ based on execinfos. See §7.

CONTINUE ($\mathbb{C}$) $\rightarrow$ {True, False}
     CONTINUE takes a set of fuzz configurations $\mathbb{C}$ as input and outputs a boolean indicating whether a new fuzz iteration should occur. This function is useful to model white-box fuzzers that can terminate when there are no more paths to discover.

2.4 Taxonomy of Fuzzers

For this paper, we have categorized fuzzers into three groups based on the granularity of semantics a fuzzer observes in each fuzz run. These three groups are called black-, grey-, and white-box fuzzers, which we define below. Note that this classification is different from traditional software testing, where there are only black- and white-box testing [164]. As we will discuss in §2.4.3, grey-box fuzzing is a variant of white-box fuzzing that can obtain only partial information from each fuzz run.

2.4.1 *Black-box Fuzzer*

The term "black-box" is commonly used in software testing [164], [35] and fuzzing to denote techniques that do *not* see the internals of the PUT—these techniques can observe only the input/output behavior of the PUT, treating it as a black-box. In software testing, black-box testing is also called IO-driven or data-driven testing [164]. Most traditional fuzzers [16], [107], [52], [8], [53] are in this category. Some modern fuzzers, e.g., funfuzz [194] and Peach [79], also take the structural information about inputs into account to generate more meaningful test cases while maintaining the characteristic of not inspecting the PUT. A similar intuition is used in adaptive random testing [60].

2.4.2 *White-box Fuzzer*

At the other extreme of the spectrum, white-box fuzzing [93] generates test cases by analyzing the internals of the PUT and the information gathered when executing the PUT. Thus, white-box fuzzers are able to explore the state space of the PUT systematically. The term *white-box fuzzing* was introduced by Godefroid [90] in 2007 and refers to dynamic symbolic execution (DSE), which is a variant of symbolic execution [42], [130], [112]. In DSE, symbolic and concrete execution operate concurrently, where concrete program states are used to simplify symbolic constraints, e.g., concretizing system calls. DSE is thus often referred to as *concolic testing* (a portmanteau of "concrete" and "symbolic") [198], [92]. In addition, white-box fuzzing has also been used to describe fuzzers that employ taint analysis [87]. The overhead of white-box fuzzing is typically much higher than that of black-box fuzzing. This is partly because DSE implementations [93], [49], [28] often employ dynamic instrumentation and SMT solving [160]. While DSE is an active research area [93], [91], [41], [179], [116], works in DSE generally do not claim to be about white-box fuzzing and so we did not include many such works. This paper does not provide a comprehensive survey on DSE and we refer the reader to recent survey papers [20], [192] for more information on this topic.

2.4.3 *Grey-box Fuzzer*

Some fuzzers [81], [71], [214] take a middle ground approach dubbed *grey-box fuzzing*. In general, grey-box fuzzers can obtain *some* information internal to the PUT and/or its executions. Unlike white-box fuzzers, grey-box fuzzers do not reason about the full semantics of the PUT; instead, they may perform lightweight static analysis on the PUT and/or gather dynamic information about its executions, such as code coverage. Grey-box fuzzers rely on approximated, imperfect information in order to gain speed and be able to test more inputs. Although there usually is some consensus among security experts, the distinction among black-, grey- and white-box fuzzing is not always clear. Black-box fuzzers may collect some information about fuzz runs, and white-box fuzzers often use some approximations. When classifying the fuzzers in this survey, particularly in Table 1, we often had to rely on our own judgement.

An early example of grey-box fuzzer is EFS [71], which uses code coverage gathered from each fuzz run to generate test cases with an evolutionary algorithm. Randoop [172] also uses a similar approach, though it was not designed specifically for finding vulnerabilities. Modern fuzzers such as AFL [241] and VUzzer [183] are exemplars in this category.

2.5 Fuzzer Genealogy and Overview

Figure 1 (p. 5) presents our categorization of existing fuzzers in chronological order. Starting from the seminal work by Miller et al. [157], we have manually selected popular fuzzers that either appeared in a major conference or obtained more than 100 stars on GitHub and show their relationships as a graph. Black-box fuzzers are in the left half of the figure, and grey- and white-box fuzzers are in the right half. Furthermore, fuzzers are subdivided depending on the type of input the PUT uses: file, network, UI, web, kernel I/O, or threads (in the case of concurrency fuzzers).

Table 1 (p. 6) presents a detailed summary of the techniques used in the most notable fuzzers in Figure 1. We had to omit some of fuzzers in Figure 1 due to space constraints. Each fuzzer is summarized based on its implementation of the five functions of our model fuzzer, and a miscellaneous section that provides other details on the fuzzer. Here we will explain the properties described by each column of the table. Column 1 indicates whether a fuzzer is black- (●), white- (○), or grey-box (◐). Two circles appear when a fuzzer has two phases which use different kinds of feedback gathering. For example, SymFuzz [55] runs a white-box analysis as a preprocessing step in order to optimize the performance of a subsequent black-box campaign (●+○), whereas hybrid fuzzers such as Driller [209] alternate between white- and grey-box fuzzing (◐+○). Column 2 shows whether the source code of a fuzzer is publicly available. Column 3 denotes whether a fuzzer needs the source code of the PUT to operate. Column 4 indicates whether a fuzzer supports in-memory fuzzing (see §3.1.3). Column 5 is about whether a fuzzer can infer models (see §5.1.2). Column 6 shows whether a fuzzer performs either static or dynamic analysis in Preprocess. Column 7 indicates if a fuzzer supports handling multiple seeds, and perform scheduling. Column 8 specifies if a fuzzer performs input mutation to generate test cases. We use ◐ to indicate fuzzers that guide input mutation based on the execution feedback. Column 9 is about whether a fuzzer generates test cases based on a model. Column 10 shows whether a fuzzer performs a symbolic analysis to generate test cases. Column 11 identifies fuzzers that leverage taint analysis to guide their test case generation. Columns 12 and 13 show whether a fuzzer performs crash triage using either stack hashing or code coverage. Column 14 indicates if a fuzzer evolves the seed pool during ConfUpdate, such as adding new seeds to the pool (see §7.1). Column 15 is about whether a fuzzer learns an input model in an online fashion. Finally, column 16 shows which fuzzers remove seeds from the seed pool (see §7.2).

3 Preprocess

Some fuzzers have a first step to prepare the main loop of Algorithm 1 by modifying the initial set of fuzz configurations before the first fuzz iteration. Such preprocessing is commonly used to instrument the PUT, to weed out potentially-redundant configurations (i.e., "seed selection" [184]), to trim seeds, and to generate driver applications. As will be detailed in §5.1.1 (p. 9), Preprocess can also be used to prepare a model for future input generation (InputGen).

3.1 Instrumentation

Unlike black-box fuzzers, both grey- and white-box fuzzers can instrument the PUT to gather execution feedback as InputEval performs fuzz runs (see §6), or to fuzz the memory contents at runtime. The amount of collected information defines the color of a fuzzer (i.e., black-, white-, or grey-box). Although there are other ways of acquiring information about the internals of the PUT (e.g., processor traces or system call usage [213], [95]), instrumentation is often the method that collects the most valuable feedback.

Program instrumentation can be either static or dynamic—the former happens before the PUT runs (Preprocess),
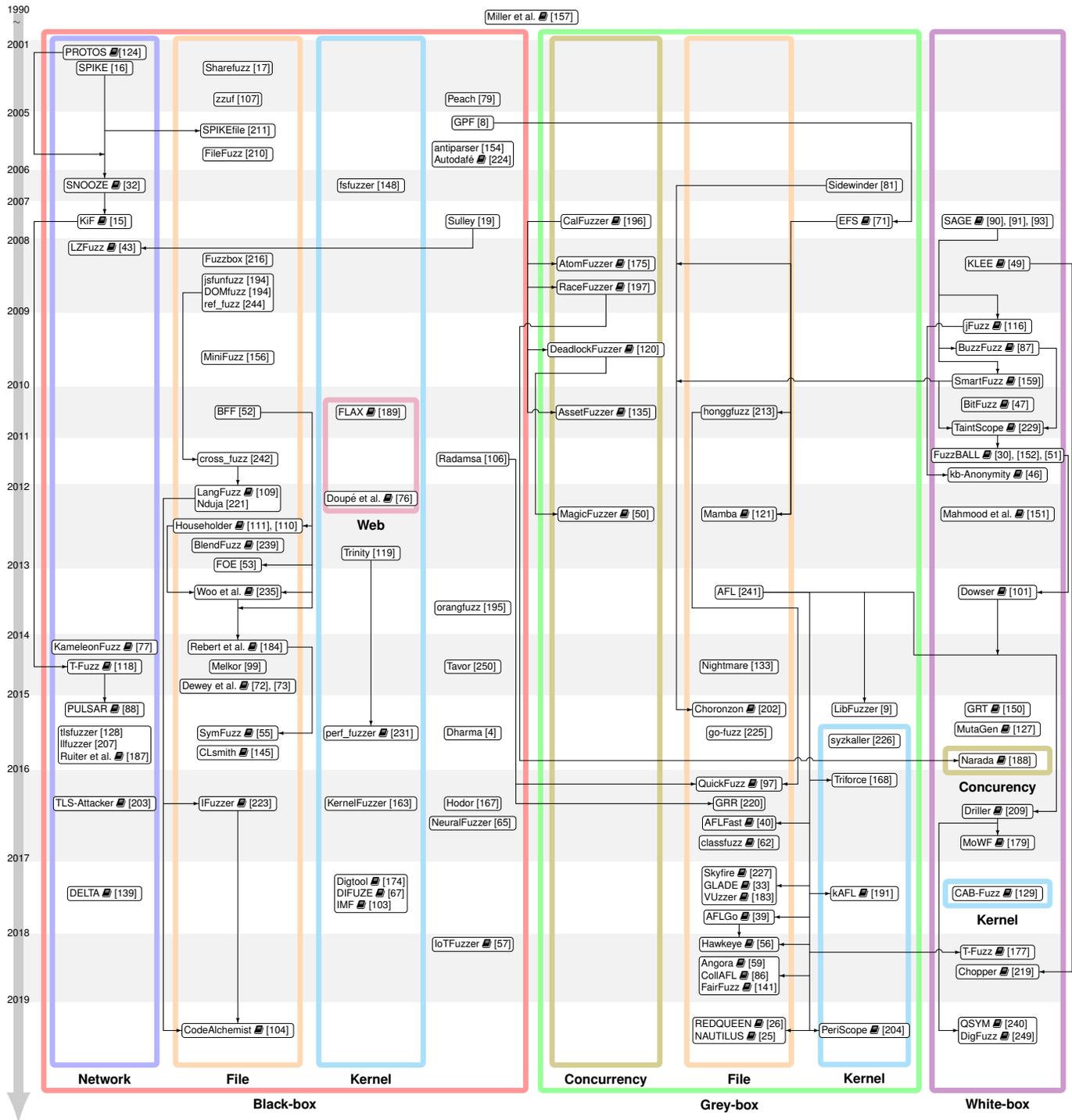
Fig. 1: Genealogy tracing significant fuzzers' lineage back to Miller *et al.*'s seminal work. Each node in the same row represents a set of fuzzers appeared in the same year. A solid arrow from X to Y indicates that Y cites, references, or otherwise uses techniques from X. ▤ denotes that a paper describing the work was published.

TABLE 1: Overview of fuzzers sorted by their instrumentation granularity and their name. ●, ◐, and ○ represent black-, grey-, and white-box, respectively.

