Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

A systematic review of fuzzing techniques

Chen Chen [a,*], *Baojiang Cui* [a], *Jinxin Ma* [b], *Runpu Wu* [b], *Jianchao Guo* [a], *Wenqian Liu* [a]

[a] *Beijing University of Posts and Telecommunications, No 10, Xitucheng Road, Haidian District, Beijing, China*
[b] *China Information Technology Security Evaluation Center, Building No.1, Courtyard No.8, Shangdixi Road, Haidian District, Beijing, China*

ARTICLE INFO

Article history:
Received 7 October 2017
Received in revised form 26 January 2018
Accepted 3 February 2018
Available online 13 February 2018

Keywords:
Software bug
Vulnerability
Fuzzing
Dynamic symbolic execution
Coverage guide
Grammar representation
Scheduling algorithms
Taint analysis
Static analysis

ABSTRACT

Fuzzing is an effective and widely used technique for finding security bugs and vulnerabilities in software. It inputs irregular test data into a target program to try to trigger a vulnerable condition in the program execution. Since the first random fuzzing system was constructed, fuzzing efficiency has been greatly improved by combination with several useful techniques, including dynamic symbolic execution, coverage guide, grammar representation, scheduling algorithms, dynamic taint analysis, static analysis and machine learning. In this paper, we will systematically review these techniques and their corresponding representative fuzzing systems. By introducing the principles, advantages and disadvantages of these techniques, we hope to provide researchers with a systematic and deeper understanding of fuzzing techniques and provide some references for this field.

1. Introduction

A software bug is an error, flaw, failure or fault in a computer program or system that causes it to produce an incorrect or unexpected result or to behave in unintended ways (Wikipedia, 2017a). Bugs usually stem from a program's source code or design. To eliminate such errors, traditional test methods use handwritten test cases, such as unit testing, mainly to validate the functionality of the software components. However, this approach is insufficient to uncover critical bugs.

Critical bugs, called software vulnerabilities, are significantly more dangerous than others and may affect the security of the entire system. Vulnerabilities allow attackers to corrupt important data structures and execute arbitrary code with the privileges of the program, even completely controlling the system on which the program is running. Vulnerability is the intersection of three elements: a system susceptibility or flaw, access to the flaw by an attacker, and the attacker's capability to exploit the flaw (Wikipedia, 2017b).

Fuzzing is a highly effective vulnerability detection technique. It tests a system with the continuous processing of test cases generated by another program. At the same time, the system is monitored to expose any defects revealed by processing this input. The first fuzzing tool was developed by Miller et al. (1990) and was originally designed to test the reliability

* *Corresponding author.*
E-mail address: 00152tenten@bupt.edu.cn (C. Chen).
https://doi.org/10.1016/j.cose.2018.02.002

of UNIX tools. Since then, increasing numbers of systems have improved the efficiency of fuzzing by introducing new techniques, and many vulnerabilities have been detected in software programs. In this article, we will focus on fuzzing techniques and fuzzing systems for expanded discussion.

The rest of this paper is structured as follows. Section 2 introduces fuzzing systems from different perspectives. Section 3 analyses different techniques used in fuzzing systems. Section 4 presents representative fuzzing systems by time period. Section 5 analyses and compares multiple representative fuzzing systems. Section 6 discusses the future directions of fuzzing. Section 7 presents the conclusions drawn.

2. Anatomy of a fuzzing system

Nearly 30 years have passed since the first fuzzing system was constructed. Several new techniques have been introduced into fuzzing systems, which are becoming increasingly complicated. In this section, we will analyse fuzzing systems and fuzzing techniques from different angles, including the structure of a fuzzing system, the classifications of fuzzing systems and fuzzing systems used in different fields.

2.1. *The structure of a fuzzing system*

Due to the inefficiency of early fuzzing systems, fuzzing techniques have been improved and combined with other techniques over the years. The structure of fuzzing systems has also changed and become more complicated.

2.1.1. *The original fuzzing system*

The structure of the early fuzzing system is simple, as shown in the blue dashed frame in Fig. 1. The words linked to the function module are the techniques implemented in this module. The relation between modules connected by red arrows is control, and the relation between modules connected by black arrows is data transfer. Five modules are contained within the system: the target program, test case generator, delivery module, bug detector and bug filter.

The target program is the program being tested. It can be a binary code with or without source code, a program for consuming files or a network service program, an application program, an operating system, a compiler, a browser, a library, etc.

The test case generator normally mutates a sample and creates an input for the application to be tested. The generated input can be a particular type of file or a network data stream. The generator can use different mutation strategies (Lcamtuf, 2014) for sample generation to improve the efficiency of fuzzing.

The delivery module accepts samples from the test case generator and sends them to the target program for consumption. The delivery module is closely related to the input nature of the target application; for example, the delivery module of a system that receives files as input is different from that of a system that receives data streams from the network.

When a target program crashes or reports errors in fuzzing, the relevant information must be collected and analysed to determine whether a bug is detected. Some tools (Hastings and Joyce, 1991; Nethercote and Seward, 2007; Serebryany et al., 2012; Slowinska et al., 2012) facilitate bug detection and can also be used to record the relevant exception information.

Security testers must make clear whether the resulting error can be exploited. Filtering vulnerabilities is usually done manually and is therefore usually time consuming and difficult to solve. Some new tools (Chen et al., 2013; Francis et al., 2004; Team MSECMSS, 2013; Zalewski, 2016) have eased the problem; e.g., !Exploitable (Team MSECMSS, 2013) built on top of GDB (Stallman et al., 2002) can assess the exploitability of a bug.

2.1.2. *The extended fuzzing system*

Due to the inefficiency of the early fuzzing systems, researchers have introduced a variety of different techniques to improve

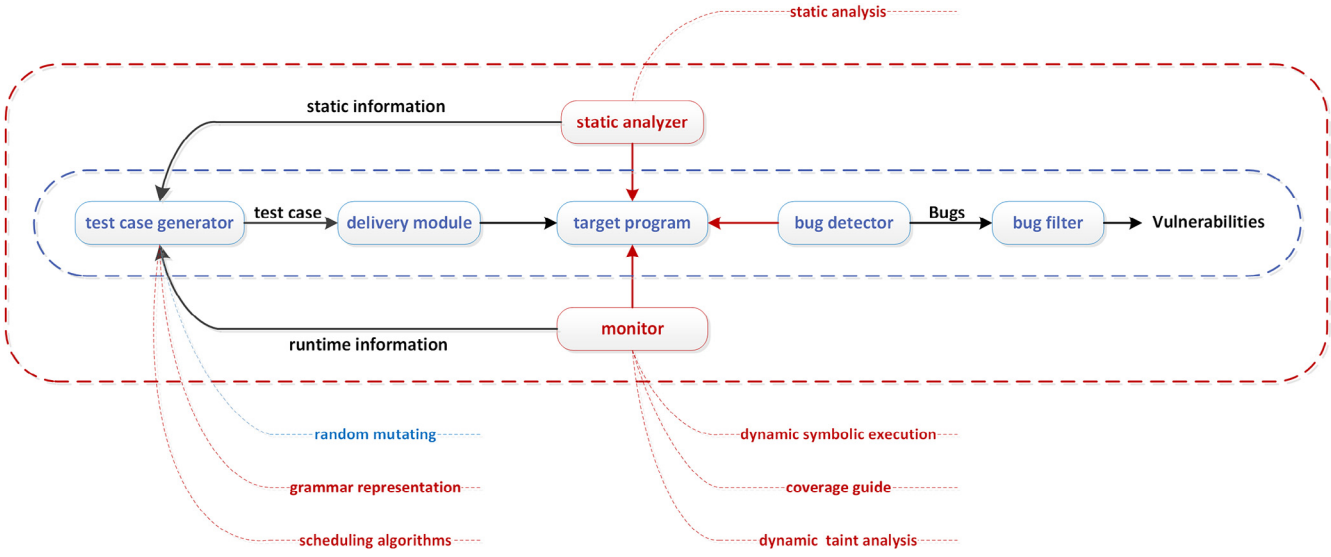

Fig. 1 – An architectural diagram of fuzzing system. (For interpretation of the references to color in this figure, the reader is referred to the web version of this article.)

the efficiency. These techniques include dynamic symbolic execution (Cadar et al., 2008; Cha et al., 2012; Chipounov et al., 2011; Godefroid et al., 2008), coverage feedback (Böhme et al., 2016, 2017; Zalewski, 2016; Google, 2017a, 2017b), grammar representation (Banks et al., 2006; Röning et al., 2002; Peachtec, 2017; Bradshaw, 2010; Fitblip 2016), taint analysis (Ganesh et al., 2009; Haller et al., 2013; Neugschwandtner et al., 2015; Wang et al., 2010), static analysis (Kinder et al., 2009; Sparks et al., 2007; Tip, 1994; Weiser, 1979, 1981, 1982), scheduling algorithms (Böhme et al., 2016, 2017; Householder and Foote, 2012; Woo et al., 2013) and machine learning (Godefroid et al., 2017). These techniques extend the structure of the fuzzing system. The current structure of fuzzing systems includes the content shown in red dashed lines in Fig. 1.

The monitor is used to obtain runtime information, which is sent to the test case generator. The runtime information includes symbolic expressions (Cadar et al., 2008; Godefroid et al., 2005, 2008; Haller et al., 2013; Neugschwandtner et al., 2015; Sen et al., 2005; Stephens et al., 2016), path coverage data (Zalewski, 2016; Google 2017a, 2017b) and taint information (Ganesh et al., 2009; Rawat et al., 2017; Wang et al., 2010). Symbolic expressions are used to generate input data. Path coverage data are used for seed selection in the next loop. Taint data are used to infer which offsets in the input influence the execution path of the program.

The static analyser extracts static information (Babić et al., 2011; Grieco et al., 2016; Haller et al., 2013; Kinder et al., 2009; Neugschwandtner et al., 2015; Sparks et al., 2007) from a binary program or source code to direct the fuzzing process (Babić et al., 2011; Grieco et al., 2016; Haller et al., 2013; Neugschwandtner et al., 2015). The static information, which includes control flow graphs (Kinder et al., 2009; Sparks et al., 2007), potentially vulnerable code (Babić et al., 2011; Haller et al., 2013; Neugschwandtner et al., 2015) and specified patterns (Grieco et al., 2016), can be used to guide the fuzzing.

In addition, grammar representation (Banks et al., 2006; Röning et al., 2002; Peachtec 2017; Bradshaw, 2010; Fitblip, 2016) and scheduling algorithms (Böhme et al., 2016, 2017; Householder and Foote, 2012; Woo et al., 2013) are added to the test case generator to enhance the fuzzing efficiency. Grammar representation is used to generate fine samples by obtaining input format information. Scheduling algorithms are used to improve the efficiency of fuzzing by using the seed choosing strategy and seed mutation strategy. All of the techniques used in the different modules are described in detail in Section 3.

2.2. *The classification of fuzzing*

Fuzzing techniques can be classified from multiple perspectives. They can be divided into black-box, white-box and grey-box fuzzing by the understanding of the target program, into mutation-based and generation-based fuzzing by the type of data generation, and into feedback and no-feedback fuzzing by the feedback type.

2.2.1. *White-box, black-box and grey-box*

Black-box fuzzing (Miller et al., 1990; Peachtec, 2017; Fitblip, 2016; Helin, 2017) does not consider the internal logic of the program but continuously provides input data and observes the output results. Black-box fuzzing lies at one extreme in terms of the level of program understanding.

At the other extreme is white-box fuzzing (Cadar et al., 2008; Godefroid et al., 2005, 2008; Haller et al., 2013), which can obtain detailed information on the program, including source code, design specifications, and detailed runtime information. This information is then utilized to improve the efficiency of the fuzzing process.

Between these two extremes lies grey-box fuzzing (Böhme et al., 2016; Zalewski, 2016), which obtains some information but not details, e.g., runtime coverage information.

2.2.2. *Mutation-based and generation-based*

Fuzzing can be divided into mutation-based (Ganesh et al., 2009; Householder and Foote, 2012; Miller et al., 1990; Peachtec, 2017) and generation-based (Röning et al., 2002; Peachtec, 2017; Spike fuzzer platform) categories according to the input generation strategy. The mutation-based approach performs random transformations on a prepared input seed, while the generation-based approach generates inputs using a formal input format specification, e.g., grammar (Röning et al., 2002), block (Bradshaw, 2010) or model (Peachtec, 2017).

Mutation-based fuzzing may cause the inputs to be rejected early in processing because the mutated input data deviates too much from the format expected by the target program. However, this deficiency is relieved by the input data specification in generation-based approaches. Therefore, mutation-based fuzzing almost invariably generates lower code coverage than generation-based fuzzing, but creating the input data specification is often very time consuming and cannot include all possible kinds of input formats.

2.2.3. *Feedback and no-feedback*

Feedback refers to whether the runtime information can serve to guide the generation of the test case in the next loop. Feedback techniques are mainly based on path coverage. No-feedback fuzzing does not obtain any information from the program execution.

Coverage-guided fuzzers (Zalewski, 2016; Google, 2017a, 2017b) obtain the path coverage information generated by the instrumentation tools (Bhansali et al., 2006; Luk et al., 2005; Nethercote and Seward, 2007; Bruening, 2017; Paradyn, 2017). This information is then used to guide the generation of the test case in the next loop to maximize the path coverage. Examples include AFL (Zalewski, 2016), honggfuzz (Google, 2017a), and Syzkaller (Google, 2017b).

