SemFuzz: Semantics-based Automatic Generation of Proof-of-Concept Exploits

Wei You[1], Peiyuan Zong[2,3], Kai Chen[2,3,*], XiaoFeng Wang[1,*], Xiaojing Liao[4], Pan Bian[5], Bin Liang[5]
[1]School of Informatics and Computing, Indiana University Bloomington, Indiana, USA
[2]SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[3]School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
[4]Department of Computer Science, William and Mary, Virginia, USA
[5]School of Information, Renmin University of China, Beijing, China
{youwei,xw7}@indiana.edu,{zongpeiyuan,chenkai}@iie.ac.cn,{xliao02}@wm.edu,{bianpan,liangb}@ruc.edu.cn

ABSTRACT

Patches and related information about software vulnerabilities are often made available to the public, aiming to facilitate timely fixes. Unfortunately, the slow paces of system updates (30 days on average) often present to the attackers enough time to recover hidden bugs for attacking the unpatched systems. Making things worse is the potential to automatically generate exploits on input-validation flaws through reverse-engineering patches, even though such vulnerabilities are relatively rare (e.g., 5% among all Linux kernel vulnerabilities in last few years). Less understood, however, are the implications of other bug-related information (e.g., bug descriptions in CVE), particularly whether utilization of such information can facilitate exploit generation, even on other vulnerability types that have never been automatically attacked.

In this paper, we seek to use such information to generate proof-of-concept (PoC) exploits for the vulnerability types never automatically attacked. Unlike an input validation flaw that is often patched by adding missing sanitization checks, fixing other vulnerability types is more complicated, usually involving replacement of the whole chunk of code. Without understanding of the code changed, automatic exploit becomes less likely. To address this challenge, we present SemFuzz, a novel technique leveraging vulnerability-related text (e.g., CVE reports and Linux git logs) to guide automatic generation of PoC exploits. Such an end-to-end approach is made possible by natural-language processing (NLP) based information extraction and a semantics-based fuzzing process guided by such information. Running over 112 Linux kernel flaws reported in the past five years, SemFuzz successfully triggered 18 of them, and further discovered one zero-day and one undisclosed vulnerabilities. These flaws include use-after-free, memory corruption, information leak, etc., indicating that more complicated flaws can also be automatically attacked. This finding calls into question the way vulnerability-related information is shared today.

∗ Corresponding Authors

CCS'17, Oct. 30–Nov. 3, 2017, Dallas, TX, USA.
 ISBN 978-1-4503-4946-8/17/10…$15.00
DOI: http://dx.doi.org/10.1145/3133956.3134085

CCS CONCEPTS

•**Security and privacy** → *Software security engineering;*

KEYWORDS

exploit generation, vulnerability, patch, fuzzing, semantics

1 INTRODUCTION

Today information and patches for software vulnerabilities, even those security-critical ones, are often publicly available, for the purpose of raising the awareness of these problems and facilitating their timely fixes. Unfortunately, system updates are often slow, even in the presence of security flaws, as evidenced by the recent WannaCry ransomware outbreak [22], which exploits the EternalBlue bug whose patch has been released months ago. As a result, miscreants are often given a large time frame (30 days on average [45]), during which they can leverage the information exposed by public patches to recover hidden bugs, and attack the systems yet to be patched. Indeed, research almost a decade ago [28] shows that it is possible to *automatically* reverse-engineer a patch to generate an exploit for the vulnerability meant to be fixed by the patch. Less understood, however, are the implications of other information, such as the reports from common vulnerabilities and exposures (CVE) systems [9], Linux git logs [15] and bug descriptions posted on forums and blogs [12–14], to this ongoing patching-exploit arms race. Particularly, *from the attacker's viewpoint, whether such information can also be leveraged for automatic construction of more complicated exploits? from the defender's side, how to control such information leaks to make the automatic attack harder to succeed?*

Challenges in automatic exploit generation. Actually, automatic exploit generation is hard. The prior study [28] only creates the attacks on *input-validation* flaws, a type of bugs relatively easy to discover and fix, given their prominent feature (missing of sanitization checks). An exploit on such flaws can be constructed from a patch by seeking an input that fails the newly added checks. In other words, to generate such exploits, an automatic approach first finds a path from the program's entry point to the new check, then recovers the constraints for reaching the check on the path. Such constraints, which are built through symbolic execution [36], are then resolved to obtain an input that fails the check and therefore is likely to cause an exploit on the vulnerability.

Table 1: The types of vulnerability addressed in this paper.

Vulnerability Type	Percentage
Information leak/disclosure	10.62%
Denial of service	9.38%
Null pointer dereference	9.04%
Uncontrolled resource consumption	7.91%
Use after free	6.67%
Buffer overflow	5.76%
Memory corruption	4.18%
Integer overflow	3.39%
Buffer over-read	3.16%
Improper access control	2.82%
Race conditions	2.60%
Numeric errors	2.49%
Double free	1.36%
Infinite loop	1.24%
Deadlock	0.68%
Divide by zero error	0.45%

Compared with such input-validation flaws, other types of vulnerabilities (like uncontrolled resource consumption, deadlock, memory corruption, etc.), however, are more complicated and *cannot* be patched by simply adding a check. Actually, more often than not, their related vulnerable statements or even the whole chunk of code need to be replaced by the patch, making the vulnerable code hard to detect, not to mention an attempt to exploit it through the aforementioned constraints finding and resolving. To the best of our knowledge, so far, little has been done to automate the exploits of these complicated, deep program flaws.

Even for the attack on input validation, symbolic execution and constraint solving are known to be difficult. For real-world programs, path constraints leading to vulnerable program locations tend to be non-linear, oftentimes, rendering current solvers (e.g., STP [19]) hard to figure out a suitable input. Making it worse are the global variables in the target program, whose values are often assigned in one thread but used in another. Once this happens, the path constraints for reaching vulnerable code would become incomplete (given the missing assignment) and cannot be made right without looking at other threads. This, however, becomes too complicated for the current symbolic execution and constraint solving systems to handle. For example, CVE-2017-6347 reports a vulnerable function *ip_cmsg_recv_checksum* in Linux kernel invoked by the system call `recvfrom`. An essential condition for triggering the vulnerable function is to fill a `sk_buff` buffer, which will be referenced in the kernel structure `socket`. However, on the path from `recvfrom` to the vulnerable function, no such code exists, and it turns out that this is done in another system call `sendto`, which is supposed to be called before invoking `recvfrom`.

Our approach. In this paper, we demonstrate that complicated vulnerabilities can also be automatically exploited, even in the absence of sophisticated constraint solving techniques. Instead, we utilize non-code text related to a vulnerability, particularly CVE reports and Linux git logs, to extract *guidances*, which are found to be sufficiently informative for helping discover and trigger a set of deep bugs. Our technique, called *semantics-based fuzzing* (*SemFuzz*), automatically analyzes bug reports to create end-to-end *proof-of-concept* (PoC) exploits [1] on various Linux kernel vulnerabilities, including double free, use-after-free and memory corruption, etc., as illustrated in Table 1. Compared with the prior work [28], which targets the input-validation vulnerabilities (only 5% among all the Linux kernel flaws reported in recently [2]), SemFuzz is capable of handling a wide range of vulnerabilities within the kernel code. Note that unlike relatively simple programs receiving a single input (e.g., a file), as studied in the prior research, the kernel code is much more complicated, with its vulnerable component only reachable through some specific system call sequences (e.g., `sendto` and then `recvfrom`).