Fuzzer	Misc.			PREPROCESS			SCHEDULE	INPUTGEN				INPUTEVAL		CONFUPDATE		
	1. Feedback Gathering Granularity	2. Open-Sourced	3. Source Code Required	4. Support In-memory Fuzzing	5. Model Construction	6. Program Analysis	7. Seed Scheduling	8. Mutation	9. Model-based	10. Constraint-based	11. Taint Analysis	12. Crash Triage: Stack Hash	13. Crash Triage: Coverage	14. Evolutionary Seed Pool Update	15. Model Update	16. Seed Pool Culling
BFF [52]	●	✓					✓	●				✓				
CodeAlchemist [104]	●	✓		◐					✓							
CLsmith [145]	●	✓						●	✓							
DELTA [139]	●							●	✓							
DIFUZE [67]	●	✓	✓	○				●	✓							
Digtool [174]	●							●								
Doupé et al. [76]	●								✓						●	
FOE [53]	●	✓					✓	●				✓				
GLADE [33]	●	✓			●		✓		✓						●	
IMF [103]	●	✓			●			●	✓							
jsfunfuzz [194]	●	✓							✓			✓				
LangFuzz [109]	●							●	✓							
Miller et al. [157]	●	✓														
Peach [79]	●	✓							✓			✓				
PULSAR [88]	●	✓			●				✓						●	
Radamsa [106]	●	✓						●	✓							
Ruiter et al. [187]	●								✓						●	
TLS-Attacker [203]	●	✓						●								
zuff [107]	●	✓						●								
FLAX [189]	●+○		✓			✓		●			✓					
IoTFuzzer [57]	●+○				●	✓		●	✓							
SymFuzz [55]	●+○	✓				✓		●				✓				
AFL [241]	◐	✓		✓			✓	●						✓	✓	✓
AFLFast [40]	◐	✓		✓			✓[†]	●						✓	✓	✓
AFLGo [39]	◐	✓	✓	✓		✓	✓[†]	●						✓	✓	✓
AssetFuzzer [135]	◐	✓					✓									
AtomFuzzer [175]	◐	✓	✓				✓									
CalFuzzer [196]	◐	✓	✓				✓									
classfuzz [62]	◐						✓	●								
CollAFL [86]	◐[†]		✓	✓			✓[†]	●						✓	✓	✓
DeadlockFuzzer [120]	◐	✓	✓				✓									
FairFuzz [141]	◐	✓		✓			✓[†]	◐[†]						✓	✓	✓
go-fuzz [225]	◐	✓	✓				✓	●	✓				✓	✓	◐	✓
Hawkeye [56]	◐		✓	✓		✓	✓	◐					✓	✓		
honggfuzz [213]	◐	✓						●				✓		✓		
kAFL [191]	◐	✓						●						✓		
LibFuzzer [9]	◐	✓	✓	✓			✓	●						✓	✓	
MagicFuzzer [50]	◐	✓	✓			✓										
Nautilus [25]	◐	✓	✓				✓	◐	✓					✓		
RaceFuzzer [197]	◐	✓	✓			✓										
RedQueen [26]	◐	✓					✓	◐						✓		
Steelix [143]	◐[†]			✓		✓	✓[†]	◐					✓	✓[†]		✓
Syzkaller [226]	◐	✓	✓				✓	●	✓				✓	✓		✓
Angora [59]	◐+○	✓	✓					◐			✓			✓		
Cyberdyne [95]	◐+○	✓		✓			✓	●		✓			✓	✓		✓
DigFuzz [249]	◐+○						✓	●		✓			✓	✓		✓
Driller [209]	◐+○	✓					✓	●		✓			✓	✓		✓
QSYM [240]	◐+○	✓					✓	●		✓			✓	✓		✓
T-Fuzz [177]	◐+○	✓		✓		✓	✓[†]	●		✓			✓	✓		✓
VUzzer [183]	◐+○	✓				✓	✓				✓			✓	◐	
BitFuzz [47]	○						✓			✓						
BuzzFuzz [87]	○		✓				✓	◐		✓	✓					
CAB-Fuzz [129]	○						✓			✓						
Chopper [219]	○	✓	✓				✓			✓						
Dewey et al. [73]	○	✓	✓						✓	✓						
Dowser [101]	○					✓				✓	✓					
GRT [150]	○		✓				✓		✓	✓	✓				○	
KLEE [49]	○	✓	✓							✓						
MoWF [179]	○								✓	✓						
MutaGen [127]	○				●			●								
Narada [188]	○	✓	✓				✓									
SAGE [93]	○									✓						
TaintScope [229]	○					✓	✓	◐			✓			✓		

[†] The corresponding fuzzer is derived from AFL, and it changed this part of the fuzzing algorithm.

whereas the latter happens while the PUT is running (INPUTEVAL). For the convenience of the reader, here we will present both static and dynamic instrumentation together.

Static instrumentation is often performed at compile time on either source code or intermediate code. Since it occurs before runtime, it generally imposes less runtime overhead than dynamic instrumentation. If the PUT relies on libraries, these have to be separately instrumented, commonly by recompiling them with the same instrumentation. Beyond source-based instrumentation, researchers have also developed binary-level static instrumentation (i.e., binary rewriting) tools [248], [136], [80].

Despite having a higher overhead than static instrumentation, dynamic instrumentation has the advantage of easily instrumenting dynamically-linked libraries because it is performed at runtime. There are several well-known dynamic instrumentation tools such as DynInst [180], DynamoRIO [45], Pin [149], Valgrind [169], and QEMU [36].

A given fuzzer can support more than one type of instrumentation. For example, AFL supports static instrumentation at the source code level with a modified compiler, or dynamic instrumentation at the binary level with the help of QEMU [36]. When using dynamic instrumentation, AFL can instrument either (1) executable code in the PUT itself, which is the default setting, or (2) executable code in the PUT and any external libraries (with the AFL_INST_LIBS option). The second option—instrumenting all encountered code—can report coverage information for code in external libraries, and thus provides a more complete picture of coverage. However, this also means AFL will fuzz additional paths in external library functions and some users may prefer avoiding this option.

3.1.1 Execution Feedback

Grey-box fuzzers typically take execution feedback as input to evolve test cases. LibFuzzer [9], AFL, and its descendants compute branch coverage by instrumenting every branch instruction in the PUT. However, they store the branch coverage information in a compact bit vector, which can become inaccurate due to path collisions. CollAFL [86] recently addressed this shortcoming by introducing a new path-sensitive hash function. Meanwhile, Syzkaller [226] uses node coverage as their execution feedback, whereas Honggfuzz [213] allows users to choose which execution feedback to use.

3.1.2 Thread Scheduling

Race condition bugs can be difficult to trigger as they rely on non-deterministic behaviors, which may occur very infrequently. However, instrumentation can also be used to trigger different non-deterministic program behaviors by explicitly controlling how threads are scheduled [196], [197], [175], [120], [135], [50], [188]. Existing work has shown that even randomly scheduling threads can be effective at finding race condition bugs [196].

3.1.3 In-Memory Fuzzing

When testing a large program, it is sometimes desirable to fuzz *only a portion* of the PUT without re-spawning a process for each fuzz iteration in order to minimize execution overhead. For example, complex (e.g., GUI) applications often require several seconds of processing before they accept input. One approach to fuzzing such programs is to take a snapshot of the PUT after the GUI is initialized. To fuzz a new test case, one can then restore the memory snapshot before writing the new test case directly into memory and executing it. The same intuition applies to fuzzing network applications that involve heavy interaction between client and server. This technique is called in-memory fuzzing [108]. As an example, GRR [220], [95] creates a snapshot before loading any input bytes. This way, it can skip over unnecessary startup code. AFL also employs a fork server to avoid some of the process startup costs. Although it has the same motivation as in-memory fuzzing, a fork server involves forking off a new process for every fuzz iteration (see §6).

Some fuzzers [9], [241] perform in-memory fuzzing on a function without restoring the state of the PUT after each iteration. We call such a technique *in-memory API fuzzing*. For example, AFL has an option called persistent mode [243], which repeatedly performs in-memory API fuzzing in a loop without restarting the process. In this case, AFL ignores potential side-effects from the function being called multiple times in the same execution.

Although efficient, in-memory API fuzzing suffers from unsound fuzzing results: bugs (or crashes) found with in-memory fuzzing may *not* be reproducible, because (1) it is not always feasible to construct a valid calling context for the target function, and (2) there can be side-effects that are not captured across multiple function calls. Notice the soundness of in-memory API fuzzing mainly depends on a good entry point function, and finding such a function can be a challenging task.

3.2 Seed Selection

Recall from §2 that fuzzers receive a set of fuzz configurations controlling the behavior of the fuzzing algorithm. However, some parameters of fuzz configurations, such as seeds for mutation-based fuzzers, can have large or even infinite domains. For example, suppose an analyst fuzzes an MP3 player that accepts MP3 files as input. Since there are an unbounded number of valid MP3 files, which seed(s) should we use for fuzzing? This problem of reducing the size of the initial seed pool is known as the *seed selection problem* [184].

Several approaches and tools exist to address this problem [184], [79]. A common approach, which is known as *minset*, finds a minimal set of seeds that maximizes a coverage metric such as node coverage. For example, suppose the current set of configurations $\mathbb{C}$ consists of two seeds s_1 and s_2 that cover the following addresses of the PUT: $\{s_1 \rightarrow \{10, 20\}, s_2 \rightarrow \{20, 30\}\}$. If we have a third seed $s_3 \rightarrow \{10, 20, 30\}$ that executes roughly as fast as s_1 and s_2, one could argue it makes sense to fuzz s_3 instead of s_1 and s_2 because s_3 tests the same set of code at half the execution time cost. This intuition is supported by Miller's report [158], which showed that a 1% increase in code coverage increased the percentage of bugs found by .92%. As is noted in §7.2, this step can also be part of CONFUPDATE, which is useful for fuzzers introducing new seeds into the seed pool throughout the campaign.

Fuzzers use a variety of different coverage metrics in practice. For example, AFL's minset is based on branch

coverage with a logarithmic counter on each branch. The rationale behind this is to consider branch counts as different only when they differ in their orders of magnitude. Honggfuzz [213] computes coverage based on the number of executed instructions, executed branches, and unique basic blocks. This metric allows the fuzzer to add longer executions to the minset, which can help discover denial of service vulnerabilities or performance problems.

3.3 Seed Trimming

Smaller seeds are likely to consume less memory and entail higher throughput. Therefore, some fuzzers attempt to reduce the size of seeds prior to fuzzing them, which is called *seed trimming*. Seed trimming can happen prior to the main fuzzing loop in PREPROCESS or as part of CONFUPDATE. One notable fuzzer that uses seed trimming is AFL [241], which uses its code coverage instrumentation to iteratively remove a portion of the seed as long as the modified seed achieves the same coverage. Meanwhile, Rebert et al. [184] reported that their size minset algorithm, which selects seeds by giving higher priority to smaller seeds in size, resulted in fewer unique bugs compared to a random seed selection. For the specific case of fuzzing Linux system call handlers, MoonShine [173] extends Syzkaller [226] to reduce the size of seeds while preserving the dependencies between calls which are detected using a static analysis.

3.4 Preparing a Driver Application

When it is difficult to directly fuzz the PUT, it makes sense to prepare a driver for fuzzing. This process is largely manual in practice although this is done only once at the beginning of a fuzz campaign. For example, when our target is a library, we need to prepare for a driver program that calls functions in the library. Similarly, kernel fuzzers may fuzz userland applications to test kernels [129], [171], [34]. IoTFuzzer [57] targets IoT devices by letting the driver communicate with the corresponding smartphone application.

4 SCHEDULING

In fuzzing, scheduling means selecting a fuzz configuration for the next fuzz iteration. As explained in §2.1, the content of a fuzz configuration depends on the type of the fuzzer. For simple fuzzers, scheduling can be straightforward— for example, zzuf [107] in its default mode allows only one configuration and thus there is simply no decision to make. But for more advanced fuzzers such as BFF [52] and AFLFast [40], a major factor to their success lies in their innovative scheduling algorithms. In this section, we will discuss scheduling algorithms for black- and grey-box fuzzing only; scheduling in white-box fuzzing requires a complex setup unique to symbolic executors and we refer the reader to another source [41].

4.1 The Fuzz Configuration Scheduling (FCS) Problem

The goal of scheduling is to analyze the currently-available information about the configurations and pick one that is more likely to lead to the most favorable outcome, e.g., finding the most number of unique bugs, or maximizing the coverage attained by the set of generated inputs. Fundamentally, every scheduling algorithm confronts with the same *exploration vs. exploitation* conflict—time can either be spent on gathering more accurate information on each configuration to inform future decisions (explore), or on fuzzing the configurations that are currently believed to lead to more favorable outcomes (exploit). Woo et al. [235] dubbed this inherent conflict the Fuzz Configuration Scheduling (FCS) Problem.

