Li *et al. Cybersecurity* (2018) 1:6
https://doi.org/10.1186/s42400-018-0002-y

Cybersecurity

SURVEY Open Access

Fuzzing: a survey

Jun Li, Bodong Zhao and Chao Zhang*

Abstract

Security vulnerability is one of the root causes of cyber-security threats. To discover vulnerabilities and fix them in advance, researchers have proposed several techniques, among which fuzzing is the most widely used one. In recent years, fuzzing solutions, like AFL, have made great improvements in vulnerability discovery. This paper presents a summary of the recent advances, analyzes how they improve the fuzzing process, and sheds light on future work in fuzzing. Firstly, we discuss the reason why fuzzing is popular, by comparing different commonly used vulnerability discovery techniques. Then we present an overview of fuzzing solutions, and discuss in detail one of the most popular type of fuzzing, i.e., coverage-based fuzzing. Then we present other techniques that could make fuzzing process smarter and more efficient. Finally, we show some applications of fuzzing, and discuss new trends of fuzzing and potential future directions.

Keywords: Vulnerability discovery, Software security, Fuzzing, Coverage-based fuzzing

Introduction

Vulnerabilities have become the root cause of threats towards cyberspace security. Defined in RFC 2828 (Shirey 2000), a vulnerability is a flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy. Attack on vulnerabilities, especially on zero day vulnerabilities, can result in serious damages. The WannaCry ransomware attack (Wikipedia and Wannacry ransomware attack 2017) outbroke in May 2017, which exploits a vulnerability in Server Message Block (SMB) protocol, is reported to have infected more than 230,000 computers in over 150 countries within one day. It has caused serious crisis management problems and huge losses to many industries, such as finance, energy and medical treatment.

Considering the serious damages caused by vulnerabilities, much effort has been devoted to vulnerability discovery techniques towards software and information systems. Techniques including static analysis, dynamic analysis, symbolic execution and fuzzing (Liu et al. 2012) are proposed. Compared with other techniques, fuzzing requires few knowledge of targets and could be easily scaled up to large applications, and thus has become the most popular vulnerability discovery solution, especially in the industry.

*Correspondence: chaoz@tsinghua.edu.cn
Tsinghua University, Beijing 100084, China

The concept of fuzzing was first proposed in 1990s (Wu et al. 2010). Though the concept stays fixed during decades of development, the way how fuzzing is performed has greatly evolved. However, years of actual practice reveals that fuzzing tends to find simple memory corruption bugs in the early stage and seems to cover very small part of target code. Besides, the randomness and blindness of fuzzing results in a low efficiency in finding bugs. Many solutions have been proposed to improve the effectiveness and efficiency of fuzzing.

The combination of feedback-driven fuzzing mode and genetic algorithms provides a more flexible and customizable fuzzing framework, and makes the fuzzing process more intelligent and efficient. With the landmark of AFL, feedback-driven fuzzing, especially coverage-guided fuzzing, has made great progress. Inspired by AFL, many efficient solutions or improvements are proposed recently. Fuzzing is much different from itself several years ago. Therefore, it's necessary to summarize recent works in fuzzing and shed lights on future works.

In this paper, we try to summarize the state-of-the-art fuzzing solution, and how they improve the effectiveness and efficiency of vulnerability discovery. Besides, we show how traditional techniques can help improving the effectiveness and efficiency of fuzzing, and make fuzzers smarter. Then, we give an overview of how state-of-the-art fuzzers detect vulnerabilities of different targets, including file format applications, kernels, and protocols.

At last, we try to point out new trends of how fuzzing technique develops.

The rest of the paper is organized as follows: "Background" section presents background knowledge on vulnerability discovery techniques, "Fuzzing" section gives a detailed introduction to fuzzing, including the basic concepts and key challenges of fuzzing. In "Coverage-based fuzzing" section, we introduce the coverage-based fuzzing and related state-of-the-art works. In "Techniques integrated in fuzzing" section we summarize that how other techniques could help improve fuzzing, and "Fuzzing towards different applications" section presents several applications of fuzzing. In "New trends of fuzzing" section, we discuss and summarize the possible new trends of fuzzing. And we conclude our paper in "Conclusion" section.

Background

In this section, we give a brief introduction to traditional vulnerability discovery techniques, including: static analysis, dynamic analysis, taint analysis, symbolic execution, and fuzzing. Then we summarize the advantages and disadvantages of each technique.

Static analysis

Static analysis is the analysis of programs that is performed without actually executing the programs (Wichmann et al. 1995). Instead, static analysis is usually performed on the source code and sometimes on the object code as well. By analysis on the lexical, grammar, semantics features, and data flow analysis, model checking, static analysis could detect hiding bugs. The advantage of static analysis is the high detection speed. An analyst could quickly check the target code with a static analysis tool and perform the operation timely. However, static analysis endures a high false rate in practice. Due to the lack of easy to use vulnerability detection model, static analysis tools are prone to a large number of false positives. Thus identifying the results of static analysis remains a tough work.

Dynamic analysis

In contrast to static analysis, in dynamic analysis of programs, an analyst need to execute the target program in real systems or emulators (Wikipedia 2017). By monitoring the running states and analyzing the runtime knowledge, dynamic analysis tools can detect program bugs precisely. The advantage of dynamic analysis is the high accuracy while there exists the following disadvantages. First, debugging, analyzing and running of the target programs in dynamic analysis cause a heavy human involvement, and result in a low efficiency. Besides, the human involvement requires strong technical skills of analysts. In short, dynamic analysis has the shortcomings of slow speed, low efficiency, high requirements on the technical level of testers, poor scalability, and is difficult to carry out large-scale testing.

Symbolic execution

Symbolic execution (King 1976) is another vulnerability discovery technique that is considered to be very promising. By symbolizing the program inputs, the symbolic execution maintains a set of constraints for each execution path. After the execution, constraint solvers will be used to solve the constraint and determine what inputs cause the execution. Technically, symbolic execution could cover any execution path in a program and has shown good effect in tests of small programs, while there exists many limitations, either. First, the path explosion problem. As with the scale of program grows, the execution states explodes, which exceeds the solving ability of constraint solvers. Selective symbolic execution is proposed as a compromise. Second, the environment interactions. In symbolic execution, when target program execution interacts with components out of the symbolic execution environments, such as system calls, handling signals, etc., consistency problems may arise. Previous work has proved that symbolic execution is still difficult to scale up to large applications (Böhme et al. 2017).

Fuzzing

Fuzzing (Sutton et al. 2007) is currently the most popular vulnerability discovery technique. Fuzzing was first proposed by Barton Miller at the University of Wisconsin in 1990s. Conceptually, a fuzzing test starts with generating massive normal and abnormal inputs to target applications, and try to detect exceptions by feeding the generated inputs to the target applications and monitoring the execution states. Compared with other techniques, fuzzing is easy to deploy and of good extensibility and applicability, and could be performed with or without the source code. Besides, as the fuzzing test is performed in the real execution, it gains a high accuracy. What's more, fuzzing requires few knowledge of target applications and could be easily scaled up to large scale applications. Though fuzzing is faced with many disadvantages such as low efficiency and low code coverage, however, outweighed the bad ones, fuzzing has become the most effective and efficient state-of-the-art vulnerability discovery technique currently.

Table 1 shows the advantages and disadvantages of different techniques.