2.3. *Fuzzing systems used in different targets*

Fuzzing can be used to detect vulnerabilities in desktop PCs and embedded devices (Wikipedia, 2017c). Researchers can effectively use different currently developed fuzzing techniques on desktop PCs (Godefroid et al., 2008; Stephens et al., 2016; Zalewski, 2016; Peachtec, 2017) but cannot use them well on embedded devices due to their limited resources. First, many fuzzing tools cannot work on the embedded devices because of the limited hard disk space, memory space and CPU speed, etc. Second, many fuzzing tools rely on rich software

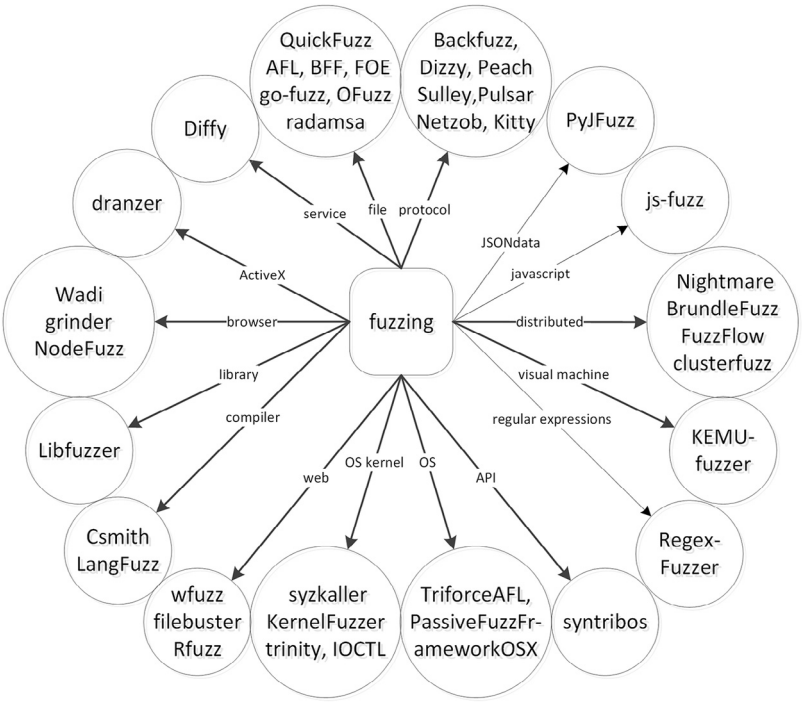

Fig. 2 – Fuzzing Tools Used in Different Targets.

interfaces of operation systems, which are pruned in embedded devices. Third, the fuzzing process is based on the observable reply of the target errors that have occurred, but error occurrence mechanisms do not exist in many embedded devices (Muench et al., 2018). Therefore, fuzzing for desktop PCs and that for embedded devices are quite different. Some studies have performed fuzzing on embedded devices (Halperin et al., 2008; Muench et al., 2018; Rouf et al., 2010; Shoshitaishvili et al., 2015; Zaddach et al., 2014). In this paper, we focus on a discussion of fuzzing techniques and fuzzing systems for the desktop PC.

Fuzzing has been widely used in different targets on desktop PCs. We introduce some open-source fuzzing tools, which are shown in Fig. 2. The main difference between these tools is the adapter interface for the target. The fuzzing techniques used in these tools may be the same or different. For example, AFL (Zalewski, 2016), libfuzzer (LLVM-admin team, 2017), and syzkaller (Google, 2017b) leverage coverage-guided techniques to perform a fuzzing test on file-type applications, libraries and the OS kernel, respectively. For the same target, for example, the file-type application, AFL (Zalewski, 2016) leverages coverage-guided techniques, and BFF (Householder and Foote, 2012) adds scheduling algorithms to random fuzzing. In this section, we do not emphasize the use of techniques in these tools; some of these techniques are not the most efficient, but they can supply a fuzzing frame for different targets. More techniques can be added to these tools for higher efficiency if researchers want to perform fuzzing on a specific target.

The tools used in file application fuzzing include AFL (Zalewski, 2016), BFF (Householder and Foote, 2012), FOE (Householder, 2012), radamsa (Helin, 2017), QuickFuzz (CIFASIS, 2017), OFuzz (Cha, 2017), and go-fuzz (Smith, 2017); in protocol fuzzing, Dizzy (Daniel, 2017), Sulley (Fitblip, 2016), Backfuzz (Choren, 2017), Pulsar (Pulsar), Netzob (Bossert, 2017), Kitty (Bsharet, 2017), and Peach (Peachtec, 2017) are used; in operation system fuzzing, PassiveFuzzFrameworkOSX (Li, 2017) and TriforceAFL (Newsham, 201) are used; in OS kernel fuzzing, KernelFuzzer (MWR Labs, 2017), syzkaller (Google, 2017b), Trinity (Jones, 2017), and IOCTL (Google, 2017) are used; in library fuzzing, Libfuzzer (LLVM-admin team, 2017) is used; in API fuzzing, syntribos (Nair, 2017) is used; in web fuzzing, filebuster (Sintra, 2017), wfuzz (Mendez, 2017), and Rfuzz (Shaw, 2017) are used; in service fuzzing, Diffy (Khanduri, 2017) is used; in ActiveX fuzzing, dranzer (Householder, 2017) is used; in virtual machine fuzzing, KEMUfuzzer (Martignoni, 2017) is used; in browser fuzzing, Wadi (El-Sherei, 2017), NodeFuzz (Kettunen, 2017), and grinder (Fewer, 2017) are used; in JSON data fuzzing, PyJFuzz (Linguaglossa, 2017) is used; in JavaScript fuzzing, jsfuzz (Peet, 2017) is used; in regular expressions fuzzing, Regex Fuzzer (Trull, 2010) is used; and in compiler fuzzing, csmith (Regehr, 2017) and LangFuzz (Holler et al., 2012) are used. In addition, Nightmare (Koret, 2017), BrundleFuzz (Prado, 2017), FuzzFlow (Talos, 2017a), and clusterfuzz (Google, 2017c) can perform distributed fuzzing.

3. Fuzzing techniques

Many techniques have been introduced into fuzzing systems to improve the efficiency of random mutation fuzzing, including dynamic symbolic execution (Cadar et al., 2008; Cha et al., 2012; Chipounov et al., 2011; Godefroid et al., 2008), coverage feedback (Böhme et al., 2016, 2017; Zalewski, 2016; Google 2017a, 2017b), grammar representation (Banks et al., 2006; Röning et al., 2002; Peachtec, 2017; Bradshaw, 2010; Fitblip, 2016), taint analysis (Ganesh et al., 2009; Haller et al., 2013;

Neugschwandtner et al., 2015; Wang et al., 2010), static analysis (Kinder et al., 2009; Sparks et al., 2007; Tip, 1994; Weiser, 1979, 1981, 1982) and scheduling algorithms (Böhme et al., 2016, 2017; Householder and Foote, 2012; Woo et al., 2013). Each technique offers different trade-offs between soundness, completeness, speed, precision, scalability and the level of automation. According to the role the fuzzing techniques play, these techniques can be divided into three categories: sample generation techniques, dynamic analysis techniques and static analysis techniques. These three types of techniques correspond to three parts in the fuzzing system structure in Fig. 1: the test case generator, monitor, and static analyser.

3.1. *Sample generation techniques*

Sample generation techniques are used to select and mutate seeds and to restrict and generate new samples. These techniques are implemented in the test case generator shown in Fig. 1 and include three main types: random mutation (Hocevar, 2010; Miller et al., 1990), grammar representation (Banks et al., 2006; Godefroid et al., 2008; Pham et al., 2016; Röning et al., 2002; Peachtec, 2017; Bradshaw, 2010; Fitblip, 2016) and scheduling algorithms (Böhme et al., 2016, 2017; Householder and Foote, 2012; Woo et al., 2013).

3.1.1. *Random mutation*

The core idea of random mutation (Hocevar, 2010; Miller et al., 1990) is to generate new input from a prepared seed by randomly mutating some field. These generated samples are input into a program, which is then watched for crashes that might occur.

Fuzzing systems based on random mutation have been extensively used for the security testing of applications. They can be quickly implemented and are highly scalable since they do not rely on complex computations (Cadar et al., 2008; Cha et al., 2012; Chipounov et al., 2011; Godefroid et al., 2008) or heavy program monitoring techniques (Bhansali et al., 2006; Luk et al., 2005; Nethercote and Seward, 2007; Bruening, 2017; Paradyn, 2017). Nevertheless, this approach suffers from serious limitations in uncovering complex flaws. First, it is blind to the current state of program execution, so there is no aim to the mutation of the seed. In addition, if the input data contains magic bytes or checksum bytes, or if the program contains nested conditions, then the seed cannot access deeper parts of the program.

3.1.2. *Grammar representation*

The random mutation technique (Hocevar, 2010; Miller et al., 1990) is not efficient for finding deep bugs in the target program due to its blind emission of random data. Most of the test cases are rejected in the early steps of the program execution at parsing or checksum verification. The grammar representation technique (Banks et al., 2006; Godefroid et al., 2008; Pham et al., 2016; Röning et al., 2002; Peachtec, 2017; Bradshaw, 2010; Fitblip, 2016), introduced in the 1970s (Hanford, 1970; Purdom, 1972), overcomes this problem by using grammar to constrain the data structure of the test case.

Several methods can be used to describe the input data structure, including block-based (Bradshaw, 2010), grammar-based (Röning et al., 2002) and model-based (Peachtec, 2017) techniques. The input data structure can be obtained automatically by using a dynamic analysis technique (Bastani et al., 2017; Cui et al., 2008; Höschele and Zeller, 2016; Kifetew et al., 2017; Kim et al., 2013; Lin and Zhang, 2008) or by manually writing grammar files (Peachtec, 2017) and codes (Bradshaw, 2010). Several fuzzing systems have combined this approach with random mutation (Banks et al., 2006; Peachtec, 2017; Bradshaw, 2010; Fitblip, 2016) and dynamic symbolic execution (Godefroid et al., 2008; Majumdar and Xu, 2007; Pham et al., 2016).

Grammar representation is most effective for testing applications with complex structured input formats. However, it requires a high level of expertise on the target application or input format. Furthermore, the specified format is typically written by hand, and this process is laborious, time consuming, and error-prone.

3.1.3. *Scheduling algorithms*

Scheduling algorithms (Böhme et al., 2016, 2017; Householder and Foote, 2012; Woo et al., 2013) are feasible methods to maximize the fuzzing outcome using an optimized seed choosing strategy and seed mutation strategy. For example, how to search over the parameter space of fuzzing (Householder and Foote, 2012; Woo et al., 2013) is stated as a multi-armed bandit (MAB) problem (Berry and Fristedt, 1985; Woo et al., 2013). A scheduling algorithm can contain many more algorithms, e.g., simulated annealing algorithms (Böhme et al., 2017), Markov algorithms (Böhme et al., 2016), and statistical algorithms (Householder and Foote, 2012; Woo et al., 2013). These algorithms can improve the efficiency of fuzzing but cannot resolve the basic problem in fuzzing.

3.2. *Dynamic analysis techniques*

Dynamic analysis techniques are used to obtain dynamic information on the running program to help generate the new sample. This information includes symbolic expressions, the executed path, taint information on the sample and codes. These techniques are implemented in the monitor in Fig. 1. Three widely used dynamic analysis techniques include dynamic symbolic execution (Cha et al., 2012; Chipounov et al., 2011; Godefroid et al., 2008), coverage feedback (Kargén and Shahmehri, 2015; Rawat et al., 2017; Stephens et al., 2016; Zalewski, 2016; Google 2017a, 2017b; LLVM-admin team, 2017) and dynamic taint analysis (Drewry and Ormandy, 2007; Ganesh et al., 2009; Haller et al., 2013; Neugschwandtner et al., 2015; Newsome and Song, 2005; Rawat et al., 2017; Wang et al., 2010).

3.2.1. *Dynamic symbolic execution*

Dynamic symbolic execution (Cha et al., 2012; Chipounov et al., 2011; Godefroid et al., 2008) is a method to determine the possible inputs of a program. This task is accomplished by using symbolic values as an input to execute the program, at the same time collecting symbolic constraints on the path set, then inverting a constraint to generate a new path; finally, a new input is generated by the SMT solver (Bjørner and Phan, 2014; De Moura and Bjørner, 2008; Ganesh and Dill, 2007) on this new constraint set, which represents a new path. Fuzzing systems using dynamic symbolic execution may have

different aims, e.g., to maximize code coverage (Campana, 2009; Caselden et al., 2013; Godefroid et al., 2005, 2008) or to focus on a specific location (Haller et al., 2013; Neugschwandtner et al., 2015) that is more likely to be vulnerable.

The main limitation of dynamic symbolic execution-based fuzzing is path explosion. When symbolic execution addresses real-world applications, exponential symbolic expressions will be impossible for the constraint solver to solve. Executing all feasible program paths cannot scale to large realistic programs. Thus, flaws that lie in the deeper logic of a program are usually discovered by manual analysis (Bucur, 2015; Chen et al., 2013; DeMott, 2014).

3.2.2. *Coverage feedback*

The coverage feedback technique (Zalewski, 2016; Google 2017a, 2017b) is used to generate inputs for traversing different paths in the program execution. It uses instrumentation (Bhansali et al., 2006; Luk et al., 2005; Nethercote and Seward, 2007; Bruening, 2010; Paradyn, 2017) to gain path coverage information on the running program. If a new path is generated, the current sample is added to the candidate seed set from which one seed will be chosen to mutate in the next loop; otherwise, it is discarded. With this coverage feedback technique, the fuzzing proceeds in a direction that constantly improves the path coverage.

Coverage-guided fuzzing (Kargén and Shahmehri, 2015; Rawat et al., 2017; Stephens et al., 2016; Zalewski, 2016; Google 2017a, 2017b; LLVM-admin team, 2017) is more efficient than random fuzzing (Hocevar, 2010; Miller et al., 1990) due to its feedback mechanism. However, the generation of test cases to pass complex checks in the application is still challenging for this fuzzer, and the feedback loop used by coverage-guided fuzzing does not relate application behaviour to the input structure to improve input generation.