In our model fuzzer (Algorithm 1), the function SCHEDULE selects the next configuration based on (i) the current set of fuzz configurations $\mathbb{C}$, (ii) the current time t_{elapsed}, and (iii) the total time budget t_{limit}. This configuration is then used for the next fuzz iteration. Notice that SCHEDULE is only about decision-making. The information on which this decision is based is acquired by PREPROCESS and CONFUPDATE, which augment $\mathbb{C}$ with this knowledge.

4.2 Black-box FCS Algorithms

In the black-box setting, the only information an FCS algorithm can use is the fuzz outcomes of a configuration—the number of crashes and bugs found with it and the amount of time spent on it so far. Householder and Foote [111] were the first to study how such information can be leveraged in the CERT BFF black-box mutational fuzzer [52]. In their work, they modeled black-box mutational fuzzing as a sequence of Bernoulli trials. By favoring configurations with higher observed success probabilities (#unique crashes / #runs), they demonstrated an increase in the number of unique crashes found by BFF in a fixed amount of time.

This result was further improved by Woo et al. [235] on multiple fronts. First, they refined the model to become *Weighted Coupon Collector's Problem with Unknown Weights* (WCCP/UW), which learns a decaying upper-bound on the success probability of each trial. Second, they applied *multi-armed bandit* (MAB) algorithms to fuzzing, which is a common coping strategy when faced with the exploration vs. exploitation conflict [37]. Third, they normalized the success probability of a configuration by the time already spent in it, thereby preferring the faster configurations. Fourth, they redefined a fuzz iteration from running a fixed number of fuzz runs to a fixed amount of time, further deprioritizing slower configurations.

4.3 Grey-box FCS Algorithms

In the grey-box setting, an FCS algorithm can choose to use a richer set of information about each configuration, e.g., the coverage attained when fuzzing a configuration. AFL [241] is the forerunner in this category and it is based on an evolutionary algorithm (EA). Intuitively, an EA maintains a population of configurations, each with some value of "fitness". An EA selects fit configurations and applies them to genetic transformations such as mutation and recombination to produce offspring, which may later become new configurations. The hypothesis is that these produced configurations are more likely to be fit.

To understand FCS in the context of an EA, we need to define (i) what makes a configuration fit, (ii) how configurations are selected, and (iii) how a selected configuration is used. As a high-level approximation, among the configurations

that exercise a control-flow edge, AFL considers the one that contains the fastest and smallest input to be fit ("favorite" in AFL parlance). AFL maintains a queue of configurations, from which it selects the next fit configuration *essentially* as if the queue is circular. Once selected, AFL allocates more runs to configurations which are fastest and have a higher branch coverage. From the perspective of FCS, notice the preference for fast configurations is common with the black-box setting.

AFLFast by Böhme et al. [40] has improved upon AFL in all three aspects. To start, it modifies configuration fitness setting and selection to prioritize exploration of new and rare paths. Moreover, AFLFast fuzzes a selected configuration a variable number of times as determined by a *power schedule*. Its FAST power schedule starts with a small "energy" value to ensure initial exploration among configurations and increases exponentially up to a limit to quickly ensure sufficient exploitation. Finally, it also normalizes the energy by the number of generated inputs that exercise the same path, thus promoting explorations of less-frequently fuzzed configurations.

The innovations in AFLFast have been partially integrated in AFL as FidgetyAFL [5]. Zalewski found the largest improvement of AFLFast was to quickly go over all newly-added seeds. As such, AFL now spend less time on each seed. In related works, AFLGo [39] extends AFLFast by modifying its priority attribution in order to target specific program locations. Hawkeye [56] further improves directed fuzzing by leveraging a static analysis in its seed scheduling and input generation. FairFuzz [141] guides the campaign to exercise rare branches by employing a mutation mask for each pair of a seed and a rare branch. QTEP [228] uses static analysis to infer which part of the binary is more "faulty" and prioritizes configurations that cover them.

5 INPUT GENERATION

Since the content of a test case directly controls whether or not a bug is triggered, the technique used for *input generation* is naturally one of the most influential design decisions in a fuzzer. Traditionally, fuzzers are categorized into either generation- or mutation-based fuzzers [212]. Generation-based fuzzers produce test cases based on a given model that describes the inputs expected by the PUT. We call such fuzzers *model-based* fuzzers in this paper. On the other hand, mutation-based fuzzers produce test cases by mutating a given *seed* input. Mutation-based fuzzers are generally considered to be *model-less* because seeds are merely example inputs and even in large numbers they do not completely describe the expected input space of the PUT. In this section, we explain and classify the various input generation techniques used by fuzzers based on the underlying test case generation (INPUTGEN) mechanism. Since, the only input of this function a configuration conf, as shown in Algorithm 1, it is based on information collected by PREPROCESS or CONFUPDATE.

5.1 Model-based (Generation-based) Fuzzers

Model-based fuzzers generate test cases based on a given model that describes the inputs or executions that the PUT may accept, such as a grammar precisely characterizing the input format or less precise constraints such as magic values identifying file types.

5.1.1 Predefined Model

Some fuzzers use a model that can be configured by the user. For example, Peach [79], PROTOS [124], and Dharma [4] take in a specification provided by the user. Autodafé [224], Sulley [19], SPIKE [16], SPIKEfile [211], and LibFuzzer [9], [12] expose APIs that allow analysts to create their own input models. Tavor [250] also takes in an input specification written in Extended Backus-Naur form (EBNF) and generates test cases conforming to the corresponding grammar. Similarly, network protocol fuzzers such as PROTOS [124], SNOOZE [32], KiF [15], and T-Fuzz [118] also take in a protocol specification from the user. Kernel API fuzzers [119], [226], [163], [168], [231] define an input model in the form of system call templates. These templates commonly specify the number and types of arguments a system call expects as inputs. The idea of using a model in kernel fuzzing originated in Koopman et al.'s seminal work [132], where they compared the robustness of OSes with a finite set of manually chosen test cases for system calls. Nautilus [25] employs grammar-based input generation for general-purpose fuzzing and also uses its grammar for seed trimming (see §3.3).

Other model-based fuzzers target a specific language or grammar, and the model of this language is built into the fuzzer itself. For example, cross_fuzz [242] and DOMfuzz [194] generate random Document Object Model (DOM) objects. Likewise, jsfunfuzz [194] produces random, but syntactically-correct JavaScript code based on its own grammar model. QuickFuzz [97] utilizes existing Haskell libraries that describe file formats when generating test cases. Some network protocol fuzzers such as Frankencerts [44], TLS-Attacker [203], tlsfuzzer [128], and llfuzzer [207] are designed with models of specific network protocols such as TLS and NFC. Dewey et al. [72], [73] proposed a way to generate test cases that are not only grammatically correct, but also semantically diverse by leveraging constraint logic programming. LangFuzz [109] produces code fragments by parsing a set of seeds that are given as input. It then randomly combines the fragments and mutates seeds with the fragments to generate test cases. Since it is provided with a grammar, it always produces syntactically-correct code. Whereas LangFuzz was applied to JavaScript and PHP, BlendFuzz [239] targets XML and regular expression parsers and is based on similar ideas as LangFuzz.

5.1.2 Inferred Model

Inferring the model rather than relying on a predefined or user-provided model has recently been gaining traction. Although there is an abundance of published research on the topic of automated input format and protocol reverse engineering [69], [48], [146], [66], [31], only a few fuzzers leverage these techniques. Similar to instrumentation (§3.1), model inference can occur in either PREPROCESS or CONFUPDATE.

5.1.2.1 Model Inference in PREPROCESS: Some fuzzers infer the model as a preprocessing step. Test-Miner [70] searches for the data available in the PUT, such as literals, to predict suitable inputs. Given a set of seeds and a grammar, Skyfire [227] uses a data-driven approach to

infer a probabilistic context-sensitive grammar and then uses it to generate a new set of seeds. Unlike previous works, it focuses on generating semantically-valid inputs. IMF [103] learns a kernel API model by analyzing system API logs, and it produces C code that invokes a sequence of API calls using the inferred model. CodeAlchemist [104] breaks JavaScript code into "code bricks" and computes assembly constraints, which describe when distinct bricks can be assembled or merged together to produce semantically-valid test cases. These constraints are computed using both static and dynamic analyses. Neural [65] and Learn&Fuzz [94] use a neural network-based machine learning technique to learn a model from a given set of test files and then generate test cases from the inferred model. Liu et al. [147] proposed a similar approach that is specific to text inputs.

5.1.2.2 *Model Inference in* CONFUPDATE: Other fuzzers can potentially update their model after each fuzz iteration. PULSAR [88] automatically infers a network protocol model from a set of captured network packets generated from a program. The learned network protocol is then used to fuzz the program. PULSAR internally builds a state machine and maps which message token is correlated with a state. This information is later used to generate test cases that cover more states in the state machine. Doupé et al. [76] propose a way to infer the state machine of a web service by observing the I/O behavior. The inferred model is then used to scan for web vulnerabilities. The work of Ruiter et al. [187] is similar, but it targets TLS and bases its implementation on LearnLib [181]. GLADE [33] synthesizes a context-free grammar from a set of I/O samples and fuzzes the PUT using the inferred grammar. go-fuzz [225] is a grey-box fuzzer, which builds a model for each of the seed it adds to its seed pool. This model is used to generate new inputs from this seed. In order to help with the limitation of symbolic execution, Shen et al. [201] use neural networks to solve difficult branch conditions.

5.1.3 Encoder Model

Fuzzing is often used to test *decoder* programs which parse a certain file format. Many file formats have corresponding *encoder* programs, which can be thought of as an implicit model of the file format. MutaGen [127] leverages the implicit model contained in encoder programs to generate new test cases. Unlike most mutation-based fuzzers, which mutate an existing *test case* (see §5.2) to generate test cases, MutaGen mutates the *encoder program*. Specifically, to produce a new test case MutaGen computes a dynamic program slice of the encoder program and runs it. The underlying idea is that the program slices will slightly change the behavior of the encoder program so that it produces test cases that are slightly malformed.

5.2 Model-less (Mutation-based) Fuzzers

Classic random testing [23], [102] is not efficient in generating test cases that satisfy specific path conditions. Suppose there is a simple C statement: if (input == 42). If input is a 32-bit integer, the probability of randomly guessing the right input value is $1/2^{32}$. The situation gets worse when we consider well-structured inputs such as MP3 files. Indeed, it is extremely unlikely that random testing will generate a

valid MP3 file as a test case in any reasonable amount of time. As a result, the MP3 player will most likely reject the generated test cases from random testing at the parsing stage before reaching deeper parts of the program.

This problem motivates the use of seed-based input generation as well as white-box input generation (see §5.3). Most model-less fuzzers use a *seed*, which is an input to the PUT, in order to generate test cases by modifying the seed. A seed is typically a well-structured input of a type supported by the PUT: a file, a network packet, or a sequence of UI events. By mutating only a fraction of a valid file, it is often possible to generate a new test case that is mostly valid, but also contains abnormal values to trigger crashes of the PUT. There are a variety of methods used to mutate seeds and we describe the common ones below.

5.2.1 Bit-Flipping

Bit-flipping is a common technique used by many model-less fuzzers [241], [213], [107], [8], [106]. Some fuzzers simply flip a fixed number of bits, while others determine the number of bits to flip at random. To randomly mutate seeds, some fuzzers employ a user-configurable parameter called the *mutation ratio*, which determines the number of bit positions to flip for a single execution of INPUTGEN. To flip K random bits in a given N-bit seed, the mutation ratio is K/N.

SymFuzz [55] showed fuzzing performance is sensitive to the mutation ratio and there is not a single ratio suitable for all PUTs. Fortunately, there are several ways to set a good mutation ratio. BFF [52] and FOE [53] use an exponentially scaled set of mutation ratios for each seed and allocate more iterations to ratios that proved to be statistically effective [111]. SymFuzz leverages a white-box program analysis to infer a good mutation ratio for each seed.

5.2.2 Arithmetic Mutation

AFL [241] and honggfuzz [213] contain another mutation operation where they consider a selected byte sequence as an integer and perform simple arithmetic on that value. The computed value is then used to replace the selected byte sequence. The key intuition is to bound the effect of mutation by a small number. For example, AFL may select a 4-byte value from a seed and treat the value as an integer i. It then replaces the value in the seed with $i \pm r$, where r is a randomly-generated small integer. The range of r depends on the fuzzer and is often user-configurable. In AFL, the default range is $0 \leq r < 35$.