Fuzzing

In this section, we try to give a perspective on fuzzing, including the basic techniques background knowledge and challenges in improving fuzzing.

Table 1 Comparison of different techniques

Technique	Easy to start ?	Accuracy	Scalability
static analysis	easy	low	relatively good
dynamic analysis	hard	high	uncertain
symbolic execution	hard	high	bad
fuzzing	easy	high	good

Working process of fuzzing

Figure 1 depicts the main processes of traditional fuzzing tests. The working process is composed of four main stages, the testcase generation stage, testcase running stage, program execution state monitoring and analysis of exceptions.

A fuzzing test starts from the generation of a bunch of program inputs, i.e., testcases. The quality of generated testcases directly effects the test effects. The inputs should meet the requirement of tested programs for the input format as far as possible. While on the other hand, the inputs should be broken enough so that processing on these inputs would very likely to fail the program. According to the target programs, inputs could be files with different file formats, network communication data, executable binaries with specified characteristics, etc. How to generate broken enough testcases is a main challenge for fuzzers. Generally, two kind of generators are used in state-of-the-art fuzzers, generation based generators and mutation based generators.

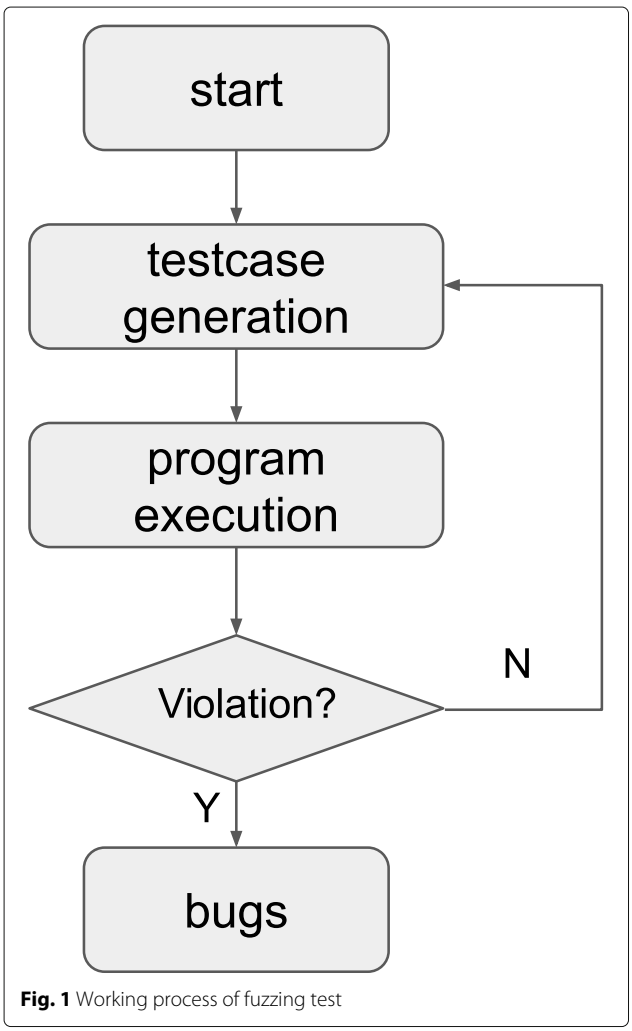

Fig. 1 Working process of fuzzing test

Testcases are fed to target programs after generated in the previous phase. Fuzzers automatically start and finish the target program process and drive the testcase handling process of target programs. Before the execution, analysts could configure the way the target programs start and finish, and predefine the parameters and environment variables. Usually, the fuzzing process stops at a predefined timeout, program execution hangs or crashes.

Fuzzers monitor the execution state during the execution of target programs, expecting exception and crashes. Common used exception monitoring methods includes monitoring on specific system signals, crashes, and other violations. For violations without intuitive program abnormal behaviors, lots of tools could be used, including AddressSanitizer (Serebryany et al. 2012), DataFlowsanitizer (The Clang Team 2017a), ThreadSanitizer (Serebryany and Iskhodzhanov 2009), LeakSanitizer (The Clang Team 2017b), etc. When violations are captured, fuzzers store the corresponding testcase for latter replay and analysis.

In the analyzing stage, analysts try to determine the location and root cause of captured violations. The analysis is often processed with the help of debuggers, like GDB, windbg, or other binary analysis tools, like IDA Pro, OllyDbg, etc. Binary instrumentation tools, like Pin (Luk et al. 2005), could also be used to monitor the exact execution state of collected testcases, such as the thread information, instructions, register information and so on. Automatically crash analysis is another important field of research.

Types of fuzzers

Fuzzers can be classified in various ways.

A fuzzer could be classified as generation based and mutation based (Van Sprundel 2005). For a generation based fuzzer, knowledge of program input is required. For file format fuzzing, usually a configuration file that predefines the file format is provided. Testcases are generated according to the configuration file. With given file format knowledge, testcases generated by generation based fuzzers are able to pass the validation of programs more easily and could be more likely to test the deeper code of target programs. However, without a friendly document, analyzing the file format is a tough work. Thus mutation based fuzzers are easier to start and more applicable, and widely used by state-of-the-art fuzzers. For mutation

based fuzzers, a set of valid initial inputs are required. Testcases are generated through the mutation of initial inputs and testcases generated during the fuzzing process. We compare generation based fuzzers and mutation based fuzzers in Table 2.

With respect to the dependence on program source code and the degree of program analysis, fuzzers could be classified as white box, gray box and black box. White box fuzzers are assumed to have access to the source code of programs, and thus more information could be collected through analysis on source code and how testcases affect the program running state. Black box fuzzers do fuzzing test without any knowledge on target program internals. Gray box fuzzers works without source code, either, and gain the internal information of target programs through program analysis. We list some common white box, gray box and black box fuzzers in Table 3.

According to the strategies of exploring the programs, fuzzers could be classified as directed fuzzing and coverage-based fuzzing. A directed fuzzer aims at generation of testcases that cover target code and target paths of programs, and a coverage-based fuzzer aims at generation of testcases that cover as much code of programs as possible. Directed fuzzers expect a faster test on programs, and coverage-based fuzzers expect a more thorough test and detect as more bugs as possible. For both directed fuzzers and coverage-based fuzzers, how to extract the information of executed paths is a key problem.

Fuzzers could be classified as dumb fuzz and smart fuzz according to whether there is a feedback between the monitoring of program execution state and testcase generation. Smart fuzzers adjustment the generation of testcases according to the collected information that how testcases affect the program behavior. For mutation based fuzzers, feedback information could be used to determine which part of testcases should be mutated and the way to mutate them. Dumb fuzzers acquires a better testing speed, while smart fuzzers generate better testcases and gain a better efficiency.

Table 2 Comparison of generation based fuzzers and mutation based fuzzers

	Easy to start ?	Priori knowledge	Coverage	Ability to pass validation
Generation based	hard	needed, hard to acquire	high	strong
Mutation based	easy	not needed	low, affected by initial inputs	weak

Table 3 Common white box, gray box and black box fuzzers

	White box fuzzers	Gray box fuzzers	Black box fuzzers
Generation based	SPIKE (Bowne 2015), Sulley (Amini 2017), Peach (PeachTech 2017)		
Mutation based	Miller (Takanen et al. 2008)	AFL (Zalewski 2017a), Driller (Stephens et al. 2016), Vuzzer (Rawat et al. 2017), TaintScope (Wang et al. 2010), Mayhem (Cha et al. 2012)	SAGE (Godefroid et al. 2012), Libfuzzer (libfuzzer 2017)

Key challenges in fuzzing

Traditional fuzzers usually utilize a random based fuzzing strategy in practice. Limitations of program analysis techniques result in a present situation that fuzzers are not smart enough. Thus fuzzing test still faces many challenges. We list some key challenges as follows.