More specifically, given a reported vulnerability, SemFuzz first utilizes *Natural Language Process* (NLP) to analyze its CVE and git log reports. CVE provides a reference method for publicly known security vulnerabilities and exposures, publishing the information such as affected versions, vulnerability type, and vulnerable functions. The Linux git log includes a patch and the description about how it works. Such information is invaluable for the exploit generation process. For example, they tell us the exact version of the vulnerable program for setting up the right testing environment. More importantly, it may also explain the types of vulnerabilities, what to expect when hitting the target (crash, hang, memory corruption, etc.), the whereabouts of a vulnerable function, and even the key variables and their values for guiding the program execution toward the bug. Leveraging the information automatically collected, SemFuzz creates a call sequence reaching the vulnerable function, and then iteratively "mutates" the parameters of individual calls to move towards the patched code inside the function, until the target vulnerability is triggered.

This semantics-based, intelligent fuzzing technique turns out to be very effective. In our research, we ran our implementation over 112 Linux kernel vulnerabilities reported by CVE in the past five years. 16% of them were successfully triggered. For the remaining CVEs, although SemFuzz did not produce end-to-end PoC exploits, it automatically discovered the inputs that move the program execution towards vulnerable functions, which can significantly speed up the process to manually build exploits. Also interestingly, our approach even discovered one *zero-day* vulnerability and one *undisclosed* vulnerability, when fuzzing the kernel for triggering known flaws. These new findings have already been confirmed by the Linux kernel developer group. Our studies show that these new vulnerabilities either appear around the known flaws or are similar problems inside equivalent components (Section 6.5). The results strongly indicate that public bug descriptions today indeed leak out critical information, which can be practically utilized to generate attack instances, exploiting the vulnerabilities that cannot be attacked automatically through patch analysis alone.

Contribution. The contributions of this paper are as follows:

[1] Following [28], we define a proof-of-concept exploit as inputs that trigger a vulnerability to crash the target program without executing further attacks such as control-flow diversion.

[2] We consider the flaws that can be fixed by adding sanitization checks on inputs as input-validation vulnerabilities, as defined in the prior work [28].

• *New technique.* We designed and implemented SemFuzz, the first semantics-based, intelligent fuzzer that automatically recovers vulnerability-related knowledge from text reports and utilizes such information to guide systematic construction of test cases for triggering a known or related unknown flaw.

• *New understanding.* Our study demonstrates that non-code textual bug descriptions (e.g., CVE, Linux git logs) are valuable information sources for reconstructing exploits on known vulnerabilities. Over 112 Linux kernel flaws reported in the past five years, SemFuzz successfully triggered 18 and further discovered two related unknown bugs. More importantly, our research goes beyond simple input-validation bugs, providing evidence that more complicated flaws can also be automatically attacked using bug-related public information. This finding calls into question the way vulnerability-related information is shared today, and could lead to more serious effort to control the information leaks from those sources.

2 BACKGROUND

Vulnerability and Patch. A vulnerability is a weakness in software or hardware components which allows an attacker to reduce a system's information assurance [20]. By exploiting such vulnerabilities, attackers could alter system resources or affect their operations, compromising integrity or availability. The consequences of attacks include millions of dollars lost in banks [1], billions of users' privacy leakage [5], etc. To mitigate the impacts of vulnerabilities, patches are designed to address them. For example, a program containing input-validation vulnerabilities [11] accepts unsafe inputs which may let the program run in an abnormal way. Their patches are usually in the form of sanitization checks that distinguish the unsafe inputs and exclude them outside the vulnerable program code. While serving to fix vulnerabilities, patches are also exposing information of the vulnerabilities at the same time. Attackers with strong capability on vulnerability analysis may reverse engineer the patches and even generate exploits for attacks. Note that the time interval between releasing a patch on developers' side and installing the patch on users' side is 30 days on average [45], which gives attackers enough time to impact lots of users. The situation becomes even worse when exploits could be generated in an automatic way [28], which lowers the bar of attackers' capability on vulnerability analysis. Fortunately, recent researches show that only input-validation vulnerabilities (5% of all vulnerabilities [6]) were prone to such problem; and in reality, attackers can only generate exploits for a subset of such vulnerabilities due to the limitation of symbolic execution [50]. However, in this paper, we are surprised to find that many vulnerability types are exposed to such problem, including uncontrolled resource consumption, deadlock, memory corruption, etc.

CVE. CVE is a reference system sponsored by US-CERT for publicly known information-security vulnerabilities and exposures [9]. Till now, it maintained more than 85,000 vulnerabilities. Each year, around 10,000 new vulnerabilities are added into the CVE system. Every user can submit descriptions (e.g., the affected product and version, the type of vulnerability, etc.) of a previously unknown vulnerability to CVE. Once the vulnerability is verified by software vendors, CVE assigns an ID to the vulnerability for reference. To maximize the protection of the affected vendors, CVE will only open vulnerability information to the public after patches are prepared. Interestingly, in this paper, we find the descriptions in CVE can actually help attackers to quickly generate PoC exploits, rather than simply serving as a reference system.

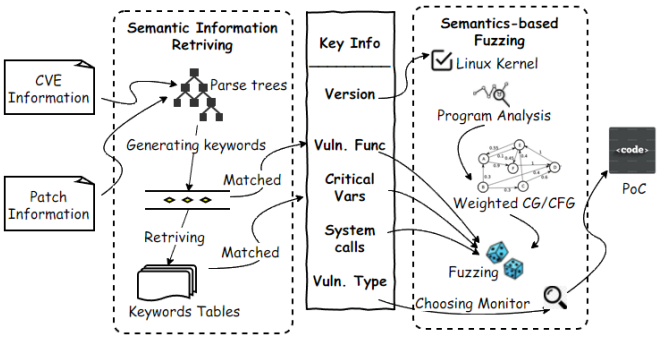


Figure 1: The architecture of SemFuzz.

Fuzzing. Fuzzing is an automated testing technique that feeds manipulated inputs (e.g., random ones) to a software program [53]. By observing the execution of a program, the tool of fuzzing (also called *fuzzer*) reports a vulnerability whenever an abnormal run (e.g., crash) is captured. Since fuzzing all the inputs of a program is almost impossible, it is vital to choose a relatively small subset of inputs that could still trigger the vulnerability. To fit this need, a fuzzer should try to collect various kinds of valuable information to guide the fuzzing process. Some recent studies observe that the running status of a program could assist the selection of inputs to avoid redundant runs [27, 49]. In this paper, we find that, besides the running status, the non-code descriptions in CVE and Linux git logs can also help the fuzzer to avoid unnecessary runs, saving a lot of time in the fuzzing process. In particular, we use the semantics-based approach (e.g., NLP) to automatically analyze the description and extract necessary information for feeding to the fuzzer.