5.2.3 Block-based Mutation

There are several block-based mutation methodologies, where a block is a sequence of bytes of a seed: (1) insert a randomly-generated block into a random position of a seed [241], [9]; (2) delete a randomly-selected block from a seed [241], [106], [213], [9]; (3) replace a randomly-selected block with a random value [241], [213], [106], [9]; (4) randomly permute the order of a sequence of blocks [106], [9]; (5) resize a seed by appending a random block [213]; and (6) take a random block from a seed to insert/replace a random block of another seed [241], [9].

5.2.4 Dictionary-based Mutation

Some fuzzers use a set of predefined values with potentially significant semantic meaning for mutation. For example, AFL [241], honggfuzz [213], and LibFuzzer [9] use values such as 0, −1, and 1 when mutating integers. Radamsa [106] employs Unicode strings and GPF [8] uses formatting characters such as `%x` and `%s` to mutate strings [58].

5.3 White-box Fuzzers

White-box fuzzers can also be categorized into either model-based or model-less fuzzers. For example, traditional dynamic symbolic execution [93], [116], [30], [152], [209] does not require any model as in mutation-based fuzzers, but some symbolic executors [91], [179], [129] leverage input models such as a grammar to guide the symbolic executor.

Although many white-box fuzzers including the seminal work by Godefroid et al. [93] use dynamic symbolic execution to generate test cases, not all white-box fuzzers are dynamic symbolic executors. Some fuzzers [229], [55], [150], [189] leverage a white-box program analysis to find information about the inputs a PUT accepts in order to use it with black- or grey-box fuzzing. In the rest of this subsection, we briefly summarize the existing white-box fuzzing techniques based on their underlying test case generation algorithm. Note that as we have mentioned in §2.2, we intentionally omit dynamic symbolic executors such as [92], [198], [63], [49], [218], [54] because their authors did not explicitly describe their work as a fuzzer.

5.3.1 Dynamic Symbolic Execution

At a high level, classic symbolic execution [130], [42], [112] runs a program with symbolic values as inputs, which represents all possible values. As it executes the PUT, it builds symbolic expressions instead of evaluating concrete values. Whenever it reaches a conditional branch instruction, it conceptually forks two symbolic interpreters, one for the true branch and another for the false branch. For every path, a symbolic interpreter builds up a path formula (or path predicate) for every branch instruction it encountered during an execution. A path formula is satisfiable if there is a concrete input that executes the desired path. One can generate concrete inputs by querying an SMT solver [160] for a solution to a path formula. Dynamic symbolic execution is a variant of traditional symbolic execution, where both symbolic execution and concrete execution operate at the same time. Thus, we often refer to dynamic symbolic execution as concolic (concrete + symbolic) testing. The idea is that concrete execution states can help reduce the complexity of symbolic constraints. An extensive review of the academic literature of dynamic symbolic execution beyond its application to fuzzing is out of the scope of this paper. A broader treatment of dynamic symbolic execution can be found in other sources [20], [192].

Dynamic symbolic execution is slow compared to grey-box or black-box approaches as it involves instrumenting and analyzing every instruction of the PUT. To cope with the high cost, a common strategy has been to narrow its usage, for instance, by letting the user to specify uninteresting parts of the code [219] or by alternating between concolic testing and grey-box fuzzing. Driller [209] and Cyberdyne [95]

have shown the usefulness of this technique at the DARPA Cyber Grand Challenge. QSYM [240] seeks to improve the integration between grey- and white-box fuzzing by implementing a fast concolic execution engine. DigFuzz [249] optimizes the switch between grey- and white-box fuzzing by first estimating the probability of exercising each path using grey-box fuzzing. This allows it to let its white-box fuzzer focus on the paths that are believed to be most challenging for grey-box fuzzing.

5.3.2 Guided Fuzzing

Some fuzzers leverage static or dynamic program analysis techniques to enhance the effectiveness of fuzzing. These techniques usually involve fuzzing in two phases: (i) a costly program analysis for obtaining useful information about the PUT, and (ii) test case generation with the guidance from the previous analysis. This is denoted in column 6 of Table 1 (p. 6). For example, TaintScope [229] uses a fine-grained taint analysis to find "hot bytes", which are the input bytes that flow into critical system calls or API calls. A similar idea is presented by other security researchers [78], [114]. Dowser [101] performs a static analysis during compilation to find loops that are likely to contain bugs based on a heuristic. Specifically, it looks for loops containing pointer dereferences. It then computes the relationship between input bytes and the candidate loops with a taint analysis. Finally, Dowser runs dynamic symbolic execution while making only the critical bytes to be symbolic hence improving performance. VUzzer [183] and GRT [150] leverage both static and dynamic analysis techniques to extract control- and data-flow features from the PUT and use them to guide input generation.

Angora [59] and RedQueen [26] decrease the cost of their analysis by first running each seed with a costly instrumentation and using this information for generating inputs which are run with a lighter instrumentation. Angora improves upon the "hot bytes" idea by using taint analysis to associate each path constraint to corresponding bytes. It then performs a search inspired by gradient descent algorithm to guide its mutations towards solving these constraints. On the other hand, RedQueen tries to detect how inputs are used in the PUT by instrumenting all comparisons and looking for correspondence between their operands and the given input. Once a match is found, it can be used to solve a constraint.

5.3.3 PUT Mutation

One of the practical challenges in fuzzing is bypassing a checksum validation. For example, when a PUT computes a checksum of an input before parsing it, many test cases will be rejected by the PUT. To handle this challenge, TaintScope [229] proposed a checksum-aware fuzzing technique, which identifies a checksum test instruction with a taint analysis and patches the PUT to bypass the checksum validation. Once they find a program crash, they generate the correct checksum for the input to generate a test case that crashes the unmodified PUT. Caballero et al. [47] suggested a technique called stitched dynamic symbolic execution that can generate test cases in the presence of checksums.

T-Fuzz [177] extends this idea to efficiently penetrate all kind of conditional branches. It first builds a set of branches that can be transformed without modifying the program logic, dubbed Non-Critical Checks (NCCs). When the fuzz

campaign stops discovering new paths, it picks an NCC, transforms it and then restarts a fuzz campaign on the modified PUT. Finally, after a crash has been found with fuzzing on a modified PUT, T-Fuzz tries to reconstruct it on the original PUT with symbolic execution.

6 INPUT EVALUATION

After an input is generated, the fuzzer executes the PUT on the input and decides what to do with the resulting execution. This process is called *input evaluation*. Although the simplicity of executing a PUT is one of the reasons that makes fuzzing attractive, there are many optimizations and design decisions related to input evaluation that effect the performance and effectiveness of a fuzzer. We will explore these considerations in this section.

6.1 Bug Oracles

The canonical security policy used with fuzz testing considers every program execution terminated by a fatal signal (such as a segmentation fault) to be a violation. This policy detects many memory vulnerabilities since a memory vulnerability that overwrites data or code pointer with an invalid value usually causes a segmentation fault or abort when it is dereferenced. Moreover, this policy is efficient and simple because operating systems allow such exceptional situations to be trapped by the fuzzer without any instrumentation. However, the traditional policy of detecting crashes will not detect some memory vulnerability that has been triggered. For example, if a stack buffer overflow overwrites a pointer on the stack with a valid memory address, the program might run to completion with an invalid result rather than crashing but the fuzzer would not detect this. As a mitigation, researchers have proposed a variety of efficient program transformations to detect unsafe or unwanted program behaviors and abort the program. These are often called *sanitizers* [205].

6.1.1 Memory and Type Safety

Memory safety errors can be separated into two classes: spatial and temporal. Spatial memory errors occur when a pointer is dereferenced outside of the object it was intended to point to. For example, buffer overflows and underflows are canonical examples of spatial memory errors. Temporal memory errors occur when a pointer is accessed after it is no longer valid. For example, a use-after-free vulnerability, in which a pointer is used after the memory it pointed to has been deallocated, is a typical temporal memory error.

Address Sanitizer (ASan) [199] is a fast memory error detector that instruments programs at compile time. ASan can detect spatial and temporal memory errors and has an average slowdown of only 73%, making it an attractive alternative to a basic crash harness. ASan employs a shadow memory that allows each memory address to be quickly checked for validity before it is dereferenced, which allows it to detect many (but not all) unsafe memory accesses, even if they would not crash the original program. MEDS [105] improves on ASan by leveraging the large memory space available on 64-bit platforms to create large chunks of inaccessible memory *red-zones* in-between allocated objects.

These red-zones make it more likely that a corrupted pointer will point to invalid memory and cause a crash.

SoftBound/CETS [165], [166] is another memory error detector that instruments programs during compilation. Rather than tracking valid memory addresses like ASan, SoftBound/CETS associates bounds and temporal information with each pointer and can thus detect all spatial and temporal memory errors in theory. However, as expected, this completeness comes with a higher average overhead of 116% [166]. CaVer [138], TypeSan [100] and HexType [117] instrument programs during compilation so that they can detect *bad-casting* in C++ type casting. Bad casting occurs when an object is cast into an incompatible type, such as when an object of a base class is cast into a derived type.

Another class of memory safety protection is *Control Flow Integrity* [13], [14] (CFI), which detects control flow transitions at runtime that are not possible in the original program. CFI can be used to detect test cases that have illegally modified the control flow of a program. A recent project focused on protecting against a subset of CFI violations has landed in the mainstream `gcc` and `clang` compilers [217].

6.1.2 Undefined Behaviors

Languages such as C contain many behaviors that are left undefined by the language specification. The compiler is free to handle these constructs in a variety of ways. In many cases, a programmer may (intentionally or otherwise) write their code so that it is only correct for some compiler implementations. Although this may not seem overly dangerous, many factors can impact how a compiler implements undefined behaviors, including optimization settings, architecture, compiler, and even compiler version. Vulnerabilities and bugs often arise when the compiler's implementation of an undefined behavior does not match the programmer's expectation [230], [3].

Memory Sanitizer (MSan) is a tool that instruments programs during compilation to detect undefined behaviors caused by uses of uninitialized memory in C and C++ [208]. Similar to ASan, MSan uses a shadow memory that represents whether each addressable bit is initialized or not. Memory Sanitizer has approximately 150% overhead. Undefined Behavior Sanitizer (UBSan) [74] modifies programs at compile-time to detect undefined behaviors. Unlike other sanitizers which focus on one particular source of undefined behavior, UBSan can detect a wide variety of undefined behaviors, such as using misaligned pointers, division by zero, dereferencing null pointers, and integer overflow. Thread Sanitizer (TSan) [200] is a compile-time modification that detects data races. A data race occurs when two threads concurrently access a shared memory location and at least one of the accesses is a write. Such bugs can cause data corruption and can be extremely difficult to reproduce due to non-determinism.

6.1.3 Input Validation

Testing for input validation vulnerabilities such as XSS (cross-site scripting) and SQL injection vulnerabilities is a challenging problem. It requires understanding the behavior of the very complicated parsers that power web browsers and database engines. KameleonFuzz [77] detects successful XSS attacks by parsing inputs with a real web browser,

extracting the Document Object Model tree, and comparing it against manually specified patterns which indicate a successful attack. μ4SQLi [21] uses a similar trick to detect SQL injections. Because it is not possible to reliably detect SQL injections from a web application response, μ4SQLi uses a database proxy that intercepts communication between the target web application and the database to detect whether an input triggered harmful behavior.

6.1.4 Semantic Difference

Semantic bugs are often discovered using a technique called *differential testing* [153], which compares the behavior of similar (but not identical) programs. Several fuzzers [44], [178], [62] have used differential testing to identify discrepancies between similar programs, which are likely to indicate a bug. Jung et al. [122] introduced *black-box differential fuzz testing*, which uses differential testing of multiple inputs on a single program to map mutations from the PUT's input to its output. These mappings are used to identify information leaks.