The challenge of how to mutate seed inputs. Mutation based generation strategy is widely used by state-of-the-art fuzzers for its convenience and easy set up. However, how to mutate and generate testcases that capable to cover more program paths and easier to trigger bugs is a key challenge(Yang et al. 2007). Specifically, mutation based fuzzers need to answer two questions when do mutation: (1) where to mutate, and (2) how to mutate. Only mutation on a few key positions would affect the control flow of the execution. Thus how to locate these key positions in testcases is of great importance. Besides, the way fuzzers mutate the key positions is another key problem,i.e, how to determine the value that could direct the testing to interesting paths of programs. In short, blind mutation of testcases result in serious waste of testing resource and better mutation strategy could significantly improve the efficiency of fuzzing.

The challenge of low code coverage. Higher code coverage represents for a higher coverage of program execution states, and a more thorough testing. Previous work has proved that better coverage results in a higher probability of finding bugs. However, most testcases only cover the same few paths, while most of the code could not be reached. As a result, it's not a wise choice to achieve high coverage only through large amounts of testcase generation and throwing into testing resources. Coverage-based fuzzers try to solve the problem with the help of program analysis techniques, like program instrumentation. We will introduce the detail in next section.

The challenge of passing the validation. Programs often validate the inputs before parsing and handling. The validation works as a guard of programs, saving the computing resource and protecting the program against invalid inputs and damage caused by malicious constructed

inputs. Invalid testcases are always ignored or discarded. Magic numbers, magic strings, version number check, and checksums are common validations used in programs. Testcases generated by black box and gray box fuzzers are hard to pass the validation for a blind generation strategy, which results in quite low efficient fuzzing. Thus, how to pass the validation is another key challenge.

Various methods are proposed as countermeasures to these challenges, and both traditional techniques, like program instrumentation and taint analysis, and new techniques, like RNN and LSTM (Godefroid et al. 2017) (Rajpal et al. 2017) are involved. How these techniques can compromise the challenges will be discussed in "Techniques integrated in fuzzing" section.

Coverage-based fuzzing

Coverage-based fuzzing strategy is widely used by state-of-the-art fuzzers, and has proved to be quite effective and efficient. To achieve a deep and thorough program fuzzing, fuzzers should try to traverse as many program running states as possible. However, there doesn't exist a simple metric for program states, for the uncertainty of program behaviors. Besides, a good metric should be easily determined during the process running. Thus measuring the code coverage becomes an approximate alternative solution. Using such a scheme, increase of code coverage is representative of new program states. Besides, with both compiled-in and external instrumentation, code coverage could be easily measurable. However, we say code coverage is an approximate measurement, because in practice, a constant code coverage does not indicate a constant number of program states. There could be a certain loss of information using this metric. In this section, we take AFL as an example and shed light on coverage-based fuzzing.

Code coverage counting

In program analysis, the program is composed by basic blocks. Basic blocks are code snippets with a single entry and exit point, instructions in basic blocks will be sequentially executed and will only be executed once. In code coverage measuring, state-of-the-art methods take basic block as the best granularity. The reasons include that, (1) basic block is the smallest coherent units of program execution, (2) measuring function or instruction would result in information loss or redundancy, (3) basic block could be identified by the address of the first instruction and basic block information could be easily extracted through code instrumentation.

Currently, there are two basic measurement choices based on basic blocks, simply counting the executed basic blocks and counting the basic block transitions. In the latter method, programs are interpreted as a graph, and vertices are used to represent for the basic blocks, edges represent for the transition between basic blocks. The latter method record edges while the former one record vertices. While the experiment shows simply counting executed basic blocks would result in serious information loss. As shown in Fig. 2, if the program path (BB1, BB2, BB3, BB4) is firstly executed, and then path (BB1, BB2, BB4) is encountered by the execution, the new edge (BB2, BB4) information is lost.

AFL is the first to introduce the edge measurement method into coverage-based fuzzing. We take AFL as an example and show how coverage-based fuzzers gain coverage information during the fuzzing process. AFL gains the coverage information via lightweight program instrumentation. According to whether the source code is provided, AFL provides two instrumentation mode, the compile-in instrumentation and external instrumentation. In compile-in instrumentation mode, AFL provides both gcc mode and llvm mode, according to the compiler we used, which will instrument code snippet when binary is generated. In external mode, AFL provides qemu mode,

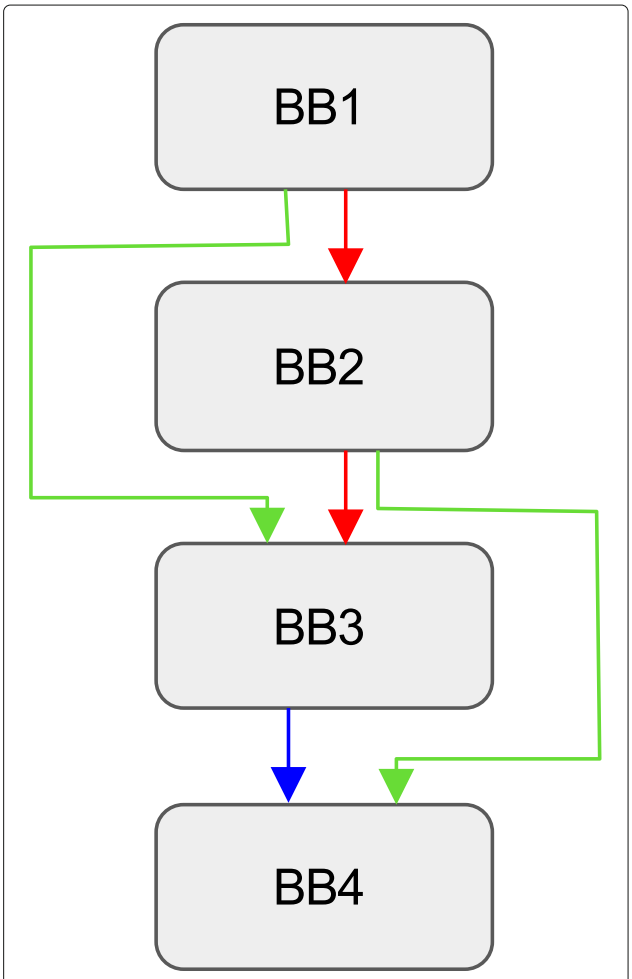

Fig. 2 A sample of BB transitions

which will instrument code snippet when basic block is translated to TCG blocks.

Listing 1 shows a sketch of instrumented code snippet (Zalewski 2017b). In instrumentation, a random ID, i.e., the variable cur_location is instrumented in basic blocks. The variable shared_mem array is a 64 KB shared memory region, each byte is mapped to a hit of a particular edge (BB_src, BB_dst). A hash number is computed when a basic block transition happens and the corresponding byte value in bitmap array will be update. Figure 3 depicts the mapping of hash and bitmap.