3 SEMFUZZ: DESIGN OVERVIEW

To address the challenges in triggering deep vulnerabilities, our solution is to fuzz the target program by leveraging semantic information collected from vulnerability-related text sources. In this way, we can avoid generating and solving complicated constraints on inputs and also leverage new knowledge discovered to guide exploit construction. The procedure is illustrated in Figure 1, which involves two main stages: (1) semantic information retrieving and (2) semantics-based fuzzing.

Specifically, given a vulnerability in the Linux kernel, as documented by CVE, the first step is to extract useful semantic information about the vulnerability from the descriptions in its CVE and its corresponding Linux git log. Such information includes *affected version*, *vulnerability type*, *vulnerable functions*, *critical variables* and *system calls*. Then, SemFuzz loads the target kernel (with the affected version) and fuzzes it using elaborately constructed test cases. The seed input (for the test cases) is first generated using the system call information collected from the text descriptions. During the fuzzing process, SemFuzz monitors the runtime status of the target kernel and mutates the inputs using the vulnerable

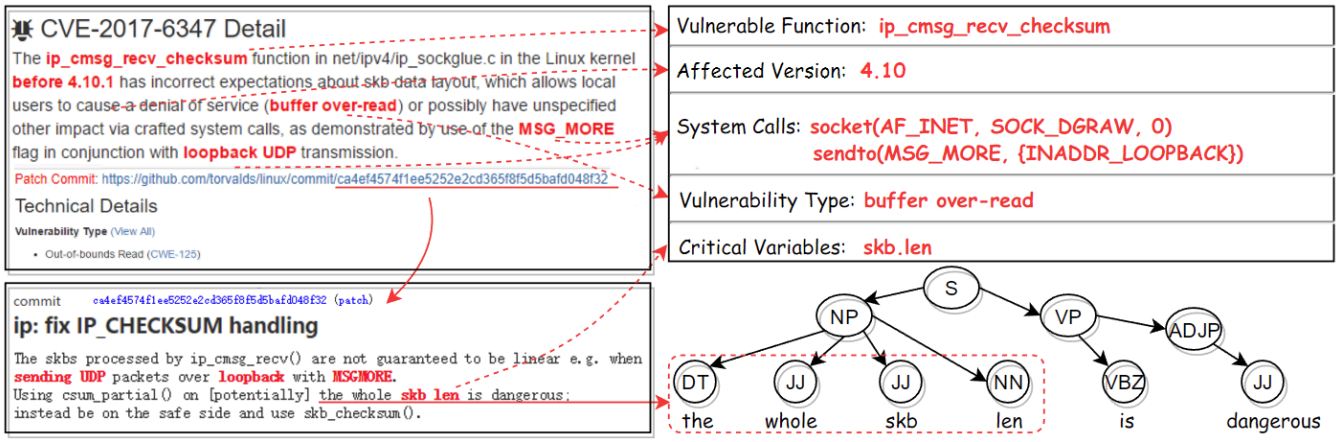


Figure 2: CVE description and Linux git log of CVE-2017-6347.

function and critical variable information, in an attempt to trigger the vulnerability. Once an anomalous event (defined corresponding to the vulnerability type) is observed, an alert will be issued to indicate the PoC exploit is successfully generated.

Example. Figure 2 presents an example that demonstrates how SemFuzz works on a given vulnerability (CVE-2017-6347) . The top-left part of the figure shows the CVE description of the vulnerability and the bottom-left part is the content of its Linux git log, which is linked to the CVE through "patch commit id". In the CVE description, "Linux kernel before 4.10.1" indicates the affected version. SemFuzz first starts a virtual machine with a pre-installed kernel 4.10 (the latest version before 4.10.1) and prepares the fuzzing environments. Using the concepts (i.e., "MSG_MORE", "loopback", "UDP") discovered from the descriptions, SemFuzz builds the seed input (system call sequence) "`socket`(AF_INET, SOCK_DGRAW, 0), `sendto`(..., MSG_MORE, ..., INADDR_LOOPBACK, ...)" to fuzz the kernel (Section 4). During this process, our approach continues to mutate the inputs, based on the information extracted from the patch (e.g., control-flow graph of the vulnerable function) and the feedback from monitoring the critical variables ("*skb.len*"), so as to reach the vulnerable function ("*ip_cmsg_recv_checksum*") as mentioned by the CVE, and further trigger the buffer over-read vulnerability (as described by the vulnerability type). Details of the mutation strategies are illustrated in Section 5.

Scope and assumption. SemFuzz is designed to automatically generate a PoC exploit from a Linux kernel patch with the help of vulnerability descriptions in CVE and Linux git log. Our current implementation can handle several types of common vulnerabilities, including use-after-free, information leak/disclosure, null pointer dereference, etc. (see Table 1 for the full list). The objective here is to generate inputs to trigger known vulnerabilities, though the technique also helps us discover a few similar but unknown flaws (Section 6.5).

4 SEMANTIC INFORMATION RETRIEVING

As mentioned earlier, semantic information (including affected version, vulnerability type, vulnerable functions, critical variables and system calls) all comes from the text content of CVE and Linux git log [3]. Such content is in natural language, without a well-defined structure. Therefore, direct extraction of knowledge, through syntactic means such as regular expression based string match, does not work well, due to the semantic ambiguity of some content components. As an example, "read" can be a verb (e.g., in the phrase "buffer over read") or a noun (e.g., in the sentence "by a read system call"). Also, the simple approach (string matching) fails to consider the dependency relations between words in a sentence. For example, in the sentence "the whole skb len is dangerous", the word "skb" modifies "len", indicating that `len` is a field in the `skb` structure. To accurately recover such target information, we utilize Natural Language Processing (NLP) techniques, including Part-of-Speech (POS) Tagging, Phrase Parsing and Syntactic Parsing. Specifically, SemFuzz builds a parse tree to recognize the POS tag of each word and to identify the syntactic clause in a sentence for semantic analysis. Using these techniques, we show that target vulnerability information can be accurately identified. Below we elaborate how our approach works.

Generating parse tree. The parse tree is an ordered, rooted tree that represents the syntactic structure of a sentence according to a Context-Free Grammar (CoFG) [30]: the root of the parse tree is labeled as the start of the tree; interior nodes are labeled as non-terminals (e.g., "VP" for verb phrase, "NP" for noun phrase, etc.), representing syntactically correlated word sequences or phrases; and the leaves of the tree are labeled as terminals (e.g., "JJ" for adjective, "NN" for noun, etc.), representing individual words of this sentence.