3.2.3. *Dynamic taint analysis*

Dynamic taint analysis (Drewry and Ormandy, 2007; Ganesh et al., 2009; Haller et al., 2013; Neugschwandtner et al., 2015; Newsome and Song, 2005; Rawat et al., 2017; Wang et al., 2010) is used to infer the structural properties of input data and which offsets in the input influence the branch conditions. This information can be used to efficiently choose and mutate the seeds. Dynamic taint analysis executes the target program by feeding it input data with tags, which can be used to specify how the program uses the input data and which program elements were tainted by the data. Dynamic taint analysis can be combined with dynamic symbol execution (Wang et al., 2010) and with random mutation (Ganesh et al., 2009) to improve the precision of fuzzing.

Dynamic taint analysis can track and detect the explicit propagation and misuse of tainted data in the memory by monitoring the program, but it has the problems of under-tainting and over-tainting (Kang et al., 2011). Under-tainting is a type of error in which values that should be marked as tainted are not, and over-tainting is a type of error in which too many values are marked as tainted.

3.3. *Static analysis techniques*

Static analysis includes control flow analysis (Kinder et al., 2009; Sparks et al., 2007) and data flow slices (Tip, 1994; Weiser, 1979, 1981, 1982). Both techniques are used to efficiently locate, lead execution to, and verify possible vulnerabilities.

A control flow graph (Kinder et al., 2009; Sparks et al., 2007) is a directed graph in which the nodes represent basic blocks, and the edges represent control flow paths. Control flow information acts as a map that leads the program execution towards the potentially vulnerable spot. Data flow slices (Tip, 1994; Weiser, 1979, 1981, 1982) extract the parts of a program that potentially affect the values computed at some point of interest, gathering statements and control predicates by way of a forward or backward traversal of the program's control flow graph (Kinder et al., 2009; Sparks et al., 2007) or program dependence graph (Ferrante et al., 1987; Kuck et al., 1981). Data flow slices (Tip, 1994; Weiser, 1979, 1981, 1982), which relate to the statement triggering the vulnerability, can be used to identify possible vulnerabilities.

The undecidability (Landi, 1992) of static analysis hinders the precise provision of dynamical memory allocations and complex data structures. Although static analysis often produces false positive results, it can be combined with other approaches to obtain highly valuable preprocessing information.

3.4. *Other*

Machine learning (Dietterich, 1997) is the science of leading computers to act without being explicitly programmed. Broadly, three types of machine learning algorithms exist: supervised learning (Caruana and Niculescu-Mizil, 2006), unsupervised learning (Barlow, 1989), and reinforcement learning (Bhatnagar et al., 1998).

A supervised learning (Caruana and Niculescu-Mizil, 2006) algorithm consists of a target variable to be predicted from a given set of independent variables. Using this set of variables, the algorithm can generate a function that maps inputs to desired outputs. For example, if we want to generate a model to predict whether all the codes in a path are vulnerable or not, we use several parameters as inputs: lines of codes in the path, whether a specific type of code is present or not, whether the memory or registers related to the specific code are tainted or not, whether a specific function is invoked or not, etc. In addition, the output is whether the codes are vulnerable or not. An unsupervised learning (Barlow, 1989) algorithm has no target or outcome variable to predict. A typical example of unsupervised learning is clustering, which purpose is to gather similar things together. A reinforcement learning (Bhatnagar et al., 1998) algorithm learns from past experience and tries to capture the best possible knowledge to make accurate business decisions.

Artificial neural networks (Koprinkova-Hristova et al., 2015) are a commonly applied type of machine learning algorithm. An artificial neural network generally comprises a collection of interconnected artificial neurons that perform some computation on input patterns and create output patterns. Artificial neural networks are adaptive systems capable of modifying their internal structure, typically the weights between nodes in the network, allowing them to be used for a variety of function approximation problems such as classification, regression, feature extraction and content-addressable memory.

Recently, increasing numbers of works have used machine learning, especially artificial neural network techniques, for program analysis and synthesis, including array sorting and

copying (Kurach et al., 2015; Reed and De Freitas, 2015), encoding input-output examples (Parisotto et al., 2016), repairing syntax errors in programs (Bhatia and Singh, 2016; Gupta et al., 2017; Pu et al., 2016), optimizing assembly programs (Bunel et al., 2016), intelligently guiding fuzzing (Godefroid et al., 2017), predicting vulnerabilities (Grieco et al., 2016; Yamaguchi et al., 2011), malware analysis (Santos et al., 2013), and attack detection (Forrest et al., 1996; Rawat et al., 2005; Tandon and Chan, 2003).

Machine learning will become a promising technique in the field of program analysis and security detection. Although machine learning has made great progress in other fields, its combination with fuzzing is still in the initial stage. It only yields a moderate improvement of fuzzing at present.

4. Representative fuzzing systems

The first fuzzing system was implemented by Miller et al. (1990) to analyse the reliability of UNIX tools. Since then, fuzzing techniques have been increasingly applied to software security detection. Increasing numbers of systems introduce various techniques to improve the efficiency of fuzzing. The development of fuzzing can be divided into 4 main stages: before 2005, 2006–2010, 2011–2015, and 2016–2017. We will introduce some representative fuzzing systems from each of these time periods.

4.1. *Before 2005*

Before 2005, fuzzing was still in the early stages of development. The fuzzing techniques in this period consist mainly of black-box fuzzing using random mutation. The first fuzzing system (Miller et al., 1990) is a typical random fuzzing approach. SPIKE (Bradshaw, 2010) and Peach (Peachtec, 2017) introduced the grammar representation technique to improve the quality of the generated samples for the random fuzzing (Hocevar, 2010; Miller et al., 1990). Fuzzing with dynamic symbolic execution, for example CUTE (Sen et al., 2005), began to be applied later in this period.

The first fuzzing system (Miller et al., 1990) was constructed in 1990. It tests the target program by inputting random strings to see whether the program crashes or not, then analyses the crashes. This fuzzer contains two programs: fuzz and ptyjig. Fuzz generates a stream of random characters to be consumed by the target program, and ptyjig automatically tests interactive utilities by writing scripts. The evaluation showed that crashes occurred in more than 24% of the 90 different utility programs in seven versions of UNIX. This fuzzing system provides a new approach to detect vulnerabilities in programs, but it has poor testing results for programs with complex structured input data because most of the mutated inputs are rejected in the early steps of the program execution either at parsing or at the checksum verification process.

SPIKE (Bradshaw, 2010) provides different APIs to generate input data, checksum and routines with less effort. It specifies complex formatted input data as nested data blocks that satisfy the relations of the different fields. The fields may be of varying types, e.g., int, float, or string. It calculates the length and checksum data to pass the integrity checking at the very beginning of the program execution and uses routines to facilitate handling network code and common marshalling routines. To extend the data format or modify the testing routines, the source code must be modified, which increases the difficulty of extension.

Peach (Peachtec, 2017) is a model-based fuzzer that uses Peach Pit to enhance the extensibility of fuzzing for different types of input structures and protocols. Peach Pit separates the description of the input structure and testing routines from the engine of the fuzzer. It consists of a series of XML files including DataModel (Peachtec, 2014a) and StateModel (Peachtec, 2014b). DataModel describes the types of data chunks and fields and the relationships between them. Each chunk, which contains the type, length, data, checksum, etc., describes an independent data structure. The field is used to describe a single type of data. StateModel describes the state transitions of the protocol and how to run the tests. Peach (Peachtec, 2017) performs fuzzing according to the routines specified by StateModel, mutates the seed according to DataModel, and uses monitor agents that check if and when the program has crashed. Peach (Peachtec, 2017) supports fixups to repair checksums and transformers for encoding, decoding and compression. The source code of Peach (Peachtec, 2017) does not need to be modified to support different programs and protocols. Extension requires only the addition of Peach Pit, but Peach (Peachtec, 2017) cannot adjust the seed selection and mutation strategies according to the runtime information of the program due to its black-box property.

CUTE (Sen et al., 2005) addresses the problem that the satisfaction of constraints in symbolic execution may be undecidable when programs have pointer operations. It separates pointer constraints from integer constraints to make symbolic execution lightweight and the constraint solving procedure efficient. Specifically, CUTE (Sen et al., 2005) represents all inputs with a logical input map that maps logical addresses to values and runs the code concretely and symbolically based on it. CUTE (Sen et al., 2005) collects constraints and negates individual ones to generate a new logical input map. It then resolves the map, generates a new input using SMT solver (Bjørner and Phan, 2014; De Moura and Bjørner, 2008; Ganesh and Dill, 2007), and repeats this process to explore feasible execution paths as extensively as possible. CUTE (Sen et al., 2005) can find bugs in actual programs but is limited to programs with source code, and path explosion is its primary challenge.

4.2. *2006–2010*

Between 2006 and 2010, symbolic execution-based fuzzing was well developed, and the taint analysis technique was used in fuzzing systems. KLEE (Cadar et al., 2008) and SAGE (Godefroid et al., 2008) improve the efficiency of fuzzing by expanding the fuzzing coverage with symbolic execution. This approach is limited by the well-known path explosion problem affecting symbolic execution. This problem becomes evident as the complexity of a binary increases. TaintScope (Wang et al., 2010), BuzzFuzz (Ganesh et al., 2009) and GWF (Godefroid et al., 2008) improve the efficiency of fuzzing by restricting the search space. TaintScope (Wang et al., 2010) aims to pass the checksum code by using symbolic execution and taint analysis. BuzzFuzz

(Ganesh et al., 2009) aims at library and system calls in the programs using taint analysis. GWF (Godefroid et al., 2008) restricts the search space to valid inputs to explore deeper paths using symbolic execution (Cadar et al., 2008; Cha et al., 2012; Chipounov et al., 2011; Godefroid et al., 2008) and grammar representation (Banks et al., 2006; Röning et al., 2002; Peachtec, 2017; Bradshaw, 2010; Fitblip, 2016).

KLEE (Cadar et al., 2008) uses search heuristics on symbolic execution to achieve high code coverage. It compiles the source code of the program to the LLVM (Lattner and Adve, 2004) byte code, an RISC-like virtual instruction set. KLEE (Cadar et al., 2008) executes the byte code using symbolic values as inputs. If the execution branches based on a symbolic value, KLEE (Cadar et al., 2008) will follow both branches, maintaining a set of constraints on each path that keep execution on that path. If a path terminates or hits a bug, a new input will be generated by solving the current constraints. Experimentation showed that KLEE (Cadar et al., 2008) could achieve higher code coverage and found several serious bugs in some applications (Meyering, 2017), but only open-source applications.

SAGE (Godefroid et al., 2008) is a successful commercial fuzzing system that uses symbolic execution (Cadar et al., 2008; Cha et al., 2012; Chipounov et al., 2011; Godefroid et al., 2008) on binary programs. It uses a generational search algorithm to improve fuzzing on applications that accept highly structured inputs by attempting to sweep through all feasible execution paths of the program. In the generational search algorithm, SAGE (Godefroid et al., 2008) collects the path constraints for the execution flow when processing the input seed file. Each of the constraints in the constraint set is negated and combined with the constraints that preceded it. If the resulting constraint set is solvable by calling the constraint solver (Bjørner and Phan, 2014; De Moura and Bjørner, 2008; Ganesh and Dill, 2007), a new input set is generated and scored based on the increase in code coverage. All of the generated seeds are stored in a candidate seed file pool, and the candidate with the highest score is selected from the seed pool and used as the seed to begin the next round of fuzzing. SAGE (Godefroid et al., 2008) has already discovered many bugs in Windows applications, several of which are potentially exploitable, but SAGE (Godefroid et al., 2008) has difficulty finding bugs in some applications with complex checksum codes, which may lead to path explosion.

TaintScope (Wang et al., 2010) is an automatic dynamic symbolic execution (Cadar et al., 2008; Cha et al., 2012; Chipounov et al., 2011; Godefroid et al., 2008) fuzzing system used in combination with dynamic taint analysis (Ganesh et al., 2009; Haller et al., 2013; Neugschwandtner et al., 2015; Wang et al., 2010) to pass the checksum mechanism employed in the target program. It identifies checksum fields in the input data using combined concrete and symbolic execution techniques such as Tupni (Cui et al., 2008), and employs a branch profiling technique to locate checksum-based integrity checks accurately, bypass such checks by altering the control flow, and automatically fix the checksum values in the generated inputs using dynamic symbolic execution (Cadar et al., 2008; Cha et al., 2012; Chipounov et al., 2011; Godefroid et al., 2008). TaintScope (Wang et al., 2010) found 27 undiscovered vulnerabilities in several commercial software programs, including Adobe Acrobat, Google Picasa, Microsoft Paint and ImageMagick (ImageMagick Studio LLC, 2017).

BuzzFuzz (Ganesh et al., 2009) improves the mutation efficiency by mutating only the tainted section of the input. It selects library and system calls as attack points and uses dynamic taint analysis (Ganesh et al., 2009; Haller et al., 2013; Neugschwandtner et al., 2015; Wang et al., 2010) to automatically locate regions of the original seed input files that influence the values used at these attack points, then mutate these identified regions of the original seed input to obtain new test input. The newly generated test input tends to pass the initial input parsing code due to preserving the syntactic structure of the original seed, so it can trigger bugs hidden in the deep layer of the program. BuzzFuzz (Ganesh et al., 2009) cannot test a binary program directly because of the interaction of its dynamic taint analysis technique with the source code of the program.