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]
++;
prev_location = cur_location >> 1;
```

Listing 1 AFL's instrumentation

Working process of coverage-based fuzzing

Algorithm 1 shows the general working process of a coverage-based fuzzer. The test starts from an initial given seed inputs. If the seed input set is not given, then the fuzzer constructs one itself. In the main fuzzing loop, the fuzzer repeatedly chooses an interesting seed for the following mutation and testcase generation. Target program is then driven to execute the generated testcases under the monitoring of fuzzer. Testcases that trigger crashes will be collected, and other interesting ones will be added to the seed pool. For a coverage-based fuzzing, testcases that reach new control flow edges are considered to be interesting. The main fuzzing loop stops at a pre-configured timeout or an abort signal.

During the process of fuzzing, fuzzers track the execution via various methods. Basically, fuzzers track the execution for two purposes, the code coverage and security violations. The code coverage information is used to pursue a thorough program state exploration, and the security violation tracking is for better bug finding. As detailed in the previous subsections, AFL tracks the code coverage through code instrumentation and AFL bitmap. Security violations tracking could be processed with the help of lots of sanitizers, such as AddressSanitizer (Serebryany et al. 2012), ThreadSanitizer (Serebryany and Iskhodzhanov 2009), LeakSanitizer (The Clang Team 2017b), etc.

Algorithm 1 Coverage-based Fuzzing

Input: Seed Inputs S

1: $T = S$
2: $Tx = \emptyset$
3: **if** $T = \emptyset$ **then**
4: $T.add(emptyfile)$
5: **end if**
6: **repeat**
7: $t = choose\_next(T)$
8: $s = assign\_energy(t)$
9: **for** i from 1 **to** s **do**
10: $t' = mutate(t)$
11: **if** t' *crashes* **then**
12: $Tx.add(t')$
13: **else if** $isInteresting(t')$ **then**
14: $T.add(t')$
15: **end if**
16: **end for**
17: **until** timeout or abort-signal

Output: Crashing Inputs Tx

Figure 4 shows the working process of AFL, a very representative coverage-based fuzzer. The target application is instrumented before execution for the coverage collection. As mentioned before, AFL supports both compile time instrumentation and external instrumentation, with gcc/llvm mode and qemu mode. An initial seed inputs should also be provided. In the main fuzzing loop, (1) the fuzzer selects a favorite seed from the seed pool according to the seed selection strategy, and AFL prefers the fastest and smallest ones. (2) seed files are mutated according to the mutation strategy, and a bunch of testcases are generated. AFL currently employs some random modifications and testcase splicing methods, including sequential bit flip

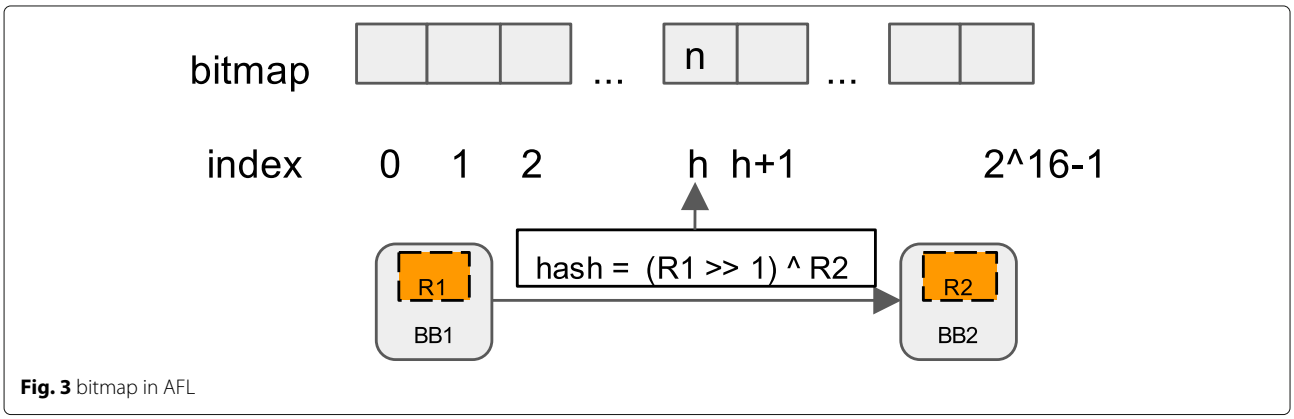

Fig. 3 bitmap in AFL

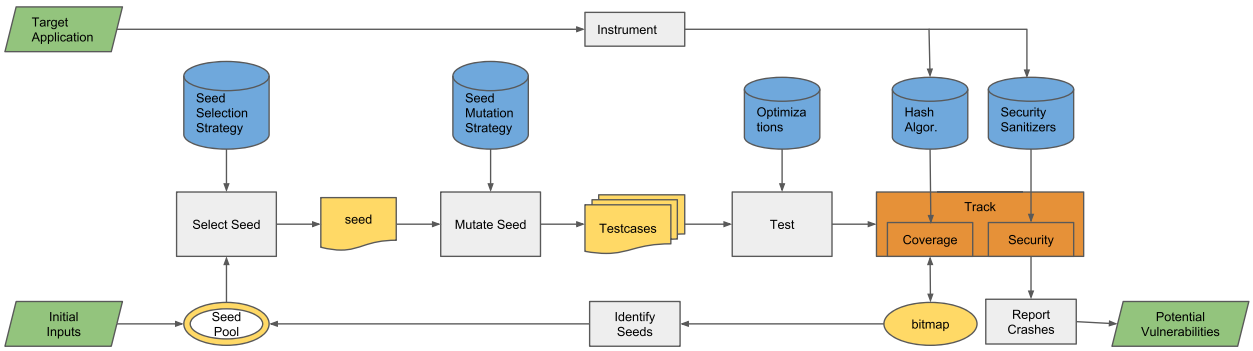

Fig. 4 Working process of AFL

with varying lengths and stepovers, sequential addition and subtraction of small integers and sequential insertion of known interesting integers like 0, 1, INT_MAX, etc. (Zalewski 2017b) (3) testcases are executed and the execution is under tracking. The coverage information is collected to determine interesting testcases, i.e. ones that reach new control flow edges. Interesting testcases are added to the seed pool for the next round run.

Key questions

Previous introduction indicates that lots of questions need to be solved to run an efficient and effective coverage-based fuzzing. Lots of explorations have been done around these questions. We summarize and list some state-of-the-art works in this subsection, as shown in Table 4.

A. How to get initial inputs? Most state-of-the-art coverage-based fuzzers employ a mutation based testcase generation strategy, which heavily depend on the quality of initial seed inputs. Good initial seed inputs can significantly improve the efficient and effectiveness of fuzzing. Specifically, (1) providing well format seed inputs could save lots of cpu times consumed by constructing one, (2) good initial inputs could meet the requirement for complicated file format, which are hard to guess in the mutation phase, (3) mutation based on well format seed input is more likely to generate testcases that could reach deeper and hard to reach paths, (4) good seed inputs could be reused during multiple test.

Common used methods of gathering seed inputs include using standard benchmarks, crawling from the Internet and using existing POC samples. Open source applications are usually released with a standard benchmark, which is free to use to test the projects. The provided benchmark is constructed according to the characteristics and functions of applications, which naturally construct a good set of seed inputs. Considering the diversity of target application inputs, crawling from the Internet is the most intuitive method. You can easily download files with certain formats. Besides, for some common used file formats, there are many open test projects on the network that provide free test data sets. Further more, using existing POC samples is also a good idea. However, too big quantity of seed inputs will result in a waste of time in the first dry run, thus bring another concern, how to distill the initial corpus. AFL provides a tool, which extracts a minimum set of inputs that achieve the same code coverage.

B. How to generate testcases? The quality of testcases is an important factor affecting the efficiency and effectiveness of fuzzing testing. Firstly, good testcases explore more program execution states and cover more code in a shorter time. Besides, good testcases could target potential vulnerable locations and bring a faster discovery of program bugs. Thus how to generate good testcases based on seed inputs is an important concern.

Rawat et al. (2017) proposed Vuzzer, an application aware grey box fuzzer that integrates with static and dynamic analysis. Mutation of seed inputs involves two key question: where to mutate and what value to use for the mutation. Specifically, Vuzzer extracts immediate values, magic values and other characteristic strings that

Table 4 Comparison of different techniques

Initial inputs get	Inputs mutation	Seed selection	Testing efficiency
Standard benchmarks;	Vuzzer (Rawat et al. 2017)	AFLFast (Böhme et al. 2017)	Forkserver (lcamtuf 2014)
Crawling from Internet;	Skyfire (Wang et al. 2017)	Vuzzer	Intel PT (Schumilo et al. 2017)
POC samples;	Learn & Fuzz (Godefroid et al. 2017)	AFLGo (2017)	Work (Xu et al. 2017)
	Faster Fuzzing (Nichols et al. 2017)	QTEP (Wang et al. 2017)	
	Work (Rajpal et al. 2017)	SlowFuzz (Petsios et al. 2017)	

affect the control flow via static analysis before the main fuzzing loop. During the program execution, Vuzzer utilize the dynamic taint analysis technique to collect information that affect the control flow branches, including specific value and the corresponding offset. By mutation with collected value and mutation at recognized locations, Vuzzer could generate testcases that are more likely to meet the branch judgment condition and pass magic value validations. However, Vuzzer still could not pass other types of validation in programs, like hash based checksum. Besides, Vuzzer's instrument, taint analysis, and main fuzzing loop is implemented based on Pin (Luk et al. 2005), which result in a relatively slow testing speed, compared to AFL.