6.2 Execution Optimizations

Our model considers individual fuzz iterations to be executed sequentially. While the straightforward implementation of such an approach would simply load the PUT every time a new process is started at the beginning of a fuzz iteration, the repetitive loading processes can be significantly reduced. To this end, modern fuzzers provide functionalities that skip over these repetitive loading processes. For example, AFL [241] provides a fork-server that allows each new fuzz iteration to fork from an already-initialized process. Similarly, in-memory fuzzing is another way to optimize the execution speed as discussed in §3.1.3. Regardless of the exact mechanism, the overhead of loading and initializing the PUT is reduced over many iterations. Xu et al. [238] further lower the cost of an iteration by designing a new system call that replaces `fork()`.

6.3 Triage

Triage is the process of analyzing and reporting test cases that cause policy violations. Triage can be separated into three steps: deduplication, prioritization, and minimization.

6.3.1 Deduplication

Deduplication is the process of pruning any test case from the output set that triggers the same bug as another test case. Ideally, deduplication would return a set of test cases in which each triggers a unique bug.

Deduplication is an important component of most fuzzers for several reasons. As a practical implementation manner, it avoids wasting disk space and other resources by storing duplicate results on the hard drive. As a usability consideration, deduplication makes it easy for users to understand roughly how many different bugs are present and to be able to analyze an example of each bug. This is useful for a variety of fuzzer users; for example, attackers may want to look for only "home run" vulnerabilities, meaning those that are likely to lead to reliable exploitation.

There are currently three major deduplication implementations used in practice: stack backtrace hashing, coverage-based deduplication, and semantics-aware deduplication.
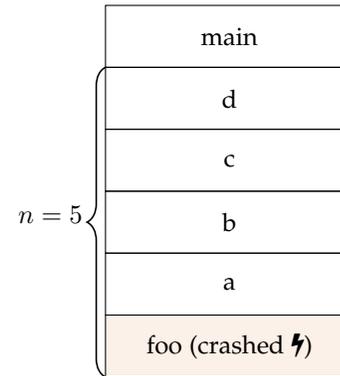

Fig. 2: Stack backtrace hashing example.

6.3.1.1 Stack Backtrace Hashing: Stack backtrace hashing [159] is one of the oldest and most-widely used methods for deduplicating crashes. In this method, an automated tool records a stack backtrace at the time of the crash and assigns a *stack hash* based on the contents of that backtrace. For example, if the program crashed while executing a line of code in function `foo` and the call stack was `main` → `d` → `c` → `b` → `a` → `foo` (see Fig. 2), then a stack backtrace hashing implementation with $n = 5$ would group that test case with other crashing executions whose backtrace ended with `d` → `c` → `b` → `a` → `foo`.

Stack hashing implementations vary widely, starting with the number of stack frames that are included in the hash. Some implementations use one [22], three [159], [235], five [85], [52], or do not have any limit [127]. Implementations also differ in the amount of information included from each stack frame. Some implementations will only hash the function's name or address, but other implementations will hash both the name and the offset or line. Neither option works well all the time, and so some implementations [155], [85] produce two hashes: a major and minor hash. The major hash is likely to group dissimilar crashes together as it only hashes the function name, whereas the minor hash is more precise since it uses the function name and line number and also includes an unlimited number of stack frames.

Although stack backtrace hashing is widely used, it is not without its shortcomings. The underlying hypothesis of stack backtrace hashing is that similar crashes are caused by similar bugs, and vice versa. However, to the best of our knowledge, this hypothesis has never been directly tested. There is some reason to doubt its veracity: some crashes do not occur near the code that caused the crash. For example, a vulnerability that causes heap corruption might only cause a crash when an unrelated part of the code attempts to allocate memory and not when the heap overflow occurred.

6.3.1.2 Coverage-based Deduplication: AFL [241] is a popular grey-box fuzzer that employs an efficient source-code instrumentation to record the edge coverage of each execution of the PUT and measure coarse hit counts for each edge. As a grey-box fuzzer, AFL primarily uses this coverage information to select new seed files. However, it also leads to a fairly unique deduplication scheme as well. As described by its documentation, AFL considers a crash to be unique if either (i) the crash covered a previously unseen edge, or (ii) the crash did *not* cover an edge that was present in all

earlier crashes [245].

6.3.1.3 Semantics-aware Deduplication: RETracer [68] performs crash triage based on the semantics recovered from a reverse data-flow analysis on each crash. Specifically, RETracer checks which pointer caused the crash by analyzing a crash dump (core dump) and recursively identifies which instruction assigned the bad value to it. It eventually finds a function that has the maximum frame leveland "blames" that function. The blamed function is then used to cluster crashes. Van Tonder et al. [222] leverages Automated Program Repair [137] techniques to map crashing test cases to bugs as a function of change in program semantics. This change approximates fixing the root cause of the bug.

6.3.2 Prioritization and Exploitability

Prioritization, a.k.a. the fuzzer taming problem [61], is the process of ranking or grouping violating test cases according to their severity and uniqueness. Fuzzing has traditionally been used to discover memory vulnerabilities, and in this context prioritization is better known as determining the *exploitability* of a crash. It informally describes the likelihood that a practical exploit could be developed for the vulnerability exposed by the test case. Both defenders and attackers are interested in exploitable bugs. Defenders generally fix exploitable bugs before non-exploitable ones, and attackers are interested in exploitable bugs for obvious reasons.

One of the first exploitability ranking systems was Microsoft's !exploitable [155], which gets its name from the !exploitable WinDbg command that it provides. !exploitable employs several heuristics paired with a simplified taint analysis [170], [192]. It classifies each crash on the following severity scale: EXPLOITABLE > PROBABLY_EXPLOITABLE > UNKNOWN > NOT_LIKELY_EXPLOITABLE, where $x > y$ means that x is more severe than y. Although these classifications are not formally defined, !exploitable is informally intended to be conservative and err on the side of reporting something as more exploitable than it is. For example, !exploitable concludes that a crash is EXPLOITABLE if an illegal instruction is executed, based on the assumption that the attacker was able to coerce control flow. On the other hand, a division by zero crash is considered NOT_LIKELY_EXPLOITABLE.

Since !exploitable was introduced, other similar rule-based heuristics systems have been proposed, including the exploitable plugin for GDB [85] and Apple's Crash-Wrangler [22]. However, their correctness has not been systematically studied and evaluated yet.

6.3.3 Test case minimization

Another important part of triage is *test case minimization*. Test case minimization is the process of identifying the portion of a violating test case that is necessary to trigger the violation, and optionally producing a test case that is smaller and simpler than the original but still causes a violation. Although test case minimization and seed trimming (see 3.3, p. 8) are conceptually similar in that they both aim to reduce the size of an input, they are distinct because a minimizer can leverage a bug oracle. In fact, many existing minimizers are inspired by delta debugging [246].

Some fuzzers use their own implementation and algorithms for minimization. BFF [52] includes a minimization

algorithm tailored to fuzzing [110] that attempts to minimize the number of bits that are different from the original seed file. AFL [241] also includes a test case minimizer which attempts to simplify the test case by opportunistically setting bytes to zero and shortening the length of the test case. Lithium [186] is a general purpose test case minimization tool that minimizes files by attempting to remove "chunks" of adjacent lines or bytes in exponentially-descending sizes. Lithium was motivated by the complicated test cases produced by JavaScript fuzzers such as jsfunfuzz [194].

There are also a variety of test case reducers that are not specifically designed for fuzzing, but can nevertheless be used on test cases identified by fuzzing. These include format-agnostic techniques such as delta debugging [246] and specialized techniques for specific formats such as C-Reduce [185] for C/C++ files. Although specialized techniques are obviously limited in the types of files they can reduce, they have the advantage that they can be significantly more efficient than generic techniques because they have an understanding of the grammar of the test cases they are trying to simplify.

7 CONFIGURATION UPDATING

The CONFUPDATE function plays a critical role in distinguishing the behavior of black-box fuzzers from grey- and white-box fuzzers. As discussed in Algorithm 1, the CONFUPDATE function can modify the set of configurations (ℂ) based on the configuration and execution information collected during the current fuzzing run. In its simplest form, CONFUPDATE returns the ℂ parameter unmodified. Black-box fuzzers do not perform any program introspection beyond evaluating the bug oracle O_{bug}, and so they typically leave ℂ unmodified because they do not have any information collected that would allow them to modify it. However, we are aware that some fuzzers add violating test cases to the set of seeds. For example, BFF [52] calls this feature "crash recycling".

In contrast, grey- and white-box fuzzers are distinguished by their more sophisticated implementations of the CONFUPDATE function, which allows them to incorporate new fuzz configurations or remove old ones that may have been superseded. CONFUPDATE enables information collected during one fuzzing iteration to be used by all future iterations. For example, white-box fuzzers typically create a new fuzz configuration for every new test case produced since they produce relatively few test cases compared to black- and grey-box fuzzers.

7.1 Evolutionary Seed Pool Update

An Evolutionary Algorithm (EA) is a heuristic-based approach that involves bio-inspired evolution mechanisms such as mutation, recombination, and selection. In the context of fuzzing, an EA maintains a *seed pool* of promising individuals (i.e., seeds) that evolves over the course of a fuzz campaign as new individuals are discovered. Although the concept of EAs is relatively simple, it forms the basis of many grey-box fuzzers [241], [9], [226]. The process of choosing the seeds to be mutated and the mutation process itself were detailed in §4.3 and §5 respectively.

Arguably, the most important step of an EA is to add a new configuration to the set of configurations ℂ, which

occurs during the CONFUPDATE step of fuzzing. Most EA-based fuzzers use node or branch coverage as a fitness function: if a new node or branch is discovered by a test case, it is added to the seed pool. As the number of reachable paths can be orders of magnitude larger than the number of seeds, the seed pool is intended to be a *diverse* sub-selection of all reachable paths in order to represent the current exploration of the PUT. Also note that seed pools of different sizes can have the same coverage (as mentioned in §3.2, p. 7).

A common strategy in EA fuzzers is to refine the fitness function so that it can detect more subtle and granular indicators of improvements. For example, AFL [241] refines its fitness function definition by recording the number of times a branch has been taken. STADS [38] presents a statistical framework inspired by ecology to estimate how many new configurations will be discovered if the fuzz campaign continues. Another common strategy is to measure the fraction of conditions that are met when complex branch conditions are evaluated. For example, LAF-INTEL [134] simply breaks multi-byte comparison into several branches, which allows it to detect when a new seed passes an intermediate byte comparison. LibFuzzer [9], Honggfuzz [213], go-fuzz [225], and Steelix [143] instrument all comparisons and add any test case that makes progress on a comparison to the seed pool. A similar idea was also released as a stand-alone instrumentation module for clang [123]. Additionally, Steelix [143] checks which input offsets influence comparison instructions. Angora [177] improves the fitness criteria of AFL by considering the calling context of each branch taken. DeepXplore [176] adapts fuzzing to neural network testing by using "neuron coverage" as its fitness function.

VUzzer [183] is an EA-based fuzzer whose fitness function relies on the results of a custom program analysis, which determines weights for each basic block. Specifically, VUzzer first uses a built-in program analysis to classify basic blocks as either normal or error-handling (EH). For a normal block, its weight is inversely proportional to the probability that a random walk on the CFG containing this block visits it according to transition probabilities defined by VUzzer. This favors configurations exercising normal blocks deemed rare by the aforementioned random walk. The weight of EH blocks is *negative*, which discourages the execution of error handling (EH) blocks. This assumes that traversing an EH block signals a lower chance of exercising a vulnerability since bugs often coincide with unhandled errors.

7.2 Maintaining a Minset

With the ability to create new fuzz configurations comes the risk of creating too many configurations. A common strategy used to mitigate this risk is to maintain a *minset*, or a minimal set of test cases that maximizes a coverage metric. Minsetting is also used during PREPROCESS, and is described in more detail in §3.2. Some fuzzers use a variant of maintaining a minset that is specialized for configuration updates. As one example, rather than completely removing configurations that are not in the minset, as Cyberdyne [95] does, AFL [241] uses a *culling* procedure to mark minset configurations as being *favorable*. Favorable configurations are given a significantly higher chance of being selected by the SCHEDULE function. The author of AFL notes that "this

provides a reasonable balance between queue cycling speed and test case diversity" [245].