GWF (Godefroid et al., 2008) enhances dynamic symbolic execution-based fuzzing (Cadar et al., 2008; Cha et al., 2012; Chipounov et al., 2011; Godefroid et al., 2008) with a grammar-based specification of valid inputs. It uses symbolic execution to directly generate the grammar-based constraints (Röning et al., 2002), whose satisfiability is checked by a custom grammar-based constraint solver. It completes a partial set of token constraints to form a fully defined valid input, so it can explore deeper paths by restricting the search space to valid inputs. An evaluation comparing it with dynamic symbolic execution-based fuzzing (Cadar et al., 2008; Cha et al., 2012; Chipounov et al., 2011; Godefroid et al., 2008), random fuzzing (Hocevar, 2010; Miller et al., 1990) and grammar-based fuzzing (Banks et al., 2006; Röning et al., 2002; Peachtec, 2017; Bradshaw, 2010; Fitblip, 2016) showed that GWF (Godefroid et al., 2008) had an advantage in code coverage using fewer tests.

4.3. 2011–2015

Between 2011 and 2015, the development of fuzzing systems had three main characteristics. First, coverage-guided fuzzing (Böhme et al., 2016, 2017; Zalewski, 2016; Google 2017a, 2017b) was widely used in academic and industrial fields; the representative system is AFL (Zalewski, 2016). Second, different scheduling algorithms (Böhme et al., 2016, 2017; Householder and Foote, 2012; Woo et al., 2013) were used in fuzzing. CERT BFF (Householder and Foote, 2012) and FuzzSim (Woo et al., 2013) model fuzzing as Bernoulli trials (Saperstein, 1973). SYMFUZZ (Cha et al., 2015) computes an optimal mutation ratio for fuzzing using taint analysis (Ganesh et al., 2009; Haller et al., 2013; Neugschwandtner et al., 2015; Wang et al., 2010). Third, a trend developed to improve fuzzing efficiency by combining multiple techniques. Dowser (Haller et al., 2013) and BORG (Neugschwandtner et al., 2015) direct fuzzing to specific points in the program that may be vulnerable by combining static analysis (Kinder et al., 2009; Sparks et al., 2007; Tip, 1994; Weiser, 1979, 1981, 1982), taint propagation (Ganesh et al., 2009; Haller et al., 2013; Neugschwandtner et al., 2015; Wang et al., 2010) and symbolic execution (Cadar et al., 2008; Cha et al., 2012; Chipounov et al., 2011; Godefroid et al., 2008).

AFL (Zalewski, 2016) is a brute-force fuzzer that automatically generates interesting test cases that are relatively likely to trigger new internal states in the target program. When fuzzing starts, AFL (Zalewski, 2016) loads user-supplied test cases

into a queue, takes the next input file from the queue and attempts to trim test cases that do not alter the measured behaviour of the program to minimize the number of test cases, repeatedly mutating the file using a balanced and well-researched variety of traditional fuzzing strategies (Lcamtuf, 2014) in which most of the work is done by using deterministic strategies first followed by random stacked modification strategies and finally test case splicing. If any of the generated mutations result in the recording of a new state transition by the instrumentation (Bhansali et al., 2006; Luk et al., 2005; Nethercote and Seward, 2007; Bruening, 2017; Paradyn, 2017), AFL (Zalewski, 2016) adds the mutated output as a new entry in the queue, then loops to take the next input file from the queue. AFL (Zalewski, 2016) substantially improves the functional coverage of the fuzzed code and has modest performance overhead. Some notable vulnerabilities and other uniquely interesting bugs have been found by AFL (Lcamtuf, 2017).

CERT BFF (Householder and Foote, 2012) improves random fuzzing by using a fuzzing parameter selection algorithm based on modelling the fuzzing process as a sequence of Bernoulli trials (Saperstein, 1973). First, given a set of seed files/ranges, each file is assigned an initial crash density, which is the empirically measured number of unique crashes found while fuzzing the file divided by the number of trials attempted. CERT BFF (Householder and Foote, 2012) next calculates the chosen probability distribution for each file/range on the set of seed files and chooses a file/range according to the distribution, fuzzes the file for an interval of iterations, calculates the upper 95% confidence interval on the observed crash density for that file/range, and updates the probability distribution of the seed file, and repeats this process continuously. An evaluation showed that CERT BFF (Householder and Foote, 2012) markedly improved the efficiency of detecting errors in FFmpeg (Niedermayer, 2017) over basic parameter selection methods.

FuzzSim (Woo et al., 2013) also considers fuzzing as independent Bernoulli trials (Saperstein, 1973). It models the fuzzing process as a weighted coupon collectors problem with unknown weights to capture the decrease in the probability of finding a new bug and designs online algorithms using the condition in the no free lunch theorem (Wolpert and Macready, 1997) to maximize the number of bugs found in a fuzz campaign. An evaluation showed an average improvement of 1.5 times over BFF (Householder and Foote, 2012) in the efficiency of discovering unique application errors.

SYMFUZZ (Cha et al., 2015) specifies an effective mutation ratio to improve the mutation strategy by using taint analysis. It analyses the execution trace for a given program-seed pair to detect dependencies among the bit data of an input by leveraging taint analysis and specifies an effective mutation ratio, which is related to the fuzzing efficiency, then performs traditional random mutational fuzzing with the specified mutation ratio. SYMFUZZ (Cha et al., 2015) does not rely on SMT solvers (De Moura and Bjørner, 2008; Ganesh and Dill, 2007) but uses a dependency relation to compute an optimal mutation ratio for the program-seed pair. An evaluation showed that an average of 38.6% more bugs were found than by the three previous fuzzers, including BFF (Householder and Foote, 2012), ZZuf (Hocevar, 2010) and AFL (Zalewski, 2016), over 8 applications in the same fuzzing time.

Dowser (Haller et al., 2013) effectively uses static analysis (Kinder et al., 2009; Sparks et al., 2007; Tip, 1994; Weiser, 1979, 1981, 1982), symbolic execution (Cadar et al., 2008; Cha et al., 2012; Chipounov et al., 2011; Godefroid et al., 2008) and taint analysis (Ganesh et al., 2009; Haller et al., 2013; Neugschwandtner et al., 2015; Wang et al., 2010) in combination to detect underflow (Taborn and Yuan, 1996) and overflow (Sidiroglou et al., 2005) vulnerabilities. It leverages static analysis to identify possible locations for buffer overflow bugs within loops and ranks them based on a metric that evaluates the complexity of the access (Gegick et al., 2008; Nguyen and Tran, 2010; Shin and Williams, 2011; Zimmermann et al., 2010), confirms the input bytes that influence these interesting cases of array access by taint analysis, then symbolizes these bytes and starts a symbolic execution process, solves the constraint paths that are most likely to lead to underflow and overflow, and finally generates the input to be verified. Dowser (Haller et al., 2013) works well in real programs. Two of the bugs found by Dowser (Haller et al., 2013) were previously undocumented buffer overflows in FFmpeg (Niedermayer, 2017) and the poppler PDF rendering library.

BORG (Neugschwandtner et al., 2015) guides symbolic execution to target instructions that are prone to triggering overread vulnerabilities (Wang et al., 2015) with a combination of static analysis (Kinder et al., 2009; Sparks et al., 2007; Tip, 1994; Weiser, 1979, 1981, 1982) and taint analysis (Ganesh et al., 2009; Haller et al., 2013; Neugschwandtner et al., 2015; Wang et al., 2010). It obtains a control flow graph (Kinder et al., 2009; Sparks et al., 2007) using static analysis, then refines it with knowledge from dynamically generated execution traces to generate an accurate model of the program's intra- and interprocedural control flow graph, which is then used to compute a distance metric between the program's current execution path and a target instruction. The target instructions are sensitive functions, such as memcpy and strcpy, if their parameters are tainted. The paths are ranked by the distance metric and used to guide symbolic execution (Cadar et al., 2008; Cha et al., 2012; Chipounov et al., 2011; Godefroid et al., 2008) to focus on the execution paths that are most relevant to the target instruction. BORG (Neugschwandtner et al., 2015) detected overread vulnerabilities (Wang et al., 2015) in several complex server applications and libraries, including lighttpd (Bohler, 2017), FFmpeg (Niedermayer, 2017) and ClamAV (Talos, 2017b).

4.4. 2016–2017

From 2016 to 2017, the development of fuzzing has shown three main characteristics. The first characteristic is the emergence of multiple fuzzing systems that improved AFL (Zalewski, 2016), such as AFLFast (Böhme et al., 2016), AFLGo (Böhme et al., 2017), Skyfire (Wang et al., 2017), VUzzer (Rawat et al., 2017) and Steelix (Li et al., 2017). The second is the continued trend of combining multiple techniques, such as in MoWF (Pham et al., 2016), Driller (Stephens et al., 2016), VUzzer (Rawat et al., 2017) and Steelix (Li et al., 2017). The third is the introduction of machine learning techniques into fuzzing, such as in Learn&Fuzz (Godefroid et al., 2017).

AFLFast (Böhme et al., 2016) improves AFL (Zalewski, 2016) by leveraging the Markov algorithm (Norris, 1998). It models the probability that fuzzing a seed that exercises program path

i generates a seed that exercises path j with the transition probability pij in a Markov chain (Norris, 1998). AFLFast (Böhme et al., 2016) implements a power schedule to assign an energy that is inversely proportional to the density of the stationary distribution. When the seed is fuzzed for the first time, a very low energy is assigned. Every time the seed is chosen thereafter, exponentially more inputs are generated up to a certain bound. This process can rapidly approach the minimum energy required to discover a new path. AFLFast (Böhme et al., 2016) exposed 3 previously uncovered vulnerabilities that were not detected by AFL (Zalewski, 2016) and exposed 6 previously uncovered vulnerabilities 7 times faster than AFL (Zalewski, 2016) in 24 hours.

AFLGo (Böhme et al., 2017) is an AFL-based fuzzing system using a simulated annealing (Kirkpatrick et al., 1983) power schedule to generate inputs with the objective of efficiently reaching a given set of target program locations. The simulated annealing-based power schedule gradually assigns more energy to seeds that are closer to the target locations while reducing energy for seeds that are further away. The target program locations are identified by setting a changed statement in patch testing (Böhme et al., 2013; Marinescu and Cadar, 2013) and by setting a method call in the stack-trace in crash reproduction (Jin and Orso, 2012; Pham et al., 2015). In an experiment, AFLGo (Böhme et al., 2017) found 39 bugs in several security-critical projects such as LibXML (Reese, 2017) and LibMing (Reczey, 2017).

Skyfire (Wang et al., 2017) uses a PCSG (probabilistic context-sensitive grammar) to generate high-quality samples from the vast number of existing samples. A PCSG has a pool of production rules to generate seed samples. In particular, each production rule is associated with a context in which the rule can be applied, which can help the generated seeds to pass syntax parsing and semantic checking. In addition, each rule is associated with the probability that the production rule is applied under the given text, which can guide subsequent seed selection and reduce seed redundancy. Skyfire (Wang et al., 2017) leverages the learned PCSG to generate seed inputs, then feeds the initial samples and the samples generated as seeds of AFL (Zalewski, 2016) to the fuzzing process. This method can be used to discover deep bugs beyond the syntax parsing and semantic checking stage. Skyfire (Wang et al., 2017) discovered 19 new memory corruption bugs and 32 denial-of-service bugs from the JavaScript and rendering engine of Internet Explorer 11.

VUzzer (Rawat et al., 2017) guides the fuzzing to penetrate more deeply into the application by calculating the fitness of each input. First, it extracts control flow features to specify the weight of each basic block. Specifically, it sets a negative value for a basic block that contains error-handling code. Then, it calculates the weighted sum of the executed basic blocks as the path fitness. Next, it extracts data flow features to accurately determine where and how to mutate such inputs by using taint analysis, then continually guides the fuzzing to the path with the high fitness score to detect the vulnerabilities hidden deep in the program. An evaluation showed that VUzzer (Rawat et al., 2017) quickly found several existing and new bugs when compared with AFL (Zalewski, 2016) on DARPA Grand Challenge binaries (DARPA, 2016), the LAVA dataset (Dolan-Gavitt et al., 2016) and a set of real-world applications.

Steelix (Li et al., 2017) improves the penetration power of AFL (Zalewski, 2016) using coverage, magic bytes comparison progress and location information. It locates the magic bytes in the inputs and performs mutations to match the magic bytes by using lightweight static analysis (Kinder et al., 2009; Sparks et al., 2007; Tip, 1994; Weiser, 1979, 1981, 1982) and binary instrumentation (Bhansali et al., 2006; Luk et al., 2005; Nethercote and Seward, 2007; Bruening, 2010; Paradyn, 2017). If a mutation makes progress towards matching magic bytes, Steelix (Li et al., 2017) will keep the new test input together with the position of the byte that was just mutated. When Steelix (Li et al., 2017) mutates the new test input, it will use a local exhaustive mutation to try all the possibilities of the two neighbour bytes. Using this method, Steelix (Li et al., 2017) can efficiently match the magic bytes and penetrate deeper than AFL (Zalewski, 2016) into program execution. Steelix (Li et al., 2017) has been evaluated on the LAVA-M dataset (Dolan-Gavitt et al., 2016), DARPA CGC (DARPA, 2016) sample binaries and five real-life programs and outperformed AFL (Zalewski, 2016) with respect to code coverage and bug detection capability.

Driller (Stephens et al., 2016) combines the speediness of coverage-guided techniques and the reasoning ability of symbolic execution techniques. The use of a coverage-guided technique (Böhme et al., 2016, 2017; Zalewski, 2016; Google 2017a, 2017b) can improve the path coverage in the program. Symbolic execution (Cadar et al., 2008; Cha et al., 2012; Chipounov et al., 2011; Godefroid et al., 2008) can reason the complex checks on specific input between compartments. These complex checks are difficult to pass by using the coverage-guided tools (Böhme et al., 2016, 2017; Zalewski, 2016; Google 2017a, 2017b). Driller (Stephens et al., 2016) uses the coverage-guided tool AFL (Zalewski, 2016) to explore the application until it reaches the complex checks on specific input between compartments. Then, it invokes its symbolic execution engine, Angr (Shellphish, 2017), to analyse the application and generate a seed to pass the check. Then, Driller (Stephens et al., 2016) returns to performing fuzzing using AFL (Zalewski, 2016). Driller (Stephens et al., 2016) was used to test 126 binaries from the DARPA Cyber Grand Challenge Qualifying Event (DARPA, 2016) and found 77 crashes, whereas 68 crashes were found using only AFL (Zalewski, 2016).