In our research, we use the NLP tool pyStatParser [18] to learn the Probabilistic Context-Free Grammar (PCoFG) from the Penn Treebanks [40] and generate a parse tree for each sentence in the CVE and git log. Figure 2 shows part of a parse tree for a sentence in the git log of CVE-2017-6347. The root of the tree is "S", representing a sentence. The left child of the root is a noun phrase (NP), and the right child of the root is a verb phrase (VP). We can see that concatenating the leaves from left to right constitutes the whole

[3]Contents from CVE and git log are often complementary. For example, system calls usually appear only in CVE while critical variables are commonly in git log.

sentence "the whole skb len is dangerous". The parent node of each leaf (i.e., the word in the sentence) is the word's POS tag. Using the parse tree, SemFuzz can understand the meaning of each word and the relationship between those words. For example, when SemFuzz checks the noun phrase in a subtree, it will find that the word skb as an adjective describes the noun len, which further helps SemFuzz to find out that len is a field of the structure skb (see "Retrieving critical variables" in this Section). At this time, SemFuzz is ready to retrieve necessary information for fuzzing.

Retrieving affected version. Affected version is the Linux kernel version that contains the given CVE vulnerability, which is necessary for SemFuzz to set up the execution environment. In most cases, it is in the form of "Linux kernel 4.10.11" or "kernel 4.1". Note that such version information cannot be directly acquired from git log commit id, since it only gives the release candidate version (e.g., 4.10-rc8) but not the specific release version (e.g., 4.10.1). To extract the release version, SemFuzz first identifies the version number with the following regular expression: "^\d(\.\d{1,2}){0,2}(\.x){0,1}", which could match words like "4.1", "4.1.1", "4.x" and "4.1.x". Then SemFuzz locates the clause containing the version number in the parse tree and checks whether it also includes the term "Linux" or "kernel". If so, SemFuzz views the version number as the affected version. Otherwise, the version number may belong to other applications (e.g., "gcc 4.1"). Sometimes, a preposition can be found before the version number (e.g., "before 4.10.2"). In this case, we use the nearest version that meets the condition (e.g., "4.10.1" here). Note that it is also possible that two version numbers are in the same clause (e.g., "Linux kernel 4.4.22 through 4.4.28"). In this case, we choose the latest version number among the two as the affected version (i.e., "4.4.27" in this case). In this way, the affected version can be successfully identified.

Retrieving vulnerability type. Vulnerability type indicates the anomalous event that SemFuzz needs to observe in the fuzzing process. Examples of the vulnerability type include "use after free", "double free", "memory consumption", etc. To retrieve such information, we first define a list of candidate types and try to find them in the content of CVE and git log. Specifically, to define the candidate types, we use the Common Weakness Enumeration (CWE), which is a community-developed list of common software security weaknesses [10]. It commonly serves as a baseline for weakness identification, mitigation, and prevention efforts. There are about 70 types of Linux kernel related CWEs in total, and we select 16 of them as shown in Table 1. Then SemFuzz looks for the CWE in the clauses identified by the parse tree. SemFuzz only focuses on NP (noun phrase) of the parse tree, where CWEs are most likely to appear. Once the vulnerability type cannot be retrieved from the parse tree, SemFuzz tries to retrieve National Vulnerability Database (NVD) [17] using the CVE number as the keyword. From the `Technical Details` field in the search results, SemFuzz can get the vulnerability type.

Retrieving vulnerable functions. A vulnerable function contains vulnerable code, which is the patched function in the git log. Identifying vulnerable function helps SemFuzz to set up the mutation strategy on inputs, which further increases the performance of fuzzing. To retrieve such a function, SemFuzz compares the unpatched version of the Linux kernel with the patched one (indicated in the git log), and locates the revised functions as the candidate vulnerable functions. We further locate the real ones based on the following observation: (1) if a patched function is also mentioned in the CVE description, this function is more likely to be the vulnerable function; (2) if a variable mentioned in the CVE description or patch description, it is more likely to be related to the vulnerable function. Therefore, SemFuzz firstly searches for the name of patched functions in the parse tree, and treats the one discovered as the vulnerable function. If nothing is found, SemFuzz compares the nouns in the parse tree with the variables in the patched functions. Any match is considered to be the vulnerable function. For example, in Figure 2, only the *ip_cmsg_recv_checksum* function is mentioned in CVE, it is treated as the vulnerable function.

Retrieving critical variables. We define a variable as a critical variable if it meets the following two conditions: (1) it appears in a (unpatched) vulnerable function; and (2) is also mentioned in the description of CVE or git log. The critical variable is closely related to the vulnerability, and may even be the root of the vulnerability. In the example shown in Figure 2, "*skb.len*" is the critical variable. To locate such variables, SemFuzz first extracts all the variables from an unpatched vulnerable function, and builds a symbol table which contains the variables and their type information. Then, SemFuzz checks whether any variable in the symbol table also appears in the clauses in the parse tree. Note that a variable must be a noun or an adjective in a phrase. SemFuzz will not consider words with other POS, such as the preposition or subordinating conjunction. When a word indicating a structure variable modifies the other word (e.g., in "skb.len", skb modifies len), SemFuzz will search the structure to find the most likely field that matches the modified word [4].

Retrieving system calls. As mentioned previously, system calls play a vital role in our fuzzing of Linux kernel. Most vulnerable functions inside the kernel are triggered by system calls. One may consider randomly selecting different system calls for fuzzing. However, considering there are around 400 system calls with more than 1500 parameters, the search space of the call combinations, together with their parameters, is too large for finding the right input to trigger the vulnerability. A randomly selected sequence of system calls (also with randomly selected values of parameters) as inputs is almost always impossible to trigger the vulnerability. After manually checking more than 100 CVEs and their corresponding Linux git logs, we find that each system call mentioned in CVE or git log may either trigger the vulnerable function or set up the running environment that is necessary for triggering the vulnerability. Therefore, correctly retrieving system calls from the description of CVE or git log will greatly improve the performance of the fuzzing.

One simple idea to retrieve system calls is to get the list of Linux system call names and look for them in the description. However, this approach may miss a large amount of valuable information. For the example in Section 3, the content mentions no name of any system call. But the three keywords ("MSG_MORE", "loopback", "UDP") in the content can actually help readers to recall the system call `sendto` and the system call `socket` due to the correlation

[4] In the rare case that the semantics of the fields is given in natural language (e.g., "length" for "len"), we can still capture them from their comments in the structure, which are often there (e.g., structure sk_buff has a comment "@len: length of actual data").