8 RELATED WORK

The literature on fuzzing had an early bloom in 2007–2008, when three trade-books on the subject were published within the two-year period [82], [212], [214]. These books took a more practical approach by presenting the different tools and techniques available at the time and their usages on a variety of targets. We note that Takanen et al. [214] already distinguished among black-, white- and grey-box fuzzers, although no formal definitions were given. Most recently, [214] had been revised after a decade. The second edition [215] contained many updates to include modern tools such as AFL [241] and ClusterFuzz [64].

We are aware of two fuzzing surveys that are concurrent to ours [142], [144]. However, whereas our goal is to provide a comprehensive study on recent developments covering the entire area, the goals of these surveys are more focused. In particular, while Li et al. [142] provided a thorough review of many recent advances in fuzzing, the authors have also decided to focus on the detail of coverage-based fuzzing and not others. More similar to ours, Liang et al. [144] proposed an informal model to describe various fuzzing techniques. However, their model does not seem to be flexible enough to encompass some of the fuzzing approaches we cover in this paper, such as model inference (see §5.1.2) and hybrid fuzzing (see §5.3).

Klees et al. [131] recently found that there has been no coherent way of evaluating fuzzing techniques, which can hamper our ability to compare the effectiveness of fuzzing techniques. In addition, they provided several useful guidelines for evaluating fuzzing algorithms. We consider their work to be orthogonal to ours as the evaluation of fuzzing algorithms is beyond the scope of this paper.

9 CONCLUDING REMARKS

As we have set forth in §1, our goal for this paper is to distill a comprehensive and coherent view of the modern fuzzing literature. To this end, we first presented a general-purpose model fuzzer to facilitate our effort to explain the many forms of fuzzing in current use. Then, we illustrated a rich taxonomy of fuzzers using Figure 1 (p. 5) and Table 1 (p. 6). We explored every stage of our model fuzzer by discussing the design decisions involved while showcasing the many achievements by the community at large. It is our hope that our work can help bring some more uniformity to future works, particularly in the terminology and in the presentation of fuzzing algorithms.

ACKNOWLEDGMENT

REFERENCES

[1] "Binspector: Evolving a security tool," https://blogs.adobe.com/security/2015/05/binspector-evolving-a-security-tool.html.

[2] "Cisco secure development lifecycle," https://www.cisco.com/c/en/us/about/security-center/security-programs/secure-development-lifecycle/sdl-process/validate.html.

[3] "Cwe-758: Reliance on undefined, unspecified, or implementation-defined behavior," https://cwe.mitre.org/data/definitions/758.html.

[4] "dharma," https://github.com/MozillaSecurity/dharma.

[5] "Fidgety afl," https://groups.google.com/forum/#!topic/afl-users/fOPeb62FZUg.

[6] "The fuzzing project," https://fuzzing-project.org/software.html.

[7] "Google chromium security," https://www.chromium.org/Home/chromium-security/bugs.

[8] "GPF," http://www.vdalabs.com/tools/efs_gpf.html.

[9] "LibFuzzer," http://llvm.org/docs/LibFuzzer.html.

[10] "Microsoft Security Development Lifecycle, verification phase," https://www.microsoft.com/en-us/sdl/process/verification.aspx.

[11] "Reddit: Iama mayhem, the hacking machine that won darpa's cyber grand challenge. ama!" https://www.reddit.com/r/IAmA/comments/4x9yn3/iama_mayhem_the_hacking_machine_that_won_darpas/.

[12] "Structure-aware fuzzing with libFuzzer," https://github.com/google/fuzzer-test-suite/blob/master/tutorial/structure-aware-fuzzing.md, 2019.

[13] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2005, pp. 340–353.

[14] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and Systems Security*, vol. 13, no. 1, pp. 4:1–4:40, 2009.

[15] H. J. Abdelnur, R. State, and O. Festor, "KiF: A stateful sip fuzzer," in *Proceedings of the International Conference on Principles*, 2007, pp. 47–56.

[16] D. Aitel, "An introduction to SPIKE, the fuzzer creation kit," in *Proceedings of the Black Hat USA*, 2001.

[17] D. Aitel, "Sharefuzz," https://sourceforge.net/projects/sharefuzz/, 2001.

[18] M. Aizatsky, K. Serebryany, O. Chang, A. Arya, and M. Whittaker, "Announcing OSS-Fuzz: Continuous fuzzing for open source software," Google Testing Blog, 2016.

[19] P. Amini, A. Portnoy, and R. Sears, "sulley," https://github.com/OpenRCE/sulley.

[20] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. Mcminn, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[21] D. Appelt, C. D. Nguyen, L. C. Briand, and N. Alshahwan, "Automated testing for sql injection vulnerabilities: An input mutation approach," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2014, pp. 259–269.

[22] Apple Inc., "Accessing crashwrangler to analyze crashes for security implications," Technical Note TN2334.

[23] A. Arcuri, M. Z. Iqbal, and L. Briand, "Random testing: Theoretical results and practical implications," *IEEE Transactions on Software Engineering*, vol. 38, no. 2, pp. 258–277, 2012.

[24] Ars Technica, "Pwn2own: The perfect antidote to fanboys who say their platform is safe," http://arstechnica.com/security/2014/03/pwn2own-the-perfect-antidote-to-fanboys-who-say-their-platform-is-safe/, 2014.

[25] C. Aschermann, P. Jauernig, T. Frassetto, A.-R. Sadeghi, T. Holz, and D. Teuchert, "NAUTILUS: Fishing for deep bugs with grammars," in *Proceedings of the Network and Distributed System Security Symposium*, 2019.

[26] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: Fuzzing with input-to-state correspondence," in *Proceedings of the Network and Distributed System Security Symposium*, 2019.

[27] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: Analyzing the android permission specification," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2012, pp. 217–228.

[28] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with Veritesting," in *Proceedings of the International Conference on Software Engineering*, 2014, pp. 1083–1094.

[29] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[30] D. Babic, L. Martignoni, S. McCamant, and D. Song, "Statically-directed dynamic automated test generation," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2011, pp. 12–22.

[31] G. Bai, J. Lei, G. Meng, S. S. Venkatraman, P. Saxena, J. Sun, Y. Liu, and J. S. Dong, "AUTHSCAN: Automatic extraction of web authentication protocols from implementations." in *Proceedings of the Network and Distributed System Security Symposium*, 2013.

[32] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna, "SNOOZE: Toward a stateful network protocol fuzzer," in *Proceedings of the International Conference on Information Security*, 2006, pp. 343–358.

[33] O. Bastani, R. Sharma, A. Aiken, and P. Liang, "Synthesizing program input grammars," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2017, pp. 95–110.

[34] I. Beer, "pwn4fun spring 2014–safari–part ii," http://googleprojectzero.blogspot.com/2014/11/pwn4fun-spring-2014-safari-part-ii.html, 2014.

[35] B. Beizer, *Black-box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, 1995.

[36] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the USENIX Annual Technical Conference*, 2005, pp. 41–46.

[37] D. A. Berry and B. Fristedt, *Bandit Problems: Sequential Allocation of Experiments*. Springer Netherlands, 1985.

[38] M. Böhme, "STADS: Software testing as species discovery," *ACM Transactions on Software Engineering and Methodology*, vol. 27, no. 2, pp. 7:1–7:52, 2018.

[39] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2017, pp. 2329–2344.

[40] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2016, pp. 1032–1043.

[41] E. Bounimova, P. Godefroid, and D. Molnar, "Billions and billions of constraints: Whitebox fuzz testing in production," in *Proceedings of the International Conference on Software Engineering*, 2013, pp. 122–131.

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

[43] S. Bratus, A. Hansen, and A. Shubina, "LZfuzz: a fast compression-based fuzzer for poorly documented protocols," Dartmouth College, Tech. Rep. TR2008-634, 2008.

[44] C. Brubaker, S. Janapa, B. Ray, S. Khurshid, and V. Shmatikov, "Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2014, pp. 114–129.

[45] D. L. Bruening, "Efficient, transparent, and comprehensive runtime code manipulation," Ph.D. dissertation, Massachusetts Institute of Technology, 2004.

[46] A. Budi, D. Lo, L. Jiang, and Lucia, "kb-Anonymity: A model for anonymized behavior-preserving test and debugging data," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2011, pp. 447–457.

[47] J. Caballero, P. Poosankam, S. McCamant, D. Babić, and D. Song, "Input generation via decomposition and re-stitching: Finding bugs in malware," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2010, pp. 413–425.

[48] J. Caballero, H. Yin, Z. Liang, and D. Song, "Polyglot: Automatic extraction of protocol message format using dynamic binary

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2946563, IEEE Transactions on Software Engineering

17

analysis," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2007, pp. 317–329.

[49] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the USENIX Symposium on Operating System Design and Implementation*, 2008, pp. 209–224.

[50] Y. Cai and W. Chan, "MagicFuzzer: Scalable deadlock detection for large-scale applications," in *Proceedings of the International Conference on Software Engineering*, 2012, pp. 606–616.

[51] D. Caselden, A. Bazhanyuk, M. Payer, L. Szekeres, S. McCamant, and D. Song, "Transformation-aware exploit generation using a HI-CFG," University of California, Tech. Rep. UCB/EECS-2013-85, 2013.

[52] CERT, "Basic Fuzzing Framework," https://www.cert.org/vulnerability-analysis/tools/bff.cfm.

[53] CERT, "Failure Observation Engine," https://www.cert.org/vulnerability-analysis/tools/foe.cfm.

[54] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2012, pp. 380–394.

[55] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2015, pp. 725–741.

[56] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a desired directed grey-box fuzzer," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2018, pp. 2095–2108.

[57] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "IoTFuzzer: Discovering memory corruptions in IoT through app-based fuzzing," in *Proceedings of the Network and Distributed System Security Symposium*, 2018.

[58] K. Chen and D. Wagner, "Large-scale analysis of format string vulnerabilities in debian linux," in *Proceedings of the Workshop on Programming Languages and Analysis for Security*, 2007, pp. 75–84.

[59] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2018, pp. 855–869.

[60] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. H. Tse, "Adaptive random testing: The ART of test case diversity," *Journal of Systems and Software*, vol. 83, no. 1, pp. 60–66, 2010.

[61] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, "Taming compiler fuzzers," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2013, pp. 197–208.

[62] Y. Chen, C. Su, C. Sun, S. Su, and J. Zhao, "Coverage-directed differential testing of jvm implementations," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2016, pp. 85–99.

[63] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011, pp. 265–278.

[64] Chrome Security Team, "Clusterfuzz," https://code.google.com/p/clusterfuzz/.

[65] CIFASIS, "Neural fuzzer," http://neural-fuzzer.org.

[66] P. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, "Prospex: Protocol specification extraction," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2009, pp. 110–125.

[67] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "DIFUZE: Interface aware fuzzing for kernel drivers," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2017, pp. 2123–2138.

[68] W. Cui, M. Peinado, S. K. Cha, Y. Fratantonio, and V. P. Kemerlis, "RETracer: Triaging crashes by reverse execution from partial memory dumps," in *Proceedings of the International Conference on Software Engineering*, 2016, pp. 820–831.

[69] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz, "Tupni: Automatic reverse engineering of input formats," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2008, pp. 391–402.

[70] L. Della Toffola, C. A. Staicu, and M. Pradel, "Saying 'hi!' is not enough: Mining inputs for effective test generation," in *Proceedings of the International Conference on Automated Software Engineering*, 2017, pp. 44–49.

[71] J. D. DeMott, R. J. Enbody, and W. F. Punch, "Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing," in *Proceedings of the Black Hat USA*, 2007.

[72] K. Dewey, J. Roesch, and B. Hardekopf, "Language fuzzing using constraint logic programming," in *Proceedings of the International Conference on Automated Software Engineering*, 2014, pp. 725–730.

[73] K. Dewey, J. Roesch, and B. Hardekopf, "Fuzzing the rust typechecker using clp," in *Proceedings of the International Conference on Automated Software Engineering*, 2015, pp. 482–493.

[74] W. Dietz, P. Li, J. Regehr, and V. Adve, "Understanding integer overflow in C/C++," in *Proceedings of the International Conference on Software Engineering*, 2012, pp. 760–770.

[75] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin, "Robust signatures for kernel data structures," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2009, pp. 566–577.