MoWF (Pham et al., 2016) uses an input model (Peachtec, 2017) to relieve the difficulty of using symbolic execution to solve the constraints generated from codes that contain complex integrity checks. It cracks all initial files and places their data components inside a fragment pool, identifies crucial branches by their dependence on a data field in a given file of enumerable type using taint analysis (Ganesh et al., 2009; Haller et al., 2013; Neugschwandtner et al., 2015; Wang et al., 2010), then learns the type of the referenced data field using the input model, adds a data chunk obtained from the fragment pool or removes the data chunk if the data field is of enumerable type, re-establishes the integrity of the file using fixup and transformers supplied by Peach (Peachtec, 2017) framework, and finally employs selective symbolic execution (Chipounov et al., 2011) to explore the crucial branches that are exercised depending on specific values of the data fields with a guide to critical locations that may expose a vulnerability if exercised by an appropriate input. These critical locations, where program instructions may trigger

divide-by-zero and null-pointer dereference vulnerabilities, are identified by lightweight analysis using IDAPro (Hex-Rays SA, 2017). MoWF (Pham et al., 2016) can find bugs hidden in the depths of a program and exposed 13 vulnerabilities in 8 large program binaries, whereas traditional white-box fuzzing (Pham et al., 2015) and model-based black-box fuzzing (Peachtec, 2017) exposed less than half as many.

Learn&Fuzz (Godefroid et al., 2017) presents a novel algorithm that intelligently guides where to fuzz input data using a seq2seq (Cho et al., 2014; Sutskever et al., 2014) neural network-based technique. This approach is used for learning the model of PDF objects over the set of PDF object characters given a large corpus of objects. The seq2seq network model learns contexts of arbitrary length to predict the next character sequence. It learns a generative model to generate new PDF objects using a set of input and output sequences in an unsupervised manner. The input sequences are the characters sequences extracted from PDF objects. The output sequences are obtained by shifting the input sequences by one position. Experimental results showed that this approach improved the coverage of the PDF parser compared to random fuzzing.

5. Analysis and comparison of fuzzing systems

Since the emergence of the first fuzzing system, many fuzzing systems have been constructed. These systems introduce different techniques to improve the efficiency of fuzzing. Some of these systems use the same technique to achieve certain goals, and some systems use different techniques to achieve the same goal. Some correlations exist among these systems, techniques and goals. In this section, we will compare these fuzzing systems from different perspectives: the timeline, symbolic execution technique (Cadar et al., 2008; Cha et al., 2012; Chipounov et al., 2011; Godefroid et al., 2008), coverage feedback (Böhme et al., 2016, 2017; Zalewski, 2016; Google, 2017a, 2017b), grammar representation (Röning et al., 2002; Peachtec, 2016; Bradshaw, 2010) and taint analysis (Ganesh et al., 2009; Haller et al., 2013; Neugschwandtner et al., 2015; Wang et al., 2010).

5.1. *The timeline of fuzzing system development*

Fuzzing techniques have continued to develop for more than 20 years since 1990, when Miller et al. (1990) implemented the first fuzzing system. Fuzzing has evolved from the original black-box fuzzing (Miller et al., 1990; Peachtec, 2017; Fitblip, 2016; Helin, 2017) to white-box (Cadar et al., 2008; Godefroid et al., 2005, 2008; Haller et al., 2013) and grey-box (Böhme et al., 2016, 2017; Zalewski, 2016; Google 2017a, 2017b) fuzzing with the combination of multiple techniques, including grammar representation (Banks et al., 2006; Röning et al., 2002; Peachtec, 2017; Bradshaw, 2010; Fitblip, 2016), static analysis (Kinder et al., 2009; Sparks et al., 2007; Tip, 1994; Weiser, 1979, 1981, 1982), taint analysis (Ganesh et al., 2009; Haller et al., 2013; Neugschwandtner et al., 2015; Wang et al., 2010), scheduling algorithms (Böhme et al., 2016, 2017; Householder and Foote, 2012; Woo et al., 2013), and machine learning (Godefroid et al., 2017). Fig. 3 lists the various techniques used in well-known fuzzing tools or systems in chronological order (Fig. 4) (Fig. 5).

(1) The complexity of techniques

As the picture above shows, all kinds of techniques are increasingly combined with each other over time, so fuzzing systems are becoming increasingly complex. In the early stage, single fuzzing techniques such as fuzz (Miller et al., 1990; Wikipedia, 2017b), KLEE (Cadar et al., 2008), SAGE (Godefroid et al., 2008), AFL (Zalewski, 2016) were used; in later fuzzing systems, fuzzing was combined with one additional technique, such as in Peach (Peachtec, 2017), GWF (Godefroid et al., 2008), TaintScope (Wang et al., 2010), and AFLFast (Böhme et al., 2016); and in recent years, combinations of multiple techniques have arisen, such as Dowser (Haller et al., 2013), BORG (Neugschwandtner et al., 2015), and MoWF (Pham et al., 2016). Before 2013, fuzzing systems were mainly based on one or two techniques. After 2013, fuzzing systems were based on combinations of multiple techniques.

A newly emerged technique will be constantly combined with other fuzzing techniques to achieve certain effects: for example, taint analysis was combined with random mutation to produce BUZZFUZZ (Ganesh et al., 2009) and SYMFUZZ (Cha et al., 2015) and with dynamic symbolic execution to produce TaintScope (Wang et al., 2010), Dowser (Haller et al., 2013), and BORG (Neugschwandtner et al., 2015). Dynamic symbolic execution was combined with grammar representation to produce GWF (Godefroid et al., 2008) and MoWF (Pham et al., 2016) and with taint analysis to produce TaintScope (Wang et al., 2010), Dowser (Haller et al., 2013) and BORG (Neugschwandtner et al., 2015). Scheduling algorithms were combined with random mutation to produce BFF (Householder and Foote, 2012) and FuzzSim (Woo et al., 2013) and with coverage guides to produce AFLGo (Böhme et al., 2017) and AFLFast (Böhme et al., 2016).

(2) The same combinations of techniques

Some fuzzing systems use the same combination of techniques shown in the diagram above but with different implementations or for different application domains.

AFL (Zalewski, 2016) and syzkaller (Google, 2017b) use coverage-guided techniques in different application domains: AFL (Zalewski, 2016) is a generic type of fuzzing tool, whereas syzkaller (Google, 2017b) is used for detecting operating system vulnerabilities.

PROTOS (Röning et al., 2002), SPIKE (Bradshaw, 2010) and Peach (Peachtec, 2017) are grammar-based fuzzers. PROTOS (Röning et al., 2002) uses simplified syntax to describe protocol interactions. SPIKE (Bradshaw, 2010) provides users with a programming interface to generate the input data. In Peach (Peachtec, 2017), the user is required to write the Peach Pit files to mutate seeds or to directly generate inputs.

Both FuzzSim (Woo et al., 2013) and BFF (Householder and Foote, 2012) combine random mutation with scheduling algorithms, which contain many more algorithms. BFF (Householder and Foote, 2012) uses an algorithm based on statistical theory to automatically select the seed files and the proportion of the files to randomize. FuzzSim (Woo et al., 2013) uses a weighted

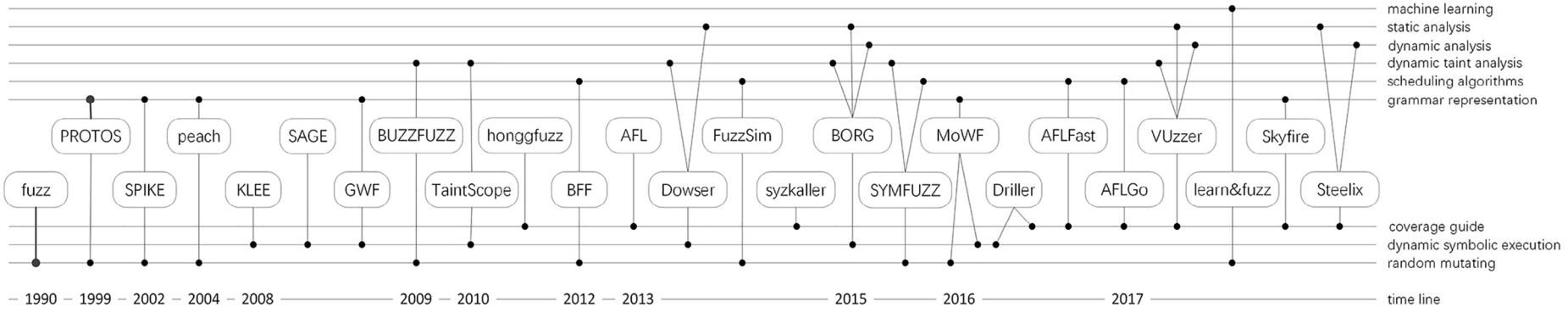

Fig. 3 – Fuzzing Technique Evolution Diagram.

System	Improving Memory Usage		Constraint Expressions Optimization		Directing symbolic execution			
	State Merging	swapping states to disk	summarizing loops	Expressions Simplication	removing checksum code	Pruning Paths	Aiming at transition	Aiming at vulnerability
MoWF						√		√
KLEE	√			√				
Mayhem	√	√						
S2E	√							
BORG	√	√	√			√		√
Taintscope					√			
Dowser				√				√
SAGE		√	√					
GWF						√		
Driller							√	

Fig. 4 – Different Method Alleviating Path Explosion in Fuzzing Systems.

random algorithm with the fixed-time rate belief metric to improve the efficiency of fuzzing.

5.2. *Comparison of fuzzing systems*

To understand the differences among different fuzzing systems in depth, we compare fuzzing systems in terms of four aspects, including dynamic symbolic execution, coverage feedback, grammar representation and taint analysis.

5.2.1. *Dynamic symbolic execution*

The symbolic execution technique has been widely used in fuzzing systems. KLEE (Cadar et al., 2008), SAGE (Godefroid et al., 2008), S2E (Chipounov et al., 2011), Mayhem (Cha et al., 2012), GWF (Godefroid et al., 2008), TaintScope (Wang et al., 2010), Dowser (Haller et al., 2013), BORG (Neugschwandtner et al., 2015), MoWF (Pham et al., 2016) and Driller (Stephens et al., 2016) are representative examples. We will compare these fuzzing systems in terms of path explosion, search strategies, supported bug types, the need or lack thereof for source code, and offline or online execution.

5.2.1.1. *Path explosion.* The main hazard of symbolic execution is path explosion, which causes three problems. The first is that it occupies a large amount of memory space. The second is that it increases the complexity of constraint solving. The

System	search strategy	If vulnerabilities specific	Need source code	categories of symbolic executors
MoWF	searching the space where it prunes paths that are exercised by invalid, malformed inputs	YES	NO	online
KLEE	heuristic and random	NO	YES	online
Mayhem	heuristic	NO	NO	hybrid
S2E	heuristic	NO	NO	online
BORG	targeting buffer over-read bugs.	YES	NO	online
Taintscope	targetlng checksum parsing code	NO	NO	offline
Dowser	targeting buffer overflow and underflow vulnerabilities	YES	YES	online
SAGE	heuristic	NO	NO	offline
GWF	heuristic	NO	NO	offline
Driller	targeting the point where AFL is unable to identify inputs to search new paths	NO	NO	hybrid

Fig. 5 – Comparison Between Fuzzing Systems in Different Angles.

third is that it increases the number of constraint expressions resulting from execution on different paths. The existing systems have alleviated these three problems through different techniques.

To reduce memory usage, KLEE (Cadar et al., 2008), Mayhem (Cha et al., 2012), S2E (Chipounov et al., 2011) and BORG (Neugschwandtner et al., 2015) use State Merging (Avgerinos et al., 2014; Bugrara and Engler, 2013; Kuznetsov et al., 2012) to reduce the number of states in memory. Mayhem (Cha et al., 2012), BORG (Neugschwandtner et al., 2015) and SAGE (Godefroid et al., 2008) transfer the state from memory to disk.

To reduce the complexity of constraint solving, SAGE (Godefroid et al., 2008) and BORG (Neugschwandtner et al., 2015) use loop-guard pattern matching rules to identify a constraint that defines the number of iterations of input-dependent loops during dynamic symbolic execution, then set new constraints representing the pre- and post-loop conditions to summarize sets of executions of that loop. KLEE (Cadar et al., 2008) and Dowser (Haller et al., 2013) use several methods to simplify the constraint expressions, including expression rewriting, constraint set simplification, implied value concretization, constraint independence and a counter-example cache.

To decrease the number of constraint expressions, various fuzzing systems lead the symbolic execution to a certain target. TaintScope (Wang et al., 2010) identifies checksum fields in input instances and accurately locates checksum-based integrity checks by using a branch profiling technique, then alters the control flow to bypass such checks, which increase the number and complexity of symbolic expressions. GWF (Godefroid et al., 2008), BORG (Neugschwandtner et al., 2015) and MoWF (Pham et al., 2016) reduce constraint expressions by pruning from the search space any paths that are exercised by invalid inputs. Driller (Stephens et al., 2016) mitigates path explosion using concolic execution only to satisfy complex checks at the transition between compartments in the application. MoWF (Pham et al., 2016), BORG (Neugschwandtner et al., 2015) and Dowser (Haller et al., 2013) concentrate symbolic execution on code that might trigger special vulnerabilities.