Wang et al. (2017) proposed Skyfire, a data-driven seed generation solution.S kyfire learns a probabilistic context-sensitive grammar (PCSG) from crawled inputs, and leverages the learned knowledge in the generation of well-structured inputs. The experiment shows that testcases generated by Skyfire cover more code than those generated by AFL, and find more bugs. The work also proves that the quality of testcases is an important factor that affect the efficiency and effectiveness of fuzzing.

With the development and widely use of machine learning techniques, some research try to use machine learning techniques to assist the generation of testcases. Godefroid et al. (2017) from Microsoft Research use neural-network-based statistical machine-learning techniques to automatically generate testcases. Specifically, they firstly learn the input format from a bunch of valid inputs via machine learning techniques, and then leverage the learned knowledge guide the testcase generation. They present a fuzzing process on the PDF parser in Microsoft's Edge browser. Though the experiment didn't give an encouraging result, it's still a good attempt. Rajpal et al. (2017) from Microsoft use neural networks to learn from past fuzzing explorations and predict which byte to mutate in input files. Nichols et al. (2017) use the Generation Adversarial Network (GAN) models to help reinitializing the system with novel seed files. The experiment shows the GAN is faster and more effective than the LSTM, and helps discover more code paths.

C. How to select seed from the pool? Fuzzers repeatedly select seed from seed pool to mutate at the beginning of a new round test in the main fuzzing loop. How to select seed from the pool is another important open problem in fuzzing. Previous work has prove that good seed selection strategy could significantly improve the fuzzing efficiency and help find more bugs, faster (Rawat et al. 2017; Böhme et al. 2017, 2017; Wang et al. 2017). With good seed selection strategies, fuzzers could (1) prioritize seeds which are more helpful, including covering more code and be more likely to trigger vulnerabilities, (2) reduce the waste of repeatedly execution of paths and save computing resource, (3) optimally select seeds that cover deeper and more vulnerable code and help identifying hidden vulnerabilities faster. AFL prefers smaller and faster testcases to pursue a fast testing speed.

Böhme et al. (2017) proposed AFLFast, a coverage-based greybox fuzzer. They observe that most of the testcases concentrate on the same few paths. For instance, in a PNG processing program, most of the testcases generated through random mutation are invalid and trigger the error handling paths. AFLFast divide the paths into high-frequent ones and low-frequent ones. During the fuzzing process, AFLFast measures the frequency of executed paths, prioritize seeds that have been fuzzed fewer number of times and allocates more energy to seeds that exercise low-frequent paths.

Rawat et al. (2017) integrates static and dynamic analysis to identify hard-to-reach deeper paths, and prioritizes seeds that reach deeper paths. Vuzzer's seed selection strategy could help find vulnerabilities hidden in deep path.

AFLGo (Böhme et al. 2017) and QTEP (Wang et al. 2017) employ a directed selection strategy. AFLGo defines some vulnerable code as target locations, and optimally select testcase that are closer to target locations. Four types of vulnerable code are mentioned in the AFLGo paper, including patches, program crashes lack enough tracking information, result verified by static analysis tools and sensitive information related code snippets. With properly directed algorithm, AFLGo could allocate more testing resource on interesting code. QTEP leverage static code analysis to detect fault-prone source code and prioritize seeds that cover more faulty code. Both AFLGo and QTEP heavily depend on the effectiveness of static analysis tools. However, the false positive of current static analysis tools is still high and can't give an accurate verification.

Characteristics of known vulnerabilities could also be used in seed selection strategy. SlowFuzz (Petsios et al. 2017) aims at algorithmic complexity vulnerabilities, which often occurs with a significantly high computing resource consuming. Thus SlowFuzz prefers the seeds that consume more resources like cpu times and memory. However, gathering resource consuming information brings a heavy overhead and brings down the fuzzing efficient. For instance, to gather the cpu time, SlowFuzz counts the number of executed instructions. Besides, SlowFuzz requires a high accuracy of resource consuming information.

D. How to efficiently test applications? Target applications are repeatedly start up and finished by fuzzers in the main fuzzing loop. As we know, for fuzzing of user-land applications, creation and finishing of process will consume large amount of cpu time. Frequently create and finish the process will badly bring down the fuzzing

efficiency. As a result, lots of optimizations are done by previous work. Both tradition system features and new features are used in the optimization. AFL employs a forkserver method, which create an identical clone of the already-loaded program and reuse the clone for each single run. Besides, AFL also provide persistent mode, which helps to avoid the overhead of the notoriously slow execve() syscall and the linking process, and parallel mode, which help to parallelize the testing on multi-core systems. Intel's Processor Trace (PT) (James 2013) technology is used in kernel fuzzing to save the overhead brought by coverage tracking. Xu et al. (2017) aim at solving the performance bottlenecks of parallel fuzzing on multi-core machines. By designing and implementing three new operating primitives, they show that there work could significantly speed up state-of-the-art fuzzers, like AFL and LibFuzzer.

Techniques integrated in fuzzing

Modern applications often use very complex data structures and parsing on complex data structures are more likely to bring into vulnerabilities. Blind fuzzing strategies that use random mutation methods result in massive invalid testcases and low fuzzing efficiency. Currently state-of-the-art fuzzers generally employ a smart fuzzing strategy. Smart fuzzers collect program control flow and data flow information through program analysis techniques and consequently leverage the collected information to improve the generation of testcases. Testcases generated by smart fuzzers are better targeted, could be more likely to fulfill the programs' requirement for data structure and logical judgment. Figure 5 depicts a sketch of smart fuzzing. To build a smart fuzzer, a variety of techniques are integrated in fuzzing. As mentioned in previous sections, fuzzing in practice is facing lots of challenges. In this section, we try to summarize the techniques used by previous work and how these techniques compromise the challenges in fuzzing process.

We summarize the main techniques integrated in fuzzing in Table 5. For each technique, we list some of the representative work in the table. Both traditional techniques, including static analysis, taint analysis, code instrumentation and symbolic execution, and some relatively new techniques, like machine learning techniques, are used. We select two key phases in fuzzing, the testcase generation phase and program execution phase, and summarize how the integrated techniques improve fuzzing.