[76] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the State: A state-aware black-box web vulnerability scanner," in *Proceedings of the USENIX Security Symposium*, 2012, pp. 523–538.

[77] F. Duchene, S. Rawat, J.-L. Richier, and R. Groz, "Kameleonfuzz: Evolutionary fuzzing for black-box XSS detection," in *Proceedings of the ACM Conference on Data and Application Security and Privacy*, 2014, pp. 37–48.

[78] D. Duran, D. Weston, and M. Miller, "Targeted taint driven fuzzing using software metrics," in *Proceedings of the CanSecWest*, 2011.

[79] M. Eddington, "Peach fuzzing platform," http://community.peachfuzzer.com/WhatIsPeach.html.

[80] A. Edwards, A. Srivastava, and H. Vo, "Vulcan: Binary transformation in a distributed environment," Microsoft Research, Tech. Rep. MSR-TR-2001-50, 2001.

[81] S. Embleton, S. Sparks, and R. Cunningham, ""sidewinder": An evolutionary guidance system for malicious input crafting," in *Proceedings of the Black Hat USA*, 2006.

[82] G. Evron, N. Rathaus, R. Fly, A. Jenik, D. Maynor, C. Miller, and Y. Naveh, *Open Source Fuzzing Tools*. Syngress, 2007.

[83] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2011, pp. 627–638.

[84] S. Fewer, "A collection of burpsuite intruder payloads, fuzz lists and file uploads," https://github.com/1N3/IntruderPayloads.

[85] J. Foote, "Gdb exploitable," https://github.com/jfoote/exploitable.

[86] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "CollAFL: Path sensitive fuzzing," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2018, pp. 660–677.

[87] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *Proceedings of the International Conference on Software Engineering*, 2009, pp. 474–484.

[88] H. Gascon, C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck, "PULSAR: Stateful black-box fuzzing of proprietary network protocols," in *Proceedings of the International Conference on Security and Privacy in Communication Systems*, 2015, pp. 330–347.

[89] GitHub, "Public fuzzers," https://github.com/search?q=fuzzing&type=Repositories.

[90] P. Godefroid, "Random testing for security: Blackbox vs. whitebox fuzzing," in *Proceedings of the International Workshop on Random Testing*, 2007, pp. 1–1.

[91] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2008, pp. 206–215.

[92] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2005, pp. 213–223.

[93] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Proceedings of the Network and Distributed System Security Symposium*, 2008, pp. 151–166.

[94] P. Godefroid, H. Peleg, and R. Singh, "Learn&Fuzz: Machine learning for input fuzzing," in *Proceedings of the International Conference on Automated Software Engineering*, 2017, pp. 50–59.

[95] P. Goodman and A. Dinaburg, "The past, present, and future of cyberdyne," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2018, pp. 61–69.

[96] GrammaTech, "Grammatech blogs: The cyber grand challenge," http://blogs.grammatech.com/the-cyber-grand-challenge.

[97] G. Grieco, M. Ceresa, and P. Buiras, "Quickfuzz: An automatic random fuzzer for common file formats," in *Proceedings of the 9th International Symposium on Haskell*, 2016, pp. 13–20.

[98] J. Guo, Y. Jiang, Y. Zhao, Q. Chen, and J. Sun, "DLFuzz: Differential fuzzing testing of deep learning systems," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2018, pp. 739–743.

[99] A. H. H, "Melkor_elf_fuzzer," https://github.com/IOActive/Melkor_ELF_Fuzzer.

[100] I. Haller, Y. Jeon, H. Peng, M. Payer, C. Giuffrida, H. Bos, and E. Van Der Kouwe, "TypeSan: Practical type confusion detection," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2016, pp. 517–528.

[101] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," in *Proceedings of the USENIX Security Symposium*, 2013, pp. 49–64.

[102] D. Hamlet, "When only random testing will do," in *Proceedings of the International Workshop on Random Testing*, 2006, pp. 1–9.

[103] H. Han and S. K. Cha, "IMF: Inferred model-based fuzzer," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2017, pp. 2345–2358.

[104] H. Han, D. Oh, and S. K. Cha, "CodeAlchemist: Semantics-aware code generation to find vulnerabilities in javascript engines," in *Proceedings of the Network and Distributed System Security Symposium*, 2019.

[105] W. Han, B. Joe, B. Lee, C. Song, and I. Shin, "Enhancing memory error detection for large-scale applications and fuzz testing," in *Proceedings of the Network and Distributed System Security Symposium*, 2018.

[106] A. Helin, "radamsa," https://github.com/aoh/radamsa.

[107] S. Hocevar, "zzuf," https://github.com/samhocevar/zzuf.

[108] G. Hoglund, "Runtime decompilation," in *Proceedings of the Black Hat USA*, 2003.

[109] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Proceedings of the USENIX Security Symposium*, 2012, pp. 445–458.

[110] A. D. Householder, "Well there's your problem: Isolating the crash-inducing bits in a fuzzed file," CERT, Tech. Rep. CMU/SEI-2012-TN-018, 2012.

[111] A. D. Householder and J. M. Foote, "Probability-based parameter selection for black-box fuzz testing," CERT, Tech. Rep. CMU/SEI-2012-TN-019, 2012.

[112] W. E. Howden, "Methodology for the generation of program test data," *IEEE Transactions on Computers*, vol. C, no. 5, pp. 554–560, 1975.

[113] InfoSec Institute, "Charlie Miller reveals his process for security research," http://resources.infosecinstitute.com/how-charlie-miller-does-research/, 2011.

[114] V. Iozzo, "0-knowledge fuzzing," in *Proceedings of the Black Hat USA*, 2010.

[115] S. Jana and V. Shmatikov, "Abusing file processing in malware detectors for fun and profit," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2012, pp. 80–94.

[116] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun, "jFuzz: A concolic whitebox fuzzer for java," in *Proceedings of the First NASA Forma Methods Symposium*, 2009, pp. 121–125.

[117] Y. Jeon, P. Biswas, S. Carr, B. Lee, and M. Payer, "HexType: Efficient detection of type confusion errors for c++," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2017, pp. 2373–2387.

[118] W. Johansson, M. Svensson, U. E. Larson, M. Almgren, and V. Gulisano, "T-Fuzz: Model-based fuzzing for robustness testing of telecommunication protocols," in *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation*, 2014, pp. 323–332.

[119] D. Jones, "Trinity," https://github.com/kernelslacker/trinity.

[120] P. Joshi, C.-S. Park, K. Sen, and M. Naik, "A randomized dynamic program analysis technique for detecting real deadlocks," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2009, pp. 110–120.

[121] R. L. S. Jr., "A framework for file format fuzzing with genetic algorithms," Ph.D. dissertation, University of Tennessee, 2012.

[122] J. Jung, A. Sheth, B. Greenstein, D. Wetherall, G. Maganis, and T. Kohno, "Privacy oracle: A system for finding application leaks with black box differential testing," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2008, pp. 279–288.

[123] M. Jurczyk, "CompareCoverage," https://github.com/googleprojectzero/CompareCoverage.

[124] R. Kaksonen, M. Laakso, and A. Takanen, "Software security assessment through specification mutations and fault injection," in *Proceedings of the IFIP TC 6/TC 11 International Conference on Communications and Multimedia Security*, 2001, pp. 173–183.

[125] A. Kanade, R. Alur, S. Rajamani, and G. Ramanlingam, "Representation dependence testing using program inversion," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2010, pp. 277–286.

[126] A. Kapravelos, C. Grier, N. Chachra, C. Kruegel, G. Vigna, and V. Paxson, "Hulk: Eliciting malicious behavior in browser extensions," in *Proceedings of the USENIX Security Symposium*, 2014, pp. 641–654.

[127] U. Kargén and N. Shahmehri, "Turning programs against each other: High coverage fuzz-testing using binary-code mutation and dynamic slicing," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2015, pp. 782–792.

[128] H. Kario, "tlsfuzzer," https://github.com/tomato42/tlsfuzzer.

[129] S. Y. Kim, S. Lee, I. Yun, W. Xu, B. Lee, Y. Yun, and T. Kim, "CAB-Fuzz: Practical concolic testing techniques for COTS operating systems," in *Proceedings of the USENIX Annual Technical Conference*, 2017, pp. 689–701.

[130] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[131] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2018, pp. 2123–2138.

[132] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, and T. Marz, "Comparing operating systems using robustness benchmarks," in *Proceedings of the Symposium on Reliable Distributed Systems*, 1997, pp. 72–79.

[133] J. Koret, "Nightmare," https://github.com/joxeankoret/nightmare.

[134] lafintel, "Circumventing fuzzing roadblocks with compiler transformations," https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/, 2016.

[135] Z. Lai, S. Cheung, and W. Chan, "Detecting atomic-set serializability violations in multithreaded programs through active randomized testing," in *Proceedings of the International Conference on Software Engineering*, 2010, pp. 235–244.

[136] M. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely, "PEBIL: Efficient static binary instrumentation for linux," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems & Software*, 2010, pp. 175–183.

[137] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Proceedings of the International Conference on Software Engineering*, 2012, pp. 3–13.

[138] B. Lee, C. Song, T. Kim, and W. Lee, "Type casting verification: Stopping an emerging attack vector," in *Proceedings of the USENIX Security Symposium*, 2015, pp. 81–96.

[139] S. Lee, C. Yoon, C. Lee, S. Shin, V. Yegneswaran, and P. Porras, "DELTA: A security assessment framework for software-defined networks," in *Proceedings of the Network and Distributed System Security Symposium*, 2017.

[140] C. Lemieux, R. Padhye, K. Sen, and D. Song, "PerfFuzz: Automatically generating pathological inputs," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2018, pp. 254–265.

[141] C. Lemieux and K. Sen, "FairFuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the International Conference on Automated Software Engineering*, 2018, pp. 475–485.

[142] J. Li, B. Zhao, and C. Zhang, "Fuzzing: a survey," *Cybersecurity*, vol. 1, no. 1, 2018.

[143] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: Program-state based binary fuzzing," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2017, pp. 627–637.

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

[145] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, "Many-core compiler fuzzing," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2015, pp. 65–76.

[146] Z. Lin and X. Zhang, "Deriving input syntactic structure from execution," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2008, pp. 83–93.

[147] P. Liu, X. Zhang, M. Pistoia, Y. Zheng, M. Marques, and L. Zeng, "Automatic text input generation for mobile testing," in *Proceedings of the International Conference on Software Engineering*, 2017, pp. 643–653.

[148] LMH, S. Grubb, I. van Sprundel, E. Sandeen, and J. Wilson, "fsfuzzer," http://people.redhat.com/sgrubb/files/fsfuzzer-0.7.tar.gz.

[149] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2005, pp. 190–200.

[150] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner, and R. Ramler, "GRT: Program-analysis-guided random testing," in *Proceedings of the International Conference on Automated Software Engineering*, 2015, pp. 212–223.

[151] R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, and A. Stavrou, "A whitebox approach for automated security testing of android applications on the cloud," in *Proceedings of the International Workshop on Automation of Software Test*, 2012, pp. 22–28.

[152] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis, "Path-exploration lifting: Hi-fi tests for lo-fi emulators," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012, pp. 337–348.

[153] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[154] D. Mckinney, "antiparser," http://antiparser.sourceforge.net/.

[155] Microsoft Corporation, "!exploitable crash analyzer – MSEC debugger extensions," https://msecdbg.codeplex.com.

[156] Microsoft Corporation, "Minifuzz," https://msdn.microsoft.com/en-us/biztalk/gg675011.

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

[158] C. Miller, "Fuzz by number: More data about fuzzing than you ever wanted to know," in *Proceedings of the CanSecWest*, 2008.

[159] D. Molnar, X. C. Li, and D. A. Wagner, "Dynamic test generation to find integer bugs in x86 binary linux programs," in *Proceedings of the USENIX Security Symposium*, 2009, pp. 67–82.

[160] L. D. Moura and N. Bjørner, "Satisfiability modulo theories: Introduction and applications," *Communications of the ACM*, vol. 54, no. 9, pp. 69–77, 2011.

[161] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What you corrupt is not what you crash: Challenges in fuzzing embedded devices," in *Proceedings of the Network and Distributed System Security Symposium*, 2018.