5.2.1.2. *Other comparisons*. In addition to the techniques of dealing with path explosion, we also compare fuzzing systems in terms of their search strategy, whether they focus on specified vulnerabilities, whether they need source code or not, and their categories of symbolic executors.

The search strategy is the way the symbolic execution is guided. Mayhem (Cha et al., 2012), S2E (Chipounov et al., 2011), SAGE (Godefroid et al., 2008) and GWF (Godefroid et al., 2008) adopt heuristic strategies to achieve high code coverage, and KLEE (Cadar et al., 2008) adds a random strategy. BORG (Neugschwandtner et al., 2015) and Dowser (Haller et al., 2013) guide the execution to suspected vulnerability locations that have been statically analysed before execution. The symbolic execution of TaintScope (Wang et al., 2010) focuses on the checksum code. MoWF (Pham et al., 2016) searches the space in which it prunes paths that are exercised by invalid, malformed inputs. Driller (Stephens et al., 2016) targets the point where AFL (Zalewski, 2016) is unable to identify inputs to search new paths.

MoWF (Pham et al., 2016), BORG (Neugschwandtner et al., 2015) and Dowser (Haller et al., 2013) can only find the specified type of vulnerability, whereas other systems can detect all the vulnerabilities that crash programs. MoWF (Pham et al., 2016) aims at divide-by-zero and null-pointer dereference vulnerabilities, whereas BORG (Neugschwandtner et al., 2015) aims at overread vulnerabilities, and Dowser (Haller et al., 2013) aims at buffer overflow vulnerabilities.

Both KLEE (Cadar et al., 2008) and Dowser (Haller et al., 2013) need source code because they are based on the LLVM (Lattner and Adve, 2004) framework, which is used to compile source code into intermediate codes (Lattner and Adve, 2004) and perform static analysis on these intermediate codes, while other systems can test binary programs.

Dynamic symbolic execution (Cadar et al., 2008; Cha et al., 2012; Chipounov et al., 2011; Godefroid et al., 2008; Haller et al., 2013; Neugschwandtner et al., 2015; Pham et al., 2016; Wang et al., 2010) can be divided into three main categories (Cha et al., 2012): offline execution, which concretely runs a single execution path and then symbolically executes it; online execution, which attempts to execute all possible paths in a single run; and hybrid symbolic execution, which alternates between online and offline executions. KLEE (Cadar et al., 2008), MoWF (Pham et al., 2016), S2E (Chipounov et al., 2011), BORG (Neugschwandtner et al., 2015) and Dowser (Haller et al., 2013) adopt an online mode. TaintScope (Wang et al., 2010), SAGE (Godefroid et al., 2008) and GWF (Godefroid et al., 2008) adopt an offline mode. Mayhem (Cha et al., 2012) and Driller (Stephens et al., 2016) adopt a hybrid mode.

5.2.2. *Coverage feedback*

AFL (Zalewski, 2016), AFLFast (Böhme et al., 2016), VUzzer (Rawat et al., 2017), Skyfire (Wang et al., 2017), AFLGo (Böhme et al., 2017) and Steelix (Li et al., 2017) use coverage feedback. AFL (Zalewski, 2016) uses a code instrumentation technique (Bhansali et al., 2006; Luk et al., 2005; Nethercote and Seward, 2007; Bruening, 2010; Paradyn, 2017) to identify inputs that proceed through new paths in program execution. These inputs are used as quality seeds for the next round of fuzzing. This approach can significantly improve the path coverage and the probability of finding vulnerabilities.

AFLGo (Böhme et al., 2017), AFLFast (Böhme et al., 2016), Steelix (Li et al., 2017) and Skyfire (Wang et al., 2017) and VUzzer (Rawat et al., 2017) are based on AFL (Zalewski, 2016). AFLGo (Böhme et al., 2017) guides the fuzzing to the locations identified by setting a changed statement in patch testing. Steelix (Li et al., 2017) uses lightweight static analysis (Kinder et al., 2009; Sparks et al., 2007; Tip, 1994; Weiser, 1979, 1981, 1982) and instrumentation (Bhansali et al., 2006; Luk et al., 2005; Nethercote and Seward, 2007; Bruening, 2010; Paradyn, 2017) to add comparative progress information in feedback. This information helps the fuzzer obtain the location of magic bytes in the test inputs and perform mutations to match the magic bytes efficiently. AFLFast (Böhme et al., 2016) uses a power scheduling strategy, which is based on a Markov chain model (Norris, 1998), to assign the most fuzzing energy to paths in low-density regions.

Skyfire (Wang et al., 2017) improves the efficiency of fuzzing by providing high-quality samples to AFL (Zalewski, 2016). It learns a PCSG from a vast number of existing samples, then leverages the learned grammar to generate well-distributed input. VUzzer (Rawat et al., 2017) generates samples more

intelligently by adding control flow (Kinder et al., 2009; Sparks et al., 2007) and data flow features (Tip, 1994; Weiser, 1979, 1981, 1982) to the feedback to help discover deeply rooted bugs. Control flow (Kinder et al., 2009; Sparks et al., 2007) is used to prioritize and deprioritize certain paths by assigning weights to basic blocks. Data flow (Tip, 1994; Weiser, 1979, 1981, 1982) is used to infer the relation between the structural properties of the input and the branch conditions.

5.2.3. *Grammar representation*

SPIKE (Bradshaw, 2010), Peach (Peachtec, 2017), Skyfire (Wang et al., 2017) and GWF (Godefroid et al., 2008) use grammar representation to generate inputs with complex format in order to discover deeply rooted bugs, but the implementation methods are different. SPIKE (Bradshaw, 2010) provides different APIs to specify input data with complex format as nested data blocks that satisfy the relations between different fields.

Peach (Peachtec, 2017) and MoWF (Pham et al., 2016) specify input data by constructing a series of XML files, called Peach Pit, which are written manually. Peach Pit can describe the types of data chunks and fields and the relationships between them. Peach (Peachtec, 2017) uses Peach Pit to mutate the seed file or directly generate the test input, while MoWF (Pham et al., 2016) uses Peach Pit to explore the branches that are exercised depending on the presence of specific chunks.

Skyfire (Wang et al., 2017) does not need to use APIs or write XML files manually to obtain the input format. It can automatically generate format information by learning from a large number of sample files. GWF (Godefroid et al., 2008) describes the format information as grammar-based constraints that are generated directly by symbolic execution. SPIKE (Bradshaw, 2010), Peach (Peachtec, 2017) and GWF (Godefroid et al., 2008) require domain knowledge of the input format, but Skyfire (Wang et al., 2017) does not.

5.2.4. *Taint analysis*

BUZZFUZZ (Ganesh et al., 2009), TaintScope (Wang et al., 2010), Dowser (Haller et al., 2013), BORG (Neugschwandtner et al., 2015), SYMFUZZ (Cha et al., 2015) and VUzzer (Rawat et al., 2017) all use taint analysis. BUZZFUZZ (Ganesh et al., 2009) and SYMFUZZ (Cha et al., 2015) combine taint analysis with random variation. BUZZFUZZ (Ganesh et al., 2009) uses taint analysis to locate input regions that influence values used at key program attack points and then mutates the data in these identified regions of the original inputs. SYMFUZZ (Cha et al., 2015) automatically finds an optimal mutation ratio for mutational fuzzing based on the input-bit dependence inference, which is obtained by symbolic analysis. SYMFUZZ (Cha et al., 2015) uses taint analysis to reduce the cost of symbolic analysis on each basic block.

TaintScope (Wang et al., 2010), Dowser (Haller et al., 2013) and BORG (Neugschwandtner et al., 2015) combine taint analysis with dynamic symbolic execution. TaintScope (Wang et al., 2010) uses taint analysis to identify which bytes in an input are used in security-sensitive operations and then modifies those bytes. Dowser (Haller et al., 2013) and BORG (Neugschwandtner et al., 2015) perform dynamic taint analysis to reduce the amount of symbolic input. They identify which parts of the input influence memory access in the target location and then symbolize these parts for dynamic symbolic execution. Both find only specific categories of bug. Dowser (Haller et al., 2013) aims at buffer overflow vulnerabilities, and BORG (Neugschwandtner et al., 2015) aims at buffer overread vulnerabilities.

VUzzer (Rawat et al., 2017) performs dynamic taint analysis to obtain information on which offsets in the input are used at several branch conditions and what values are used as branch constraints. This information is added to the feedback sent to the mutator to generate new inputs.

6. Future directions

In this section, we will discuss the possible future research directions of fuzzing techniques on the basis of analysing different fuzzing techniques and fuzzing systems. We will discuss the application of different techniques in fuzzing systems in terms of three aspects: techniques used in fuzzing systems, techniques newly introduced into fuzzing systems, and techniques not used in fuzzing systems.

6.1. *Techniques used in fuzzing systems*

We can see from Sections 4.3 and 4.4 that the development of fuzzing techniques has entered an era of combining different techniques. With analysis of the advantages and disadvantages of the existing techniques, different combinations of techniques may provide new ideas for improving the effectiveness of fuzzing in the future. For example, we can reduce the number of fields to be symbolized and determine where to perform symbolic execution (Cadar et al., 2008; Cha et al., 2012; Chipounov et al., 2011; Godefroid et al., 2008) by incorporating grammar representation (Banks et al., 2006; Röning et al., 2002; Peachtec, 2017; Bradshaw, 2010; Fitblip, 2016). First, we symbolize all the fields in the input data, perform symbolic execution (Cadar et al., 2008; Cha et al., 2012; Chipounov et al., 2011; Godefroid et al., 2008), and analyse certain special fields, such as checksum fields. In the symbolic trace, we mark the code position where the special field is used for the last time. Then, we begin the second symbolic execution at the marked position without symbolizing the special field, generate partial data from symbolic expressions using the SMT solver (Bjørner and Phan, 2014; De Moura and Bjørner, 2008; Ganesh and Dill, 2007), and compute the special fields with the help of grammar representation techniques (Banks et al., 2006; Röning et al., 2002; Peachtec, 2017; Bradshaw, 2010; Fitblip, 2016). This method can alleviate the problem of path explosion in the symbolic execution by reducing the symbolic expressions produced by calculating checksum data and other complicated computation, so it can improve code coverage because the samples can access the deeper code hidden behind the checksum code.

6.2. *Techniques newly introduced into fuzzing systems*

Currently, machine learning (Dietterich, 1997) has penetrated various fields and shows extraordinarily good results. Papers in the last two years describe the attempts of researchers to combine machine learning with fuzzing techniques (Godefroid

et al., 2017). This trend will continue in the future. For example, machine learning can be used to identify the potentially dangerous paths and sections in the program by learning from paths and sections in many programs that hide vulnerabilities or in a program into which we add some vulnerabilities using tools, e.g., LAVA (Dolan-Gavitt et al., 2016), or to help choose a mutation strategy after learning the effects of using different mutation strategy before. These mutation strategies may include which seed to choose, which section in the seed to mutate, and the strategy by which to mutate these sections. Deep integration between fuzzing techniques and machine learning will increase the efficiency of fuzzing and even lead to a breakthrough in this field.

6.3. *Techniques not used in fuzzing systems*

Some techniques not currently used in fuzzing systems might nevertheless improve the fuzzing efficiency, for example, PSO (particle swarm optimization) (Kennedy, 2011). Solving a large number of constraint expressions often presents difficulty in symbolic execution. The symbolic expressions can be considered as a set of equations, and the solution to the equations is the input data that can be executed along the path that generates the symbolic expressions. A set of input data can represent a particle, and the solution of all equations is expressed as the optimal problem in PSO (Kennedy, 2011). PSO (Kennedy, 2011) can quickly approximate the optimal solution through mutual learning between particles. Thus, the problem of symbolic resolution can be transformed into a problem of finding the optimal solution by PSO (Kennedy, 2011).

7. Conclusion

Fuzzing efficiency has greatly improved in the past 20 years. It has developed from the original black-box fuzzing to white-box fuzzing and grey-box fuzzing, from mutational fuzzing to generational fuzzing, and from no-feedback fuzzing to feedback fuzzing. Grammar-based fuzzing eases the difficulties in fuzzing and digs out deep-seated vulnerabilities to some degree. Dynamic symbolic execution transforms the process of fuzzing into a mathematical problem. Taint analysis allows the fuzzing process to provide an accurate impact of the input data on the target program. Static analysis provides an excellent global view for fuzzing. Path coverage-based fuzzing greatly improves the efficiency of fuzzing. Machine learning now introduces a new method to fuzzing. With the continuous deeper understanding of the fuzzing process and the effective use of existing techniques, fuzzing techniques will provide improved technical support for software security.

Acknowledgements

This work is supported by the National Natural Science Foundation of China (No. 61502536).