Testcase generation

As mentioned before, testcases in fuzzing are generated in generation based method or mutation based method. How to generate testcases that fulfill the requirement of complex data structure and more likely to trigger hard-to-reach paths is a key challenge. Previous work proposed a variety of countermeasures that integrated with different techniques.

In generation based fuzzing, the generator generate testcases according to the knowledge of inputs' data format. Though several common used file format are provided with documentation, much more are not. How to obtain the format information of inputs is a hard open problem. Machine learning techniques and format methods are used to solve this problem. Work (Godefroid et al. 2017) uses machine learning techniques, specifically, the recurrent neural networks, to learn the grammar of input files and consequently use the learned grammar to generate format-fulfilled testcases. Work (Wang et al. 2017) uses format method, specifically, it defines a probabilistic context-sensitive grammar and extract the format knowledge to generate well-format seed inputs.

More state-of-the-art fuzzers employ a mutation-based fuzzing strategy. Testcases are generated by modifying part of the seed inputs in the mutation process. In a blind mutation fuzzing process, mutators randomly modify bytes of seeds with random values or several special values, which is proved to be quite inefficient. Thus how to determine the location to modify and the value used in modifying is another key challenge. In coverage based fuzzing, bytes that could affect the control flow transfer should be first modified. Taint analysis technique is used to track the affection of bytes on control flow to locate key bytes of seeds in mutation (Rawat et al. 2017). Known the key locations is just the beginning. Fuzzing process

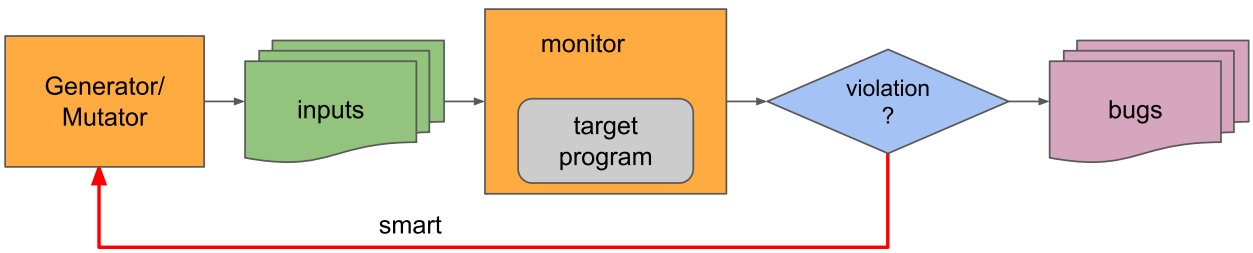

Fig. 5 A sketch of smart fuzz

Table 5 Techniques integrated in fuzzing

	Testcase Generation		Program execution	
Techniques	Generation	Mutation	Guiding	Path exploration
Static analysis		√	√	√
Taint analysis		√	√	
Instrumentation		√	√	√
Symbolic execution				√
Machine learning	√	√		
Format Method	√			

are often blocked in some branches, including validations and checks. For example, magic bytes and other value comparison in condition judgment. Techniques including reverse engineering and taint analysis are used. By scanning the binary code and collecting immediate values from condition judgment statements and utilizing the collected values as candidate values in the mutation process, fuzzers could pass some key validations and checks, like magic bytes and version check. Rawat et al. (2017) New techniques like machine learning techniques are also tried to solve old challenges. Researchers from Microsoft utilize machine learning techniques like deep neural networks (DNN) to predict which bytes to mutate and what value to use in mutation based on previous fuzzing experience via LSTM.

Program execution

In the main fuzzing loop, target programs are executed repeatedly. Information of program execution status are extracted and used to improve the program execution. Two key problems involved in the execution phase is how to guide the fuzzing process and how to explore new path.

Fuzzing process is often guided to cover more code and discover bugs faster, thus path execution information is required. Instrumentation technique is used to record the path execution and calculate the coverage information in coverage based fuzzing. According to whether or not the source code if provided, both complied-in instrumentation and external instrumentation are used. For directed fuzzing, static analysis techniques like pattern recognition are used to specify and identify the target code, witch is more vulnerable. Static analysis techniques could also be used to gather control flow information, e.g. the path depth, which could be used as another reference in the guiding strategy (Rawat et al. 2017). Path execution information collected via instrumentation could help direct the fuzzing process. Some new system features and hardware features are also used in the execution information collection. Intel Processor Trace (Intel PT) is a new feature provided by Intel processors that could expose an accurate and detailed trace of activity with triggering and filtering capabilities to help with isolating the tracing that matters (James 2013). With the advantage of high execution speed and no source dependency, Intel PT could be used to trace the execution accurately and efficiently. The feature is utilized in fuzzing on OS kernels in KAFL (Schumilo et al. 2017), and proved to be quite efficient.

Another concern in testing execution is to explore new path. Fuzzers need to pass complex condition judgment in the control flow of programs. Program analysis techniques including static analysis, taint analysis and .etc., could be used to identify the block point in the execution for consequent solving. Symbolic execution technique has a natural advantage in path exploration. By solving the constrain set, symbolic execution technique could compute values that fulfill specific conditional requirement. TaintScope (Wang et al. 2010) utilize the symbolic execution technique to solve the checksum validation that always block the fuzzing process. Driller (Stephens et al. 2016) leverages the concolic execution to bypass the conditional judgment and find deeper bugs.

After years of development, fuzzing has become more fine-grained, flexible and smarter than ever. Feedback-driven fuzzing provides an efficient way of guided testing, traditional and new techniques play roles of sensors to gain various information during the testing execution and make the fuzzing guided accurately.

Fuzzing towards different applications

Fuzzing has been used to detect vulnerabilities on massive applications since its appearance. According to characteristics of different target applications, different fuzzers and different strategies are used in practice. In this section, we present and summarize several mainly fuzzed types of applications.

File format fuzzing

Most applications involve file handling, and fuzzing is widely used in finding bugs of these applications. Fuzzing test could be operated with files both with or without standard format. Most common used document files, images and media files are files with standard formats. Most researches on fuzzing mainly focus on file format fuzzing, and lots of fuzzing tools are proposed, like Peach (PeachTech 2017), state-of-the-art AFL and its extensions (Rawat et al. 2017; Böhme et al. 2017, 2017). Previous introduction has involved with a variety of file format fuzzers, and we will not emphasize other tools here.

An important subfield of file format fuzzing is fuzzing on web browsers. With the development of web browsers, browsers are extended to support more function than ever. And the type of file handled by browsers has

expanded from traditional HTML, CSS, JS files to other types of files, like pdf, svg and other file formats handled by browser extensions. Specifically, browsers parse the web pages into a DOM tree, which interprets the web page into a document object tree involved with events and responses. Particularly, the DOM parsing and page rendering of browsers are currently popular fuzzing targets. Well known fuzzing tools towards web browsers include the Grinder framework (Stephenfewer 2016), COMRaider (Zimmer 2013), BF3 (Aldeid 2013) and so on.