[162] C. Mulliner, N. Golde, and J.-P. Seifert, "SMS of death: from analyzing to attacking mobile phones on a large scale," in *Proceedings of the USENIX Security Symposium*, 2011, pp. 24–24.

[163] MWR Labs, "KernelFuzzer," https://github.com/mwrlabs/KernelFuzzer.

[164] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*. Wiley, 2011.

[165] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "SoftBound: Highly compatible and complete spatial memory safety for C," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2009, pp. 245–258.

[166] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "CETS: Compiler enforced temporal safety for C," in *Proceedings of the International Symposium on Memory Management*, 2010, pp. 31–40.

[167] NCC Group, "Hodor fuzzer," https://github.com/nccgroup/hodor.

[168] NCC Group, "Triforce linux syscall fuzzer," https://github.com/nccgroup/TriforceLinuxSyscallFuzzer.

[169] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2007, pp. 89–100.

[170] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proceedings of the Network and Distributed System Security Symposium*, 2005.

[171] D. Oleksiuk, "Ioctl fuzzer," https://github.com/Cr4sh/ioctlfuzzer, 2009.

[172] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proceedings of the International Conference on Software Engineering*, 2007, pp. 75–84.

[173] S. Pailoor, A. Aday, and S. Jana, "MoonShine: Optimizing OS fuzzer seed selection with trace distillation," in *Proceedings of the USENIX Security Symposium*, 2018, pp. 729–743.

[174] J. Pan, G. Yan, and X. Fan, "Digtool: A virtualization-based framework for detecting kernel vulnerabilities," in *Proceedings of the USENIX Security Symposium*, 2017, pp. 149–165.

[175] C.-S. Park and K. Sen, "Randomized active atomicity violation detection in concurrent programs," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2008, pp. 135–145.

[176] K. Pei, Y. Cao, J. Yang, and S. Jana, "DeepXplore: Automated whitebox testing of deep learning systems," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 1–18.

[177] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-Fuzz: Fuzzing by program transformation," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2018, pp. 917–930.

[178] T. Petsios, A. Tang, S. J. Stolfo, A. D. Keromytis, and S. Jana, "NEZHA: Efficient domain-independent differential testing," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2017, pp. 615–632.

[179] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Model-based whitebox fuzzing for program binaries," in *Proceedings of the International Conference on Automated Software Engineering*, 2016, pp. 543–553.

[180] P. Project, "DynInst: Putting the performance in high performance computing," http://www.dyninst.org/.

[181] H. Raffelt, B. Steffen, and T. Berg, "LearnLib: A library for automata learning and experimentation," in *Proceedings of the International Workshop on Formal Methods for Industrial Critical Systems*, 2005, pp. 62–71.

[182] S. Rasthofer, S. Arzt, S. Triller, and M. Pradel, "Making malory behave maliciously: Targeted fuzzing of android execution environments," in *Proceedings of the International Conference on Software Engineering*, 2017, pp. 300–311.

[183] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware evolutionary fuzzing," in *Proceedings of the Network and Distributed System Security Symposium*, 2017.

[184] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *Proceedings of the USENIX Security Symposium*, 2014, pp. 861–875.

[185] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, , and X. Yang, "Test-case reduction for C compiler bugs," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2012, pp. 335–346.

[186] J. Ruderman, "Lithium," https://github.com/MozillaSecurity/lithium/.

[187] J. D. Ruiter and E. Poll, "Protocol state fuzzing of tls implementations," in *Proceedings of the USENIX Security Symposium*, 2015, pp. 193–206.

[188] M. Samak, M. K. Ramanathan, and S. Jagannathan, "Synthesizing racy tests," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2015, pp. 175–185.

[189] P. Saxena, S. Hanna, P. Poosankam, and D. Song, "FLAX: Systematic discovery of client-side validation vulnerabilities in rich web applications," in *Proceedings of the Network and Distributed System Security Symposium*, 2010.

[190] F. B. Schneider, "Enforceable security policies," *ACM Transactions on Information System Security*, vol. 3, no. 1, pp. 30–50, 2000.

[191] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-assisted feedback fuzzing for os kernels," in *Proceedings of the USENIX Security Symposium*, 2017, pp. 167–182.

[192] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2010, pp. 317–331.

[193] M. Schwarz, D. Gruss, M. Lipp, C. Maurice, T. Schuster, A. Fogh, and S. Mangard, "Automated detection, exploitation, and elimination of double-fetch bugs using modern CPU features," in

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2946563, IEEE Transactions on Software Engineering

20

Proceedings of the ACM Symposium on Information, Computer and Communications Security, 2018, pp. 587–600.

[194] M. Security, "funfuzz," https://github.com/MozillaSecurity/funfuzz.

[195] M. Security, "orangfuzz," https://github.com/MozillaSecurity/orangfuzz.

[196] K. Sen, "Effective random testing of concurrent programs," in *Proceedings of the International Conference on Automated Software Engineering*, 2007, pp. 323–332.

[197] K. Sen, "Race directed random testing of concurrent programs," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2008, pp. 11–21.

[198] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2005, pp. 263–272.

[199] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Proceedings of the USENIX Annual Technical Conference*, 2012, pp. 309–318.

[200] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer: data race detection in practice," in *Proceedings of the Workshop on Binary Instrumentation and Applications*, 2009, pp. 62–71.

[201] S. Shen, S. Shinde, S. Ramesh, A. Roychoudhury, and P. Saxena, "Neuro-symbolic execution: Augmenting symbolic execution with neural constraints," in *Proceedings of the Network and Distributed System Security Symposium*, 2019.

[202] Z. Sialveras and N. Naziridis, "Introducing Choronzon: An approach at knowledge-based evolutionary fuzzing," in *Proceedings of the ZeroNights*, 2015.

[203] J. Somorovsky, "Systematic fuzzing and testing of tls libraries," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2016, pp. 1492–1504.

[204] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J.-P. Seifert, and M. Franz, "Periscope: An effective probing and fuzzing framework for the hardware-os boundary," in *Proceedings of the Network and Distributed System Security Symposium*, 2019.

[205] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, "SoK: Sanitizing for security," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2019.

[206] W. Song, X. Qian, and J. Huang, "EHBDroid: Beyond GUI testing for android applications," in *Proceedings of the International Conference on Automated Software Engineering*, 2017, pp. 27–37.

[207] C. Spensky and H. Hu, "Ll-fuzzer," https://github.com/mit-ll/LL-Fuzzer.

[208] E. Stepanov and K. Serebryany, "MemorySanitizer: fast detector of uninitialized memory use in C++," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2015, pp. 46–55.

[209] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *Proceedings of the Network and Distributed System Security Symposium*, 2016.

[210] M. Sutton, "Filefuzz," http://osdir.com/ml/security.securiteam/2005-09/msg00007.html.

[211] M. Sutton and A. Greene, "The art of file format fuzzing," in *Proceedings of the Black Hat Asia*, 2005.

[212] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

[213] R. Swiecki and F. Gröbert, "honggfuzz," https://github.com/google/honggfuzz.

[214] A. Takanen, J. D. DeMott, and C. Miller, *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, 2008.

[215] A. Takanen, J. D. DeMott, C. Miller, and A. Kettunen, *Fuzzing for Software Security Testing and Quality Assurance*, 2nd ed. Artech House, 2018.

[216] D. Thiel, "Exposing vulnerabilities in media software," in *Proceedings of the Black Hat EU*, 2008.

[217] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, U. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in gcc & llvm," in *Proceedings of the USENIX Security Symposium*, 2014, pp. 941–955.

[218] N. Tillmann and J. De Halleux, "Pex–white box test generation for .NET," in *Proceedings of the International Conference on Tests and Proofs*, 2008, pp. 134–153.

[219] D. Trabish, A. Mattavelli, N. Rinetzky, and C. Cadar, "Chopped symbolic execution," in *Proceedings of the International Conference on Software Engineering*, 2018, pp. 350–360.

[220] Trail of Bits, "GRR," https://github.com/trailofbits/grr.

[221] R. Valotta, "Taking browsers fuzzing to the next (dom) level," in *Proceedings of the DeepSec*, 2012.

[222] R. van Tonder, J. Kotheimer, and C. Le Goues, "Semantic crash bucketing," in *Proceedings of the International Conference on Automated Software Engineering*, 2018.

[223] S. Veggalam, S. Rawat, I. Haller, and H. Bos, "IFuzzer: An evolutionary interpreter fuzzer using genetic programming," in *Proceedings of the European Symposium on Research in Computer Security*, 2016, pp. 581–601.

[224] M. Vuagnoux, "Autodafé: an act of software torture," in *Proceedings of the Chaos Communication Congress*, 2005, pp. 47–58.

[225] D. Vyukov, "go-fuzz," https://github.com/dvyukov/go-fuzz.

[226] D. Vyukov, "syzkaller," https://github.com/google/syzkaller.

[227] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2017, pp. 579–594.

[228] S. Wang, J. Nam, and L. Tan, "QTEP: Quality-aware test case prioritization," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2017, pp. 523–534.

[229] T. Wang, T. Wei, G. Gu, and W. Zou, "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2010, pp. 497–512.

[230] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama, "Towards optimization-safe systems: Analyzing the impact of undefined behavior," in *Proceedings of the ACM Symposium on Operating System Principles*, 2013, pp. 260–275.

[231] V. M. Weaver and D. Jones, "perf_fuzzer: Targeted fuzzing of the perf_event_open() system call," UMaine VMW Group, Tech. Rep., 2015.

[232] J. Wei, J. Chen, Y. Feng, K. Ferles, and I. Dillig, "Singularity: Pattern fuzzing for worst case complexity," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2018, pp. 213–223.

[233] S. Winter, C. Sârbu, N. Suri, and B. Murphy, "The impact of fault models on software robustness evaluations," in *Proceedings of the International Conference on Software Engineering*, 2011, pp. 51–60.

[234] M. Y. Wong and D. Lie, "Intellidroid: A targeted input generator for the dynamic analysis of android malware," in *Proceedings of the Network and Distributed System Security Symposium*, 2016.

[235] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling black-box mutational fuzzing," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2013, pp. 511–522.

[236] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Fitness-guided path exploration in dynamic symbolic execution," in *Proceedings of the International Conference on Dependable Systems Networks*, 2009, pp. 359–368.

[237] D. Xu, J. Ming, and D. Wu, "Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2017, pp. 921–937.

[238] W. Xu, S. Kashyap, C. Min, and T. Kim, "Designing new operating primitives to improve fuzzing performance," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2017, pp. 2313–2328.

[239] D. Yang, Y. Zhang, and Q. Liu, "Blendfuzz: A model-based framework for fuzz testing programs with grammatical inputs," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2012, pp. 1070–1076.

[240] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A practical concolic execution engine tailored for hybrid fuzzing," in *Proceedings of the USENIX Security Symposium*, 2018, pp. 745–762.

[241] M. Zalewski, "American Fuzzy Lop," http://lcamtuf.coredump.cx/afl/.

[242] M. Zalewski, "Crossfuzz," https://lcamtuf.blogspot.com/2011/01/announcing-crossfuzz-potential-0-day-in.html.

[243] M. Zalewski, "New in AFL: persistent mode," https://lcamtuf.blogspot.com/2015/06/new-in-afl-persistent-mode.html.

[244] M. Zalewski, "ref_fuzz," http://lcamtuf.blogspot.com/2010/06/announcing-reffuzz-2yo-fuzzer.html.

[245] M. Zalewski, "Technical "whitepaper" for afl-fuzz," http://lcamtuf.coredump.cx/afl/technical_details.txt.

[246] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[247] K. Zetter, "A famed hacker is grading thousands of programs—and may revolutionize software in the process," https://goo.gl/LRwaVl.

[248] M. Zhang, R. Qiao, N. Hasabnis, and R. Sekar, "A platform for secure static binary instrumentation," in *Proceedings of the International Conference on Virtual Execution Environments*, 2014, pp. 129–140.

[249] L. Zhao, Y. Duan, H. Yin, and J. Xuan, "Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing," in *Proceedings of the Network and Distributed System Security Symposium*, 2019.

[250] M. Zimmermann, "Tavor," https://github.com/zimmski/tavor.