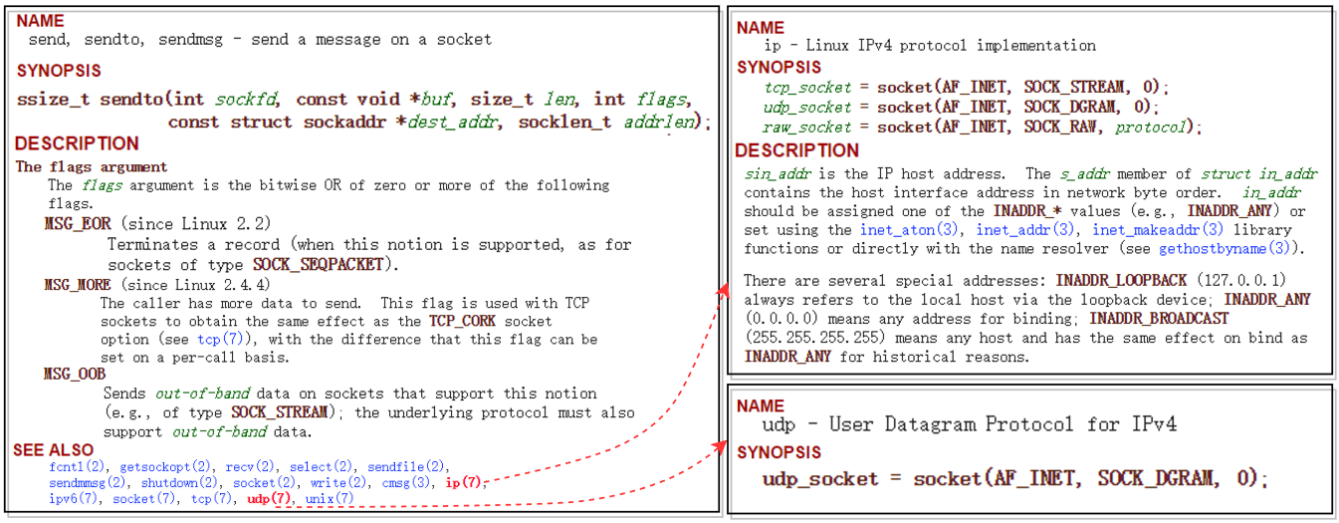


Figure 3: Example of collecting system call information from Linux Programmer Manual.

between them. To build such correlation, our idea is to customize NLP for recovering syscall-related information. Particularly, we built a knowledge base (the relations among system calls and their parameters) for correlating the keywords in CVE or git log descriptions to domain-specific concepts (e.g., linking MSG_MORE to the `flags` parameter of the `sendto` system call). Below we elaborate the details of our approach.

SemFuzz first correlates a system call with its return type and the types of parameters. If the type is an enumeration, the values of parameter should also be included. Such correlation can be automatically achieved by parsing Linux Programmer Manual (LPM) [16] which contains the prototype of every Linux system call [5]. A prototype is a declaration of a function that specifies the function's name, number of parameters, data types of parameters, and return type. The format of a declaration is fixed, and it always appears in the SYNOPSIS field in LPM. For parameters of the enumeration type, all possible enumeration values are presented in the DESCRIPTION field in LPM. What SemFuzz needs to do is to parse all the documents of LPM, extracting the prototype of system calls and the enumeration values of parameters. An example is shown in the left part of Figure 3, which is a manual page for the system call `sendto`. From the figure, we can see that the system call `sendto` has six parameters, as shown in the SYNOPSIS field. The parameter `flags` is an enumeration with values show in the DESCRIPTION field (e.g., MSG_MORE). Each value of the enumeration is connected to `sendto`. Also the parameter `dest_addr` is a pointer to `sockaddr`. The structure `sockaddr` and its fields (which are extracted from the code) are related to `sendto`.

Besides correlating a system call with its prototype and possible enumeration values, SemFuzz also checks the SEE ALSO field in LPM, which usually contains other information (e.g., the protocol that a system call may use). For example, in Figure 3, the SEE ALSO field of the `sendto` system call includes other LPM pages such as `tcp`, `udp` and `ip`. These pages describe the related values of system calls' parameters. Generally, two fields need to be taken care. The first one is the SYNOPSIS field, which gives typical sample code of system calls with specific values of its parameters. SemFuzz connects the page's name (e.g., ip and udp) with the contents in SYNOPSIS. In the example of `udp` (see bottom-right of Figure 3), SemFuzz correlates the keyword `udp` with the system call "socket(AF_INET, SOCK_DGRAM, 0)". The second field is the DESCRIPTION field in the page. Sometimes, it gives special values of critical structures. For example, in the `ip` page (see top-right of Figure 3), SemFuzz recognizes INADDR_LOOPBACK is a special value of the ip address, which can be used to fill the `dest_addr` parameter of the `sendto` system call. After analyzing all the pages in this way, SemFuzz is able to retrieve a system call and its parameters when its keyword is identified in the leaves of the parse tree with POS label NN (i.e., noun). Recall that UDP is used in the CVE description (see Figure 2), SemFuzz can generate the system call with its parameters as follows: "socket(AF_INET, SOCK_DGRAM, 0)".

SemFuzz also correlates a system call with other system calls. As we know, the parameters of a system call *A* may be the return value of another system call *B*. As a result, only after B is executed, A can run with the output of B. SemFuzz bridges the correlations between two system calls when the parameter name of a system call equals the name of another system call's return variable. In this way, when a system call is identified from the description of CVE or git log, any correlated system calls could also be included. Note that this operation may map one keyword to several system calls. Once this situation happens, SemFuzz selects the system call that can cover the most keywords. For example, the first parameter of the system call `sendto` is `sockfd`, which equals the name of the return variable of the system call `socket` and the system call `accept`. Compared

[5] One may think about getting system call prototype from the header files. However, header files do not have information about system call dependencies and the values of parameters and their relations. For example, MSG_MORE can only be used in `sendto` after calling `socket` (to establish connection). Such information cannot be found from header files.

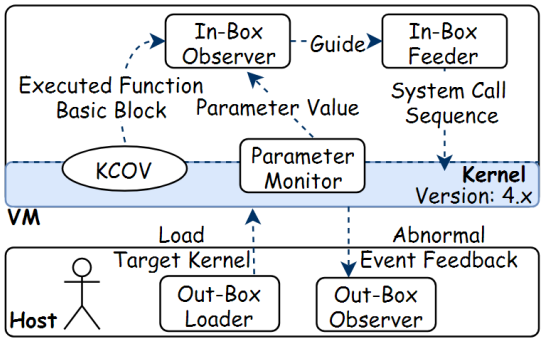


Figure 4: Setting up testing environment.

with `accept`, `socket` can cover more keywords (e.g., udp). In that case, when `sendto` is retrieved, `socket` should also be extracted.

Using this approach, SemFuzz automatically analyzes 1082 LPM pages, and correlates 373 system calls with more than 2000 keywords, which is five more times than only using system call names as the keywords. From our evaluation of 112 CVEs, SemFuzz can successfully retrieve the system calls for 96 (86%) of them for further fuzzing.

5 SEMANTICS-GUIDED FUZZING

SemFuzz extracts necessary information, or *guidances*, from non-code text in CVE and Linux git log, to guide the fuzzing process. Particularly, the retrieved "affected version" helps SemFuzz to set up the right testing environment. Then SemFuzz generates the first input (i.e., the seed input) using the retrieved "system calls". In the fuzzing process, SemFuzz performs a coarse mutation on the inputs to find a system call sequence that can move the execution towards the "vulnerable functions". After that, SemFuzz continues to perform a fine-grained mutation on the system call sequence by monitoring the "critical variables", until the target vulnerability is found to be triggered, according to the signs of the attack result specified by the "vulnerability type".