REFERENCES

Avgerinos T, Rebert A, Cha SK, Brumley D. Enhancing symbolic execution with veritesting. In: Proceedings of the 36th international conference on software engineering. ACM; 2014. p. 1083–94.
Babić D, Martignoni L, McCamant S, Song D. Statically-directed dynamic automated test generation. In: Proceedings of the 2011 international symposium on software testing and analysis. ACM; 2011. p. 12–22.
Banks G, Cova M, Felmetsger V, Almeroth K, Kemmerer R, Vigna G. Snooze: toward a stateful network protocol fuzzer. In: ISC, vol. 4176. Springer; 2006. p. 343–58.
Barlow HB. Unsupervised learning. John Wiley & Sons, Ltd; 1989.
Bastani O, Sharma R, Aiken A, Liang P. Synthesizing program input grammars. In: Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation. ACM; 2017. p. 95–110.
Berry DA, Fristedt B. Bandit problems: sequential allocation of experiments (monographs on statistics and applied probability), vol. 12. Springer; 1985.
Bhansali S, Chen WK, Jong SD, Edwards A, Murray R, Chau J. Framework for instruction-level tracing and analysis of program executions. In: International conference on virtual execution environments. 2006. p. 154–63.
Bhatia S, Singh R. Automated correction for syntax errors in programming assignments using recurrent neural networks; 2016. arXiv preprint arXiv:160306129.
Bhatnagar S, Prasad H, Prashanth L. Reinforcement learning. Springer Int 1998;11(5):126–34.
Bjørner N, Phan AD. *νz*-maximal satisfaction with z3. SCSS 2014;30:1–9.
Bohler S. Lighttpd; 2017. Available from: http://www.lighttpd.net. [Accessed 6 October 2017].
Böhme M, Oliveira B, Roychoudhury A. Regression tests to expose change interaction errors. In: Proceedings of the 2013 9th joint meeting on foundations of software engineering. ACM; 2013. p. 334–44.
Böhme M, Pham VT, Nguyen MD, Roychoudhury A. Directed greybox fuzzing. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS17). 2017.
Böhme M, Pham VT, Roychoudhury A. Coverage-based greybox fuzzing as Markov chain. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security. ACM; 2016. p. 1032–43.
Bossert G. Netzob; 2017. Available from: https://www.netzob.org/. [Accessed 6 October 2017].
Bradshaw S. SPIKE fuzzer platform; 2010. Available from: http://resources.infosecinstitute.com/fuzzer-automation-with-spike/. [Accessed 6 October 2017].
Bruening D. Dynamorio; 2017. Available from: http://www.dynamorio.org. [Accessed 6 October 2017].
Bsharet. Kitty; 2017. Available from: https://github.com/cisco-sas/kitty. [Accessed 6 October 2017].
Bucur S. Improving scalability of symbolic execution for software with complex environment interfaces [Ph.D. thesis], École Polytechnique Fédérale de Lausanne; 2015.
Bugrara S, Engler D. Redundant state detection for dynamic symbolic execution. In: Proceedings of the 2013 USENIX conference on annual technical conference. USENIX Association; 2013. p. 199–212.
Bunel RR, Desmaison A, Mudigonda PK, Kohli P, Torr P. Adaptive neural compilation. Adv Neural Inf Process Syst 2016;1444–52.
Cadar C, Dunbar D, Engler DR. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI, vol. 8. 2008. p. 209–24.

Cadar C, Ganesh V, Pawlowski PM, Dill DL, Engler DR. EXE: automatically generating inputs of death. ACM Trans Inform Syst Secur 2008;12(2):10.
Campana G. Fuzzgrind: un outil de fuzzing automatique. SSTIC 2009;213–29.
Caruana R, Niculescu-Mizil A. An empirical comparison of supervised learning algorithms. In: International conference on machine learning. 2006. p. 161–8.
Caselden D, Bazhanyuk A, Payer M, Szekeres L, McCamant S, Song D. Transformation-aware exploit generation using a HI-CFG. Tech. Rep.; California Univ Berkeley Dept of Electrical Engineering and Computer Science; 2013.
Cha SK. OFuzz; 2017. Available from: https://github.com/sangkilc/ofuzz. [Accessed 6 October 2017].
Cha SK, Avgerinos T, Rebert A, Brumley D. Unleashing mayhem on binary code. In: Security and Privacy (SP), 2012 IEEE symposium on. IEEE; 2012. p. 380–94.
Cha SK, Woo M, Brumley D. Program-adaptive mutational fuzzing. In: Security and Privacy (SP), 2015 IEEE symposium on. IEEE; 2015. p. 725–41.
Chen Y, Groce A, Zhang C, Wong WK, Fern X, Eide E, et al. Taming compiler fuzzers. In: ACM SIGPLAN notices, vol. 48. ACM; 2013. p. 197–208.
Chipounov V, Kuznetsov V, Candea G. S2e: a platform for in-vivo multi-path analysis of software systems. ACM SIGPLAN Notices 2011;46(3):265–78.
Cho K, Van Merriënboer B, Gulcehre C, Bahdanau D, Bougares F, Schwenk H, et al. Learning phrase representations using RNN encoder-decoder for statistical machine translation; 2014. arXiv preprint arXiv:14061078.
Choren M. BackFuzz; 2017. Available from: https://github.com/localh0t/backfuzz. [Accessed 6 October 2017].
CIFASIS. QuickFuzz; 2017. Available from: http://quickfuzz.org/. [Accessed 6 October 2017].
Cui W, Peinado M, Chen K, Wang HJ, Irun-Briz L. Tupni: automatic reverse engineering of input formats. In: Proceedings of the 15th ACM conference on computer and communications security. ACM; 2008. p. 391–402.
Daniel. Dizzy; 2017. Available from: https://github.com/ernw/dizzy. [Accessed 6 October 2017].
DARPA. Cyber grand challenge; 2016. Available from: https://www.darpa.mil/program/cyber-grand-challenge. [Accessed 6 October 2017].
De Moura L, Bjørner N. Z3: an efficient SMT solver. Tools Algorithms Constr Anal Syst 2008;337–40.
DeMott J. Understanding how fuzzing relates to a vulnerability like heartbleed; 2014. Available from: http://labs.bromium.com/2014/05/14/understanding-how-fuzzing-relates-to-a-vulnerability-like-heartbleed/. [Accessed 6 October 2017].
Dietterich TG. Machine-learning research. AI Mag 1997;18(4):97–136.
Dolan-Gavitt B, Hulin P, Kirda E, Leek T, Mambretti A, Robertson W, et al. Lava: large-scale automated vulnerability addition. In: Security and Privacy (SP), 2016 IEEE symposium on. IEEE; 2016. p. 110–21.
Drewry W, Ormandy T. Flayer: exposing application internals. WOOT 2007;7:1–9.
El-Sherei S. Wadi; 2017. Available from: https://github.com/sensepost/wadi/. [Accessed 6 October 2017].
Ferrante J, Ottenstein KJ, Warren JD. The program dependence graph and its use in optimization. ACM Trans Program Lang Syst 1987;9(3):319–49.
Fewer S. Grinder; 2017. Available from: https://github.com/stephenfewer/grinder. [Accessed 6 October 2017].
Fitblip. Suley fuzzer; 2016. Available from: https://github.com/OpenRCE/sulley. [Accessed 6 October 2017].
Forrest S, Hofmeyr SA, Somayaji A, Longstaff TA. A sense of self for unix processes. In: Proceedings of the 1996 IEEE symposium on security and privacy, 1996. IEEE; 1996. p. 120–8.
Francis P, Leon D, Minch M, Podgurski A. Tree-based methods for classifying software failures. In: Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on. IEEE; 2004. p. 451–62.
Ganesh V, Dill DL. A decision procedure for bit-vectors and arrays. In: CAV, vol. 4590. Springer; 2007. p. 519–31.
Ganesh V, Leek T, Rinard M. Taint-based directed whitebox fuzzing. In: Proceedings of the 31st international conference on software engineering. IEEE Computer Society; 2009. p. 474–84.
Gascon H. Pulsar. Available from: https://github.com/hgascon/pulsar. [Accessed 6 October 2017].
Gegick M, Williams L, Osborne J, Vouk M. Prioritizing software security fortification throughcode-level metrics. In: Proceedings of the 4th ACM workshop on quality of protection. ACM; 2008. p. 31–8.
Godefroid P, Kiezun A, Levin MY. Grammar-based whitebox fuzzing. In: ACM SIGPLAN notices, vol. 43. ACM; 2008. p. 206–15.
Godefroid P, Klarlund N, Sen K. Dart: directed automated random testing. In: ACM SIGPLAN notices, vol. 40. ACM; 2005. p. 213–23.
Godefroid P, Levin MY, Molnar DA. Automated whitebox fuzz testing. In: NDSS, vol. 8. 2008. p. 151–66.
Godefroid P, Peleg H, Singh R. Learn&fuzz: machine learning for input fuzzing; 2017. arXiv preprint arXiv:170107232.
Google. IOCTL; 2017. Available from: https://code.google.com/p/ioctlfuzzer/. [Accessed 6 October 2017].
Google. Honggfuzz; 2017a. Available from: https://github.com/google/honggfuzz. [Accessed 6 October 2017].
Google. Syzkaller; 2017b. Available from: https://github.com/google/syzkaller. [Accessed 6 October 2017].
Google. Cluster fuzz; 2017c. Available from: https://github.com/google/oss-fuzz/blob/master/docs/clusterfuzz.md. [Accessed 6 October 2017].
Grieco G, Grinblat GL, Uzal L, Rawat S, Feist J, Mounier L. Toward large-scale vulnerability discovery using machine learning. In: Proceedings of the sixth ACM conference on data and application security and privacy. ACM; 2016. p. 85–96.
Gupta R, Pal S, Kanade A, Shevade S. Deepfix: fixing common c language errors by deep learning. In: AAAI. 2017. p. 1345–51.
Haller I, Slowinska A, Neugschwandtner M, Bos H. Dowsing for overflows: a guided fuzzer to find buffer boundary violations. In: USENIX security symposium. 2013. p. 49–64.
Halperin D, Heydt-Benjamin TS, Ransford B, Clark SS. Pacemakers and implantable cardiac defibrillators: software radio attacks and zero-power defenses. In: Proceedings – IEEE symposium on security and privacy. 2008. p. 129–42.
Hanford KV. Automatic generation of test cases. IBM Syst J 1970;9(4):242–57.
Hastings R, Joyce B. Purify: fast detection of memory leaks and access errors. In: Proceedings of the Winter 1992 USENIX conference. Citeseer; 1991.
Helin A. Radamsa; 2017. Available from: https://github.com/aoh/radamsa. [Accessed 6 October 2017].
Hex-Rays SA. Ida; 2017. Available from: https://www.hex-rays.com/products/ida/. [Accessed 6 October 2017].
Hocevar S. Zzuf; 2010. Available from: http://caca.zoy.org/wiki/zzuf/. [Accessed 6 October 2017].
Holler C, Herzig K, Zeller A. Fuzzing with code fragments. In: USENIX security symposium. 2012. p. 445–58.
Höschele M, Zeller A. Mining input grammars from dynamic taints. In: Proceedings of the 31st IEEE/ACM international conference on automated software engineering. ACM; 2016. p. 720–5.