Kernel fuzzing

Fuzzing on OS kernels is always a hard problem involved with many challenges. First, different with userland fuzzing, crashes and hangs in kernel will bring down the whole system, and how to catch the crashes is an open problem. Secondly, the system authority mechanism result in a relatively closed execution environment, considering that fuzzers are generally run in ring 3 and how to interact with kernels is another challenge. The current best practice of communication with kernel is calling kernel API functions. Besides, widely used kernels like Windows kernel and MacOS kernel are closed source, and is hard to instrument with a low performance overhead. With the development of smart fuzzing, some new progress has been made in kernel fuzzing.

Generally, OS kernels are fuzzed by randomly calling kernel API functions with randomly generated parameter values. According to the focus of fuzzers, kernel fuzzers could be divided into two categories: knowledge based fuzzers and coverage guided fuzzers.

In knowledge based fuzzers, knowledge on kernel API functions are leveraged in the fuzzing process. Specifically, fuzzing with kernel API function calls is faced with two main challenges: (1) the parameters of API calls should have random yet well-formed values that follow the API specification, and (2) the ordering of kernel API calls should appear to be valid (Han and Cha 2017). Representative work include Trinity (Jones 2010) and IMF (Han and Cha 2017). Trinity is a type aware kernel fuzzer. In Trinity, testcases are generated based on the type of parameters. Parameters of syscalls are modified according to the data type. Besides, certain enumeration values and the range of values are also provided to help generating well-formed testcases. IMF tries to learn the right order of API execution and the value dependence among API calls, and leverage the learned knowledge into the generation of testcases.

Coverage based fuzzing has proved a great success in finding userland application bugs. And people begin to apply the coverage based fuzzing method in finding kernel vulnerabilities. Representative work include syzkaller (Vyukov 2015), TriforceAFL (Hertz 2015) and kAFL (Schumilo et al. 2017). Syzkaller instrument the kernel via compilation and run the kernel over a set of QEMU virtual machines. Both coverage and security violations are tracked during fuzzing. TriforceAFL is a modified version of AFL that supports kernel fuzzing with QEMU full-system emulation. KAFL utilized the new hardware feature, Intel PT, to track the coverage and only track kernel code. The experiment shows that KAFL is about 40 times faster than Triforce and greatly improve the efficiency.

Fuzzing of protocols

Currently, a lot of local applications are transformed to network service, in a B/S mode. Services are deployed on network and client applications communicate with servers via network protocols. Security testing on network protocols becomes another important concern. Security problems in protocol could result in more serious damage than local applications, such as the denial of service, information leakage and so on. Cooperate fuzzing with protocols involves in different challenges compared to file format fuzzing. First, services may define their own communication protocols, which are difficult to determine the protocol standards. Besides, even for documented protocols with standard definition, it is still very hard to follow the specification such as RFC document.

Representative protocol fuzzers include SPIKE, which provides set of tools that allows users to quickly create network protocol stress testers. Serge Gorbunov and Arnold Rosenbloom proposed AutoFuzz (Gorbunov and Rosenbloom 2010), which learns the protocol implementation by constructing a Finite State Automaton and further leverage the learned knowledge to generate testcases. Greg Banks et al. proposed SNOOZE (Banks et al. 2006), which identifies the protocol flaws with a stateful fuzzing approach. Joeri de Ruiter's work (De Ruiter and Poll 2015) propose a protocol state fuzzing method, which describe the TLS working state in a state machine and process the fuzzing according to the logical flow. Previous work generally employs a stateful method to model the protocol working process and generates testcases according to protocol specifications.

New trends of fuzzing

As an automated method for detecting vulnerabilities, fuzzing has shown its high effectiveness and efficiency. However, as mentioned in previous sections, there are still a lot of challenges need to be solved. In this section, we give a brief introduction of our own understanding for reference.

First, smart fuzzing provides more possibilities for the improvement of fuzzing. In previous work, traditional static and dynamic analysis are integrated in fuzzing to help improve this process. A certain improvement has been made, but limited. By collecting the target program

execution information via various ways, smart fuzzing provides a more elaborate control of the fuzzing process, and lots of fuzzing strategies are proposed. With a deeper understanding of different types of vulnerabilities, and utilizing the characteristic of vulnerabilities in fuzzing, smart fuzzing could help find more sophisticated vulnerabilities.

Second, new techniques could help improve vulnerability in many ways. New techniques, like machine learning and related techniques have been used to improve the test-case generation in fuzzing. How to combine the advantages and characteristics of new techniques with fuzzing, and how to transform or split the key challenges in fuzzing into problems that new techniques are good at is another question worthy of consideration.

Third, new system features and hardware features should not be ignored. Work (Vyukov 2015) and (Schumilo 2017) has shown that new hardware features greatly improved the efficiency of fuzzing, and gave us a good inspiration.

Conclusion

Fuzzing is currently the most effective and efficient vulnerability discovery solution. In this paper, we give a comprehensive review and summary of fuzzing and its latest progress. Firstly, we compared fuzzing with other vulnerability discovery solutions, and then introduced the concepts and key challenges of fuzzing. We emphatically introduced the state-of-the-art coverage based fuzzing, which have made great process in recent years. At last, we summarized the techniques integrated with fuzzing, the applications and possible new trends of fuzzing.

Abbreviations

AFL: American Fuzzy Lop; BB: Basic Block; DNN: Deep Neural Networks; LSTM: Long Short-Term Memory; POC: Proof of Concept

Acknowledgements

This research was supported in part by the National Natural Science Foundation of China (Grant No. 61772308 61472209, and U1736209), and Young Elite Scientists Spon- sorship Program by CAST (Grant No. 2016QNRC001), and award from Tsinghua Information Science And Technology National Laboratory.

Authors' contributions

JL drafted most of the manuscripts, BZ helped proofreading and summarized parts of the literature, and Prof. CZ abstracted the classifier for existing solutions and designed the overall structure of the paper. All authors read and approved the final manuscript.