5.1 Setting up the Testing Environment

The tasks of setting up the testing environment include: running the vulnerable Linux kernel version and observing the execution status of the Linux kernel. The first task is to load a vulnerable Linux kernel and make a sequence of system calls. As we know, a kernel cannot load itself, so we build a Linux kernel inside a virtual machine and let SemFuzz load it from outside (i.e., on the host machine), as shown in Figure 4. In particular, we pre-build 103 Linux kernel versions to avoid the redundant building for time saving. When the "affected version" is retrieved from the CVE description, the out-box loader of SemFuzz loads the corresponding version. Then the in-box feeder of SemFuzz feeds the target kernel with a sequence of system calls. Such feeding can be achieved by a user-level application installed in the Linux operating system (inside the virtual machine). What the application does is to invoke system calls according to the given system call sequence.

The second task is to observe the execution status of the kernel, including the executed functions, values of critical variables and the abnormal events of the kernel, which is essential for feeding back to SemFuzz as the further guidance of the input mutation. Similar to the first task, SemFuzz works both inside and outside the virtual machine. (a) To monitor the executed functions, an in-box observer of SemFuzz leverages system support called `KCOV` (kernel code coverage) [3], which is designed to track the executed code in Linux kernel. (b) Tracking the critical variables in kernel is much more complicated than expected. One idea is to add instrumentation code around the variable in the source code. When the code runs, it gives the value of the variable. However, this approach is not flexible since the kernel has to be recompiled each time when SemFuzz needs to observe a different variable. One may also think of dynamic instrumentation which, in runtime, locates the target variable and instruments code around it. However, such variable may not be locatable since it would get optimized out when the kernel is compiled. To solve this problem, our basic idea is to observe the parameters of a kernel function instead of the critical variables. Compared with a variable inside a kernel function, function parameters will not be optimized out. In detail, we statically perform backward intra-procedure data-flow and control-flow analysis on the critical variables, trying to find the parameters that the critical variables depend on. For example, in Figure 2, the variable *skb.len* is the critical variable retrieved from CVE, which data depends on the function parameter *skb*. So SemFuzz dynamically instruments the parameter *skb* in the function *ip_cmsg_recv_checksum*. When the function is invoked, the value of the parameter can be obtained. (c) To capture an abnormal event (e.g., memory corruption) of the kernel, SemFuzz lets the out-box observer watch the kernel outside the virtual machine. Once an abnormal event happens, SemFuzz will be alerted.

In our implementation, we built SemFuzz based on the state-of-the-art Linux system call fuzzer, called Syzkaller [4]. Regarding the in-box observer, Syzkaller can directly call the API of `KCOV` and gets the execution status of the kernel. Besides this, Syzkaller can perform the fuzzing by randomly adding, removing or changing a system call together with its parameter(s) in the sequence. We implemented the seed input generation and the strategy of mutating the seed input according to the non-code text in CVE and Linux git log (see Section 5.2, 5.3 and 5.4), which greatly improves the performance of fuzzing for over 1.6 times (see Section 6.3). As for out-box observer, Syzkaller monitors whether the kernel crashes or hangs, and also checks the output of the internal kernel error detectors (e.g., `KASAN` for detecting memory errors and `UBSAN` for detecting undefined behaviors such as integer overflow). SemFuzz retrieves the report of Syzkaller to check whether the behavior specified by the target vulnerability type occurs (e.g., KASAN generates a bug report "KASAN: Double free or freeing an invalid pointer", indicating the double free vulnerability is triggered).

5.2 Generating Seed Input

Before starting the fuzzing process, an initial seed input should be generated. A good seed input can move the execution of the target kernel closer to the vulnerable function, improving the performance of the later mutation process. Different from randomly generating a seed input by traditional fuzzing approaches, SemFuzz leverages "system calls" retrieved from CVE and git log. Although such system

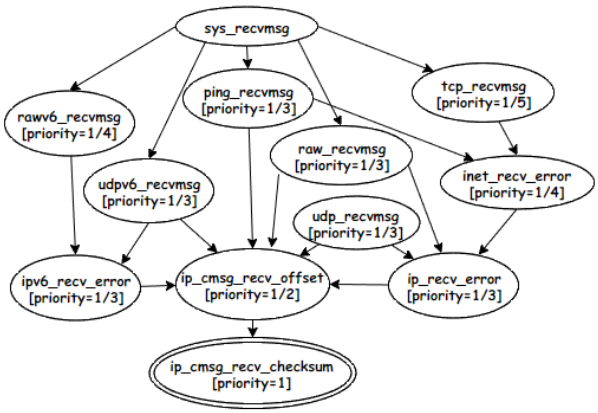


Figure 5: Reverse call graph of *ip_cmsg_recv_checksum*. For brevity, we only present the important functions in the graph.

calls are highly related to the vulnerability, they are insufficient for building the whole input to trigger the vulnerability by themselves. Thus, other necessary information should also be included to build a suitable seed input. For example, using the description of CVE-2017-6347, SemFuzz generates the system calls `socket` and `sendto`. However, between executing `socket` and `sendto`, the system call `bind` must be executed to assign a local socket address to the socket. Particularly, the seed input is built through the following two steps. Firstly, all the retrieved system calls (along with the retrieved values of parameters) are put together as an incomplete seed input. If the parameter is a structure, we fill each field in the structure. For enumeration fields, we fill them with the related enumeration values learned from LPM. For other fields, we populated them with random values compatible with their types. Secondly, SemFuzz correlates other system calls with the retrieved ones, and puts them into the seed input. As mentioned in Section 4, SemFuzz correlates two system calls if one's parameters are returned from another one. Here, we further extend such correlation to the system calls that share the same type of system resources (e.g., files, sockets). In this way, all the related system calls are put together in the seed input. Although this may bring some useless system calls to trigger the vulnerability, it increases the probability to hit the vulnerable functions.

5.3 Coarse-level Mutation

The goal of this step is to generate an input that could let the execution reach the vulnerable function. Different from symbolic execution that generates inputs by solving the constraints on the path from the program start to the vulnerable function, fuzzing achieves this in a "mutate-and-check" way, that is, continuously mutating inputs and checking whether the vulnerable function is reached. In this way, a good strategy of mutation can greatly increase the fuzzing speed. Our idea is to leverage the guidance "system calls" and "vulnerable functions". We start with the seed input and mutate it for fuzzing. We refer to each running using an input as a *fuzzing instance*. Then for each instance, we observe the execution of Linux kernel through the in-box observer, and measure the *distance* between the vulnerable functions and the execution trace of the fuzzing instance. The input corresponding to the shortest distance is chosen as a new seed input for another round of fuzzing until any vulnerable function is reached.