Householder A. FOE; 2012. Available from: https://insights.sei.cmu.edu/cert/2012/07/cert-failure-observation-engine-20-released.html. [Accessed 6 October 2017].
Householder A. Dranzer; 2017. Available from: https://github.com/CERTCC-Vulnerability-Analysis/dranzer. [Accessed 6 October 2017].
Householder AD, Foote JM. Probability-based parameter selection for black-box fuzz testing. Tech. Rep.; Carnegie-Mellon Univ Pittsburgh PA Software Engineering INST; 2012.
ImageMagick Studio LLC. Imagemagick; 2017. Available from: https://www.imagemagick.org. [Accessed 6 October 2017].
Jin W, Orso A. Bugredux: reproducing field failures for in-house debugging. In: Software Engineering (ICSE), 2012 34th International Conference on. IEEE; 2012. p. 474–84.
Jones D. Trinity; 2017. Available from: https://github.com/kernelslacker/trinity. [Accessed 6 October 2017].
Kang MG, McCamant S, Poosankam P, Song D. Dta++: dynamic taint analysis with targeted control-flow propagation. In: NDSS. 2011.
Kargén U, Shahmehri N. Turning programs against each other: high coverage fuzz-testing using binary-code mutation and dynamic slicing. In: Proceedings of the 2015 10th joint meeting on foundations of software engineering. ACM; 2015. p. 782–92.
Kennedy J. Particle swarm optimization. In: Encyclopedia of machine learning. Springer; 2011. p. 760–6.
Kettunen A. Nodefuzz; 2017. Available from: https://github.com/attekett/NodeFuzz. [Accessed 6 October 2017].
Khanduri P. Diffy; 2017. Available from: https://github.com/twitter/diffy. [Accessed 6 October 2017].
Kifetew FM, Tiella R, Tonella P. Generating valid grammar-based test inputs by means of genetic programming and annotated grammars. Empirical Softw Eng 2017;22(2):928–61.
Kim SY, Cha S, Bae DH. Automatic and lightweight grammar generation for fuzz testing. Comput Secur 2013;36:1–11.
Kinder J, Zuleger F, Veith H. An abstract interpretation-based framework for control flow reconstruction from binaries. In: International workshop on verification, model checking, and abstract interpretation. Springer; 2009. p. 214–28.
Kirkpatrick S, Gelatt CD, Vecchi MP. Optimization by simulated annealing. Science 1983;220(4598):671–80.
Koprinkova-Hristova P, Mladenov V, Kasabov NK. Artificial neural networks. Eur Urol 2015;40(1):245.
Koret J. Nightmare; 2017. Available from: https://github.com/joxeankoret/nightmare. [Accessed 6 October 2017].
Kuck DJ, Kuhn RH, Padua DA, Leasure B, Wolfe M. Dependence graphs and compiler optimizations. In: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on principles of programming languages. ACM; 1981. p. 207–18.
Kurach K, Andrychowicz M, Sutskever I. Neural random-access machines; 2015. arXiv preprint arXiv:151106392.
Kuznetsov V, Kinder J, Bucur S, Candea G. Efficient state merging in symbolic execution. ACM SIGPLAN Notices 2012;47(6):193–204.
Landi W. Undecidability of static analysis. ACM Lett Program Lang Syst 1992;1(4):323–37.
Lattner C, Adve V. LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of the international symposium on code generation and optimization: feedback-directed and runtime optimization. IEEE Computer Society; 2004. p. 75.
Lcamtuf. AFL fuzzing strategies; 2014. Available from: https://lcamtuf.blogspot.jp/2014/08/binary-fuzzing-strategies-what-works.html. [Accessed 6 October 2017].
Lcamtuf. AFL bugs; 2017. Available from: http://lcamtuf.coredump.cx/afl/. [Accessed 6 October 2017].
Li M. Passive Fuzz Framework OSX; 2017. Available from: https://github.com/SilverMoonSecurity/PassiveFuzzFrameworkOSX. [Accessed 6 October 2017].
Li Y, Chen B, Chandramohan M, Lin SW, Liu Y, Tiu A. Steelix: program-state based binary fuzzing. In: Proceedings of the 2017 11th joint meeting on foundations of software engineering. ACM; 2017. p. 627–37.
Lin Z, Zhang X. Deriving input syntactic structure from execution. In: Proceedings of the 16th ACM SIGSOFT international symposium on foundations of software engineering. ACM; 2008. p. 83–93.
Linguaglossa D. Pyjfuzz; 2017. Available from: https://github.com/mseclab/PyJFuzz. [Accessed 6 October 2017].
LLVM-admin team. Libfuzzer; 2017. Available from: http://llvm.org/docs/LibFuzzer.html. [Accessed 6 October 2017].
Luk CK, Cohn R, Muth R, Patil H, Klauser A, Lowney G, et al. Pin: building customized program analysis tools with dynamic instrumentation. In: ACM SIGPLAN notices, vol. 40. ACM; 2005. p. 190–200.
Majumdar R, Xu RG. Directed test generation using symbolic grammars. In: Proceedings of the twenty-second IEEE/ACM international conference on automated software engineering. ACM; 2007. p. 134–43.
Marinescu PD, Cadar C. KATCH: high-coverage testing of software patches. In: Proceedings of the 2013 9th joint meeting on foundations of software engineering. ACM; 2013. p. 235–45.
Martignoni L. Kemufuzzer; 2017. Available from: https://github.com/jrmuizel/kemufuzzer. [Accessed 6 October 2017].
Mendez X. Wfuzz; 2017. Available from: https://github.com/xmendez/wfuzz. [Accessed 6 October 2017].
Meyering J. Coreutils; 2017. Available from: http://www.gnu.org/software/coreutils/coreutils.html. [Accessed 6 October 2017].
Miller BP, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities. Commun ACM 1990;33(12):32–44.
Muench M, Stijohann J, Kargl F, Francillon A, Balzarotti D. What you corrupt is not what you crash: challenges in fuzzing embedded devices. In: Proceedings of the Network and Distributed System Security Symposium (NDSS). 2018.
MWR Labs. Kernel fuzzer; 2017. Available from: https://github.com/mwrlabs/KernelFuzzer. [Accessed 6 October 2017].
Nair R. Syntribos; 2017. Available from: https://github.com/openstack/syntribos. [Accessed 6 October 2017].
Nethercote N, Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In: ACM SIGPLAN notices, vol. 42. ACM; 2007. p. 89–100.
Neugschwandtner M, Milani Comparetti P, Haller I, Bos H. The BORG: nanoprobing binaries for buffer overreads. In: Proceedings of the 5th ACM conference on data and application security and privacy. ACM; 2015. p. 87–97.
Newsham. Triforce afl; 2017. Available from: https://github.com/nccgroup/TriforceAFL. [Accessed 6 October 2017].
Newsome J, Song D. Dynamic taint analysis: automatic detection, analysis, and signature generation of exploit attacks on commodity software. In: Proceedings of the 12th network and distributed systems security symposium. Citeseer; 2005.
Nguyen VH, Tran LMS. Predicting vulnerable software components with dependency graphs. In: Proceedings of the 6th international workshop on security measurements and metrics. ACM; 2010. p. 3.
Niedermayer M. FFmpeg developers; 2017. Available from: http://ffmpeg.org/. [Accessed 6 October 2017].
Norris J. Markov chains, Cambridge series in statistical and probabilistic mathematics. Cambridge University Press; 1998.
Paradyn. Dyninst; 2017. Available from: http://www.dyninst.org. [Accessed 6 October 2017].

Parisotto E, Mohamed A, Singh R, Li L, Zhou D, Kohli P. Neurosymbolic program synthesis. arXiv preprint arXiv:161101855 2016.
Peachtec. Datamodel; 2014a. Available from: http://community.peachfuzzer.com/v3/DataModeling.html. [Accessed 6 October 2017].
Peachtec. Statemodel; 2014b. Available from: http://community.peachfuzzer.com/v3/StateModel.html. [Accessed 6 October 2017].
Peachtec. Peach; 2017. Available from: http://www.peachfuzzer.com/products/peach-platform. [Accessed 6 October 2017].
Peet C. Js fuzz; 2017. Available from: https://github.com/connor4312/js-fuzz. [Accessed 6 October 2017].
Pham VT, Böhme M, Roychoudhury A. Model-based whitebox fuzzing for program binaries. In: Proceedings of the 31st IEEE/ACM international conference on automated software engineering. ACM; 2016. p. 543–53.
Pham VT, Ng WB, Rubinov K, Roychoudhury A. Hercules: reproducing crashes in real-world application binaries. In: Proceedings of the 37th international conference on software engineering, vol. 1. IEEE Press; 2015. p. 891–901.
Prado CG. Brundlefuzz; 2017. Available from: https://github.com/carlosgprado/BrundleFuzz. [Accessed 6 October 2017].
Pu Y, Narasimhan K, Solar-Lezama A, Barzilay R. sk_p: a neural program corrector for moocs. In: Companion proceedings of the 2016 ACM SIG-PLAN international conference on systems, programming, languages and applications: software for humanity. ACM; 2016. p. 39–40.
Purdom P. A sentence generator for testing parsers. BIT 1972;12(3):366–75.
Rawat S, Gulati VP, Pujari AK. A fast host-based intrusion detection system using rough set theory. In: Transactions on rough sets IV. Springer; 2005. p. 144–61.
Rawat S, Jain V, Kumar A, Cojocar L, Giuffrida C, Bos H. VUzzer: application-aware evolutionary fuzzing. In: Proceedings of the Network and Distributed System Security symposium (NDSS). 2017.
Reczey B. Libming; 2017. Available from: http://www.libming.org. [Accessed 6 October 2017].
Reed S, De Freitas N. Neural programmer-interpreters; 2015. arXiv preprint arXiv:151106279.
Reese B. LibXML; 2017. Available from: http://xmlsoft.org. [Accessed 6 October 2017].
Regehr J. Csmith; 2017. Available from: https://github.com/csmith-project/csmith. [Accessed 6 October 2017].
Röning J, Lasko M, Takanen A, Kaksonen R. Protos-systematic approach to eliminate software vulnerabilities; 2002. Invited presentation at Microsoft Research.
Rouf I, Miller R, Mustafa H, Taylor T, Oh S, Xu W, et al. Security and privacy vulnerabilities of in-car wireless networks: a tire pressure monitoring system case study. In: USENIX security symposium, Washington, DC, USA, August 11-13, 2010, proceedings. 2010. p. 323–38.
Santos I, Devesa J, Brezo F, Nieves J, Bringas PG. OPEM: a static-dynamic approach for machine-learning-based malware detection. In: International joint conference CISIS12-ICEUTE 12-SOCO 12 special sessions. Springer; 2013. p. 271–80.
Saperstein B. On the occurrence of n successes within n Bernoulli trials. Technometrics 1973;15(4):809–18.
Sen K, Marinov D, Agha G. CUTE: a concolic unit testing engine for C. In: ACM SIGSOFT software engineering notes, vol. 30. ACM; 2005. p. 263–72.
Serebryany K, Bruening D, Potapenko A, Vyukov D. AddressSanitizer: a fast address sanity checker. In: USENIX annual technical conference. 2012. p. 309–18.
Shaw ZA. Rfuzz; 2017. Available from: https://github.com/zedshaw/rfuzz. [Accessed 6 October 2017].
Shellphish. Angr; 2017. Available from: https://github.com/angr/angr. [Accessed 6 October 2017].
Shin Y, Williams L. An initial study on the use of execution complexity metrics as indicators of software vulnerabilities. In: Proceedings of the 7th international workshop on software engineering for secure systems. ACM; 2011. p. 1–7.
Shoshitaishvili Y, Wang R, Hauser C, Kruegel C, Vigna G. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In: NDSS. 2015.
Sidiroglou S, Giovanidis G, Keromytis AD. A dynamic mechanism for recovering from buffer overflow attacks. In: Information Security, international conference, ISC 2005, Singapore, September 20–23, 2005, proceeding. 2005. p. 1–15.
Sintra T. Filebuster; 2017. Available from: https://github.com/henshin/filebuster. [Accessed 6 October 2017].
Slowinska A, Stancescu T, Bos H. Body armor for binaries: preventing buffer overflows without recompilation. In: USENIX annual technical conference. 2012. p. 125–37.
Smith D. Go-Fuzz; 2017. Available from: https://github.com/google/gofuzz. [Accessed 6 October 2017].
Sparks S, Embleton S, Cunningham R, Zou C. Automated vulnerability analysis: leveraging control flow for evolutionary input crafting. ACSAC 2007;477–86.
Stallman RM, Pesch RH, Shebs S. Debugging with GDB; 2002.
Stephens N, Grosen J, Salls C, Dutcher A, Wang R, Corbetta J, et al. Driller: augmenting fuzzing through selective symbolic execution. In: NDSS, vol. 16. 2016. p. 1–16.
Sutskever I, Vinyals O, Le QV. Sequence to sequence learning with neural networks. In: Advances in neural information processing systems. 2014. p. 3104–12.
Taborn MP, Yuan JK. Method and apparatus for detecting underflow and overflow; 1996. Google Patents.
Talos C. FuzzFlow; 2017a. Available from: https://github.com/talos-vulndev/FuzzFlow. [Accessed 6 October 2017].
Talos C. Clamav; 2017b. Available from: http://www.clamav.net/. [Accessed 6 October 2017].
Tandon G, Chan PK. Learning rules from system call arguments and sequences for anomaly detection. Tech Rep 2003.
Team MSECMSS. !exploitable; 2013. Available from: http://msecdbg.codeplex.com. [Accessed 6 October 2017].
Tip F. A survey of program slicing techniques; 1994. Centrum voor Wiskunde en Informatica.
Trull J. Regex-fuzzer; 2010. Available from: https://cloudblogs.microsoft.com/microsoftsecure/2010/10/12/new-tool-sdl-regex-fuzzer/. [Accessed 6 October 2017].
Wang J, Chen B, Wei L, Liu Y. Skyfire: data-driven seed generation for fuzzing. In: 2017 IEEE symposium on security and privacy. IEEE; 2017. p. 579–94.
Wang J, Zhao M, Zeng Q, Wu D, Liu P. Risk assessment of buffer "heartbleed" over-read vulnerabilities. In: IEEE/IFIP international conference on dependable systems and networks. 2015. p. 555–62.
Wang T, Wei T, Gu G, Zou W. Taintscope: a checksum-aware directed fuzzing tool for automatic software vulnerability detection. In: Security and privacy (SP), 2010 IEEE symposium on. IEEE; 2010. p. 497–512.
Weiser M. Program slicing. In: Proceedings of the 5th international conference on software engineering. IEEE Press; 1981. p. 439–49.
Weiser M. Programmers use slices when debugging. Commun ACM 1982;25(7):446–52.
Weiser MD. Program slices: formal, psychological, and practical investigations of an automatic program abstraction method; 1979. University of Michigan.
Wikipedia. Definition of software bug; 2017a. Available from: https://en.wikipedia.org/wiki/Software_bug. [Accessed 6 October 2017].

Wikipedia. The three tenets of cyber security; 2017b. Available from: https://en.wikipedia.org/wiki/Vulnerability_(computing). [Accessed 6 October 2017].
Wikipedia. Definition of embedded system; 2017c. https://en.wikipedia.org/wiki/Embedded_system. [Accessed 6 October 2017].
Wolpert DH, Macready WG. No free lunch theorems for optimization. IEEE Trans Evolution Comput 1997;1(1):67–82.
Woo M, Cha SK, Gottlieb S, Brumley D. Scheduling black-box mutational fuzzing. In: Proceedings of the 2013 ACM SIGSAC conference on computer & communications security. ACM; 2013. p. 511–22.
Yamaguchi F, Lindner F, Rieck K. Vulnerability extrapolation: assisted discovery of vulnerabilities using machine learning. In: Proceedings of the 5th USENIX conference on offensive technologies. USENIX Association; 2011. p. 13.
Zaddach J, Bruno L, Francillon A, Balzarotti D. AVATAR: a framework to support dynamic security analysis of embedded systems firmwares. In: Network and Distributed System Security Symposium. 2014.
Zalewski M. American fuzzy lop; 2016. Available from: https://github.com/mirrorer/afl. [Accessed 6 October 2017].
Zimmermann T, Nagappan N, Williams L. Searching for a needle in a haystack: predicting security vulnerabilities for windows vista. In: Software Testing, verification and validation (ICST), 2010 third International Conference on. IEEE; 2010. p. 421–8.

Chen Chen is a doctoral student at the Beijing University of Posts and Telecommunications. Her research field focuses on software security detection and embedded system security detection.

Baojiang Cui is a doctor and a professor at the Beijing University of Posts and Telecommunications. His research field includes network and host security behaviour analysis, software security detection, analysis of web/software and operating system security defects, smart terminals and mobile internet security, and internet of things security.

Jinxin Ma holds a doctoral degree and is an associate research fellow at China Information Technology Security Evaluation Center. His research field focuses on information safety.

Runpu Wu holds a master's degree and is an associate research fellow at China Information Technology Security Evaluation Center. His research field focuses on information safety.

Jianchao Guo is a master's degree candidate of the Beijing University of Posts and Telecommunications. Her research field focuses on software security detection and embedded system security detection.

Wenqian Liu is a master's degree candidate of the Beijing University of Posts and Telecommunications. Her research field focuses onsoftware security detection and embedded system security detection.