Competing interests

CZ is currently serving on the editorial board for Journal of Cybersecurity.

Publisher's Note

Received: 3 January 2018 Accepted: 17 April 2018
Published online: 05 June 2018

References

Aldeid (2013) Browser fuzzer 3. https://www.aldeid.com/wiki/Bf3. Accessed 25 Dec 2017

Amini P (2017) Sulley fuzzing framework. https://github.com/OpenRCE/sulley. Accessed 25 Dec 2017

Banks G, Cova M, Felmetsger V, Almeroth K, Kemmerer R, Vigna G (2006) Snooze: toward a stateful network protocol fuzzer. In: International Conference on Information Security. Springer, Berlin. pp 343–358

Böhme M, Pham V-T, Nguyen M-D, Roychoudhury A (2017) Directed greybox fuzzing. In: Proceeding CCS '17 Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. ACM, New York. pp 2329–2344. https://doi.org/10.1145/3133956.3134020

Böhme M, Pham VT, Roychoudhury A (2017) Coverage-based greybox fuzzing as markov chain. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM. pp 1032–1043

Bowne S (2015) Fuzzing with spike. https://samsclass.info/127/proj/p18-spike.htm. Accessed 25 Dec 2017

Cha SK, Avgerinos T, Rebert A, Brumley D (2012) Unleashing mayhem on binary code. In: Security and Privacy (SP) 2012 IEEE Symposium on. IEEE, San Francisco. pp 380–394. https://doi.org/10.1109/SP.2012.31

De Ruiter J, Poll E (2015) Protocol state fuzzing of tls implementations. In: Proceeding SEC'15 Proceedings of the 24th USENIX Conference on Security Symposium. USENIX Association, Berkeley. pp 193–206

Godefroid P, Levin MY, Molnar D (2012) Sage: whitebox fuzzing for security testing. Queue 10(1):20

Godefroid P, Peleg H, Singh R (2017) Learn & fuzz: Machine learning for input fuzzing. In: Proceeding ASE 2017 Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. IEEE Press, Piscataway. pp 50–59

Gorbunov S, Rosenbloom A (2010) Autofuzz: Automated network protocol fuzzing framework. IJCSNS 10(8):239

Han H, Cha SK (2017) Imf: Inferred model-based fuzzer. In: Proceeding CCS '17 Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. ACM, New York. pp 2345–2358. https://doi.org/10.1145/3133956.3134103

Hertz J (2015) Triforceafl . https://github.com/nccgroup/TriforceAFL. Accessed 25 Dec 2017

James R (2013) Processor tracing. https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing. Accessed 25 Dec 2017

Jones D (2010) trinity. https://github.com/kernelslacker/trinity. Accessed 25 Dec 2017

King JC (1976) Symbolic execution and program testing. Commun ACM 19(7):385–394

lcamtuf (2014) Fuzzing random programs without execve(). https://lcamtuf.blogspot.jp/2014/10/fuzzing-binaries-without-execve.html. Accessed 25 Dec 2017

libfuzzer (2017) A library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html. Accessed 25 Dec 2017

Liu B, Shi L, Cai Z, Li M (2012) Software vulnerability discovery techniques: A survey. In: Multimedia Information Networking and Security (MINES), 2012 Fourth International Conference on. IEEE, Nanjing. pp 152–156. https://doi.org/10.1109/MINES.2012.202

Luk C-K, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi VJ, Hazelwood K (2005) Pin: building customized program analysis tools with dynamic instrumentation. In: Acm sigplan notices, volume 40. ACM, Chicago. pp 190–200

Nichols N, Raugas M, Jasper R, Hilliard N (2017) Faster fuzzing: Reinitialization with deep neural models. arXiv preprint arXiv:1711.02807

PeachTech (2017) Peach. https://www.peach.tech/. Accessed 25 Dec 2017

Petsios T, Zhao J, Keromytis AD, Jana S (2017) Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In: Proceeding CCS '17 Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. ACM, New York. pp 2155–2168. https://doi.org/10.1145/3133956.3134073

Rajpal M, Blum W, Singh R (2017) Not all bytes are equal: Neural byte sieve for fuzzing. arXiv preprint arXiv:1711.04596

Rawat S, Jain V, Kumar A, Cojocar L, Giuffrida C, Bos H (2017) Vuzzer: Application-aware evolutionary fuzzing. In: Proceedings of the Network and Distributed System Security Symposium (NDSS). https://www.vusec.net/download/?t=papers/vuzzer_ndss17.pdf

Schumilo S, Aschermann C, Gawlik R, Schinzel S, Holz T (2017) kAFL: Hardware-assisted feedback fuzzing for OS kernels. In: Kirda E, Ristenpart T (eds). 26th USENIX Security Symposium, USENIX Security 2017. USENIX Association, Vancouver. pp 167–182

Serebryany K, Bruening D, Potapenko A, Vyukov D (2012) Addresssanitizer: A fast address sanity checker. In: Proceeding USENIX ATC'12 Proceedings of the 2012 USENIX conference on Annual Technical Conference. USENIX Association, Berkeley. pp 309–318

Serebryany K, Iskhodzhanov T (2009) Threadsanitizer: data race detection in practice. In: Proceedings of the Workshop on Binary Instrumentation and Applications. pp 62–71

Shirey RW (2000) Internet security glossary. https://tools.ietf.org/html/rfc2828. Accessed 25 Dec 2017

Stephenfewer (2016) Grinder. https://github.com/stephenfewer/grinder. Accessed 25 Dec 2017

Stephens N, Grosen J, Salls C, Dutcher A, Wang R, Corbetta J, Shoshitaishvili Y, Kruegel C, Vigna G (2016) Driller: Augmenting fuzzing through selective symbolic execution. In: NDSS, volume 16, San Diego. pp 1–16

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

Takanen A, Demott JD, Miller C (2008) Fuzzing for software security testing and quality assurance. Artech House

The Clang Team (2017) Dataflowsanitizer. https://clang.llvm.org/docs/DataFlowSanitizerDesign.html. Accessed 25 Dec 2017

The Clang Team (2017) Leaksanitizer. https://clang.llvm.org/docs/LeakSanitizer.html. Accessed 25 Dec 2017

Van Sprundel I (2005) Fuzzing: Breaking software in an automated fashion. Decmember 8th

Vyukov D (2015) Syzkaller. https://github.com/google/syzkaller. Accessed 25 Dec 2017

Wang J, Chen B, Wei L, Liu Y (2017) Skyfire: Data-driven seed generation for fuzzing. In: Security and Privacy (SP), 2017 IEEE Symposium on. IEEE, San Jose. https://doi.org/10.1109/SP.2017.23

Wang S, Nam J, Tan L (2017) Qtep: quality-aware test case prioritization. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. ACM, New York. pp 523–534. https://doi.org/10.1145/3106237.3106258

Wang T, Wei T, Gu G, Zou W (2010) Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In: Security and privacy (SP) 2010 IEEE symposium on. IEEE, Berkeley. pp 497–512. https://doi.org/10.1109/SP.2010.37

Wichmann BA, Canning AA, Clutterbuck DL, Winsborrow LA, Ward NJ, Marsh DWR (1995) Industrial perspective on static analysis. Softw Eng J 10(2):69–75

Wikipedia, Wannacry ransomware attack (2017). https://en.wikipedia.org/wiki/WannaCry_ransomware_attack. Accessed 25 Dec 2017

Wikipedia (2017) Dynamic program analysis. https://en.wikipedia.org/wiki/Dynamic_program_analysis. Accessed 25 Dec 2017

Wu Z-Y, Wang H-C, Sun L-C, Pan Z-L, Liu J-J (2010) Survey of fuzzing. Appl Res Comput 27(3):829–832

Xu W, Kashyap S, Min C, Kim T (2017) Designing new operating primitives to improve fuzzing performance. In: Proceeding CCS '17 Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. ACM, New York. pp 2313–2328. https://doi.org/10.1145/3133956.3134046

Yang Q, Li JJ, Weiss DM (2007) A survey of coverage-based testing tools. The Computer Journal 52(5):589–597

Zalewski, M (2017) American fuzzy lop. http://lcamtuf.coredump.cx/afl/. Accessed 25 Dec 2017

Zalewski M (2017) Afl technical details. http://lcamtuf.coredump.cx/afl/technical_details.txt. Accessed 25 Dec 2017

Zimmer D (2013) Comraider. http://sandsprite.com/tools.php?id=16. Accessed 25 Dec 2017