Formally speaking, let VUL be the set of vulnerable functions. Given a vulnerable function $f \in VUL$, we construct its reverse call graph by performing backward reachability analysis on f. Particularly, we modify GCC to collect call information during kernel compilation. We consider function pointers but ignore callbacks [6]. The caller is linked to all candidate callees having the same prototype specified by the function pointer under a path-insensitive and context-insensitive pointer analysis. In the generated graph, each node represents a function that can reach f, and each edge connecting two nodes n_1 and n_2 means that n_2 is directly called by n_1. We define the distance of two nodes $dist(n_1, n_2)$ as the number of the nodes in the shortest path between n_1 and n_2. A special situation is that, when $n_1 == n_2$, the $dist$ equals to 1. Let $RCG(f)$ be the set of functions in the reverse call graph of f, $KCOV(s)$ be the set of functions that are executed in a fuzzing instance using the input s. For $\forall g \in KCOV(s)$, we define its priority $prio$ to f as:

$$prio_f(g) = \begin{cases} 1/dist(g, f), & if\ g \in RCG(f) \\ 0, & otherwise \end{cases}$$

Then, the priority of the system call sequence s is defined as:

$$prio(s) = max_{f \in VUL} \{prio_f(g) \mid g \in KCOV(s)\}$$

In this way, the priority of a candidate system call sequence is represented in the range $[0, 1]$, where 0 means the system call sequence is not likely to reach the vulnerable functions, while 1 means the system call sequence reaches to the vulnerable functions.

For the case of CVE-2017-6347, the vulnerable function is *ip_cmsg_recv_checksum*, whose reverse call graph is shown in Figure 5. Consider two system call sequences s_1 and s_2. The former one s_1 can reach the function *udp_recvmsg*, whose distance to the vulnerable function is 3, hence $prio(s_1) = 1/3$. The latter one s_2 can reach the function *inet_recv_error*, whose *distance* to the vulnerable function is 4, hence $prio(s_2) = 1/4$. By comparing $prio(s_1)$ and $prio(s_2)$, we prefer to choose s_1 instead of s_2 as the candidate input for further mutation.

5.4 Fine-grained Mutation

After finding an input I that lets the kernel run the vulnerable function, SemFuzz continues to mutate the input with the feedback from monitoring the "critical variables". Basically, SemFuzz does not add new system calls or delete any existing system calls in I. What SemFuzz does is to mutate the values of the system call parameters, and to repeat existing system calls, which is then called fine-grained mutation. Regarding observing the "critical variables", recall that the in-box observer cannot directly monitor the changes of critical variables (Section 5.1). So SemFuzz only observes the function parameters that the critical variables depend on, to check whether the values are impacted when a given input is executed.

Similar to the coarse-level mutation, SemFuzz needs to determine whether an input is "better" than others. If so, the input will be

[6] This may cause some links between function calls missing.

```
1 int r0 = syscall(__NR_socket, AF_INET, SOCK_DGRAW, 0);
2 syscall(__NR_setsockopt, r0, ......);
3 syscall(__NR_bind, r0, lookback,....)
4 int r1 = syscall(__NR_socket, AF_INET, SOCK_DGRAW, 0);
5 syscall(__NR_sendto, r1, NULL, 0, MSG_MORE, loopback);
6 syscall(__NR_sendto, r1, buff, buff_size, 0, loopback);
7 syscall(__NR_poll, .....);
8 syscall(__NR_recvfrom, r0, ..., loopaddr);
```

Figure 6: The sequence of system calls to trigger CVE-2017-6347.

selected and used for future mutations. We measure the input quality using the *distance* between basic blocks. As we know, a basic block is a sequence of instructions with no branches in except to the entry and no branches out except at the exit [25]. A function can be represented as a control flow graph, in which each node is a basic block and each edge between two basic blocks indicates there is a control flow relationship between them. We measure the distance between two basic blocks (b_1 and b_2) by the number of basic blocks in the shortest path from b_1 to b_2, denoted as $dist_B(b_1, b_2)$. A special situation is that, when $b_1 == b_2$, the $dist_B$ equals to 1. In this way, we can define the priority of an input as follows.

Given a vulnerable function $f \in VUL$ with an entry point e. Suppose the patched code of function f is in the set of basic blocks $PATCH = \{p_1, p_2, ..., p_n\}$. Let $KCOV_B(s)$ be the set of covered basic blocks in a system call sequence s. For $\forall b \in KCOV_B(s)$, we define its priority $prio'$ to a patched block $p \in PATCH$ as:

$$prio'_p(b) = \begin{cases} \frac{dist_B(e,b) \times 100}{dist_B(e,b) + dist_B(b,p) - 1}, & if\ e \rightsquigarrow b\ and\ b \rightsquigarrow p \\ 1, & otherwise \end{cases}$$

where $e \rightsquigarrow b$ means there is a path from the entry e to the basic block b, and $b \rightsquigarrow p$ means there is a path from the basic block b to patched basic block p. The priority of the system call sequence s is defined as:

$$prio'(s) = max_{f \in VUL, p \in PATCH} \{prio'_p(b) \mid b \in KCOV_B(s)\}$$

From this definition, the priority of a given system call sequence can be represented in the range [1, 100], where 1 means the patched code is less likely to be reached, while 100 means the patched code is reached. The higher the value, the more likely that the patched code could be reached. SemFuzz chooses the input that has the highest value for further fuzzing.

We take CVE-2017-6347 as an example. The critical variable of the vulnerable function *ip_cmsg_recv_checksum* is *skb.len*, where *skb* is the parameter of the function. Consider two system call sequences s_1 and s_2. The former one can reach a basic block whose distance to the function entry is 95 in a path with distance 100 (from the function entry to the patched basic block). Then the $prio'(s_1) = (95/100) \times 100 = 95$. The latter one s_2 can reach a basic block whose distance to the function entry is 90 in the same path. Then the $prio'(s_2) = (90/100) \times 100 = 90$. Thus, we prefer to choose s_1 instead of s_2 for fine-grained mutation. We randomly select a mutation operation on s_1 to generate a new system call sequence s'_1. When executing s'_1, we observe the *len* field of the *skb* parameter. If its value is changed, we prioritize the selected operation in further mutations. Figure 6 shows a system call sequence in the fined-grained mutation process. For brevity, we omit some tedious parameters in the figure, and use *loopback* to present the *sock_addr* structure assigned with the loop back address (i.e., 127.0.0.1). The feedback from the in-box observer indicates that by mutating the parameter *buff* of the system call `sendto` (line 7), the *skb.len* will be impacted. So SemFuzz focuses on such a mutation and finally triggers the vulnerability when the *buff* argument is fulfilling with more than 512 bytes data.

6 EVALUATION AND FINDINGS

In this section, we evaluate the effectiveness and performance of SemFuzz. Also included is the evaluation on the accuracy of the retrieved information using natural language processing, which supports the high performance of SemFuzz. Interestingly, two unknown vulnerabilities were found in this process, which were presented in the case study. We demonstrate that the leaked information from CVE and Linux git log could help attackers to automatically generate PoC exploits.

6.1 Settings

CVEs and Linux git logs. We collected the CVEs in last five years that target x86/x86_64 Linux kernel from version 4.0 to the latest version 4.11. We filtered out the vulnerabilities that require specific devices to trigger (e.g., CVE-2016-2782 requires a specific USB device that lacks a bulk-in or interrupt-in endpoint) and the logical vulnerabilities whose abnormal behaviors cannot be directly observed (e.g., bypass the intended file system access restriction as shown in CVE-2015-8660). Finally, we got 112 CVEs, which covers the most common vulnerability types (e.g., double free, use-after-free, buffer over-read and buffer overflow, etc., as shown in Table 1) and various subsystems (e.g., networking, file system, keys, etc.) of the Linux kernel. For each CVE, we collected the corresponding Linux git log, including the log message and patching code.

We compiled 103 versions of Linux kernel from 4.0 to 4.11 with the `allyesconfig` configuration while opting out the test modules that may crash the kernel. To support the code coverage collection, we enabled the `KCOV` functionality, which was introduced into the kernel since version 4.6. We ported the `KCOV` functionality to the old kernels before 4.6. We also enabled internal kernel error detectors (e.g., `KASAN` detector and the UBSAN detector) to enhance the ability of SemFuzz in capturing abnormal behaviors.

Computing environment. All the experiments were run on a 64-bit Ubuntu server with 40 cores (2.3GHz Intel®Xeon®CPU E5-2650), 256GB memory and 70TB hard drive. For each CVE, the fuzzing process continued for 48 hours or until the given vulnerability is successfully triggered.

6.2 Effectiveness

Effective of exploit generation. We evaluated the number of CVEs from which SemFuzz successfully generates the PoC exploits, indicating the effectiveness of this end-to-end approach. Further, we made a deep analysis on the rest CVEs to check why SemFuzz does

Table 2: Precision and recall of semantic information retrieving.

Information	TP	FP	FN	TN	Precision	Recall
Affected Version	112	0	0	0	100.00%	100.00%
Vuln. Type	112	0	0	0	100.00%	100.00%
Vuln. Functions	95	16	0	1	85.59%	100.00%
Critical Variables	33	7	5	67	82.50%	86.84%
System Calls	70	13	3	26	84.34%	95.89%

not trigger the vulnerabilities. This gives us further understanding on both our approach and the descriptions of CVEs/Linux git logs.

SemFuzz successfully generated PoC exploits for 18 (16%) CVEs, including use-after-free, null pointer deference, buffer over-read, etc. The details of these 18 CVEs are shown in Table 4 in Appendix. The affected Linux versions and vulnerable functions are correctly pointed out. Interestingly, we note that only 5 of the 18 CVEs have been studied and the corresponding exploits were released on the Internet, which shows that most of the exploits are not generated or only owned by very few attackers. We then compared the exploits from Internet and ours. Actually, we find the sequences of system calls between them are not the same. Neither the values of the parameters. One reason is that the vulnerable function can be triggered in different execution traces.

For the rest 94 CVEs, we analyzed the intermediate results. We found 49% (46/94) of them give correct inputs that can lead the execution to the vulnerable functions. 20% (19/94) of them give correct inputs that can even let the execution run to the patched basic block in the vulnerable functions. We manually examined why SemFuzz does not generate PoC exploits for these CVEs and found that it is mainly due to two reasons. Firstly, some vulnerabilities can only be triggered when the inputs (especially the parameters of system calls) meet some specific conditions, which is hard for SemFuzz to generate in limited time. Secondly, some vulnerabilities are only possible if race conditions can occur. Still SemFuzz needs more time to trigger such conditions due to non-determinism of concurrent executions. Researches on augmenting fuzzing through selective symbolic execution [52] and manipulating thread scheduling [29, 58] could help further improve the performance of SemFuzz.

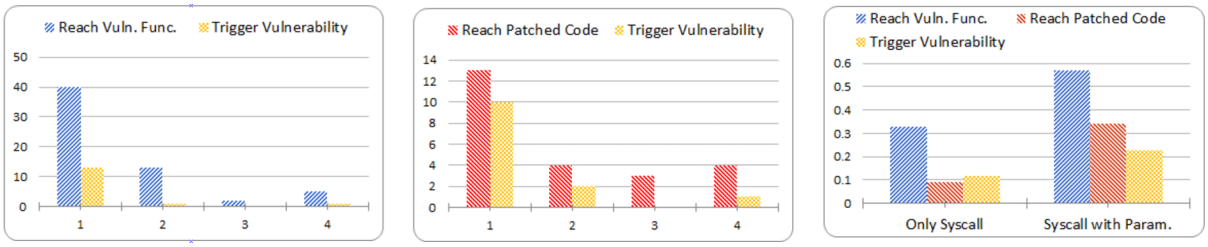
Effective of semantic information retrieving. In this part, we measured whether the extracted semantic information is accurate and whether there is any important information missed by our natural language processing. This evaluation was performed for each guidance. In general, we compared the retrieved information with manually retrieved ones from the descriptions of CVEs and Linux git logs, and computed the precision and recall. Details are elaborated in Table 2.

For "affected version" and "vulnerability type", we found the extracted information is very accurate, with 100% precision and 100% recall. That is, our NLP-based approach can correctly retrieve the affected version and vulnerability type from all the CVEs. For the "vulnerable functions", SemFuzz can always find them by comparing the patched code with the original one (i.e., with 100% recall). Considering some of the identified "vulnerable functions" may not be related to the vulnerable code, we manually checked them on such relationship. The results show that 85.59% of the identified vulnerable functions have relation to the vulnerable code. For the "critical variables", we evaluated our NLP tool (for extracting the variables) by manually comparing the descriptions with the variables identified. The results show that the precision is 82.5% and the recall is 86.84%, which means our NLP-based approach can provide precise guidance of the critical variables.

Table 3: Performance evaluation.

CVE	Reach Vuln. Function		Trigger Vulnerability	
	SemFuzz	Syzkaller	SemFuzz	Syzkaller
CVE-2015-0275	0.12 h	>48.00 h	3.31 h	>48.00 h
CVE-2015-1333	2.73 h	8.68 h	8.17 h	37.26 h
CVE-2015-5706	0.07 h	0.29 h	0.10 h	>48.00 h
CVE-2015-6937	9.18 h	16.33 h	11.64 h	>48.00 h
CVE-2015-7872	4.16 h	6.11 h	12.32 h	27.61 h
CVE-2015-7990	3.29 h	9.82 h	4.72 h	21.54 h
CVE-2016-0728	0.06 h	0.39 h	6.97 h	42.81 h
CVE-2016-10147	5.63 h	17.89 h	31.96 h	>48.00 h
CVE-2016-3134	1.92 h	>48.00 h	29.35 h	>48.00 h
CVE-2016-3841	0.03 h	0.18 h	9.44 h	>48.00 h
CVE-2016-4482	0.01 h	>48.00 h	0.04 h	>48.00 h
CVE-2016-4794	0.08 h	0.17 h	5.51 h	26.84 h
CVE-2016-6213	0.11 h	2.03 h	16.53 h	>48.00 h
CVE-2016-8646	3.56 h	8.87 h	38.29 h	>48.00 h
CVE-2016-9555	0.47 h	>48.00 h	23.16 h	>48.00 h
CVE-2016-9793	0.01 h	0.14 h	17.05 h	>48.00 h
CVE-2017-6074	1.13 h	1.65 h	10.91 h	39.12 h
CVE-2017-6347	0.10 h	0.42 h	7.76 h	41.83 h

To measure the precision and recall of "system calls", for each vulnerability, we first manually analyze its CVE and git log descriptions to extract the necessary system calls (for triggering the vulnerability) as the ground truth (S1). Then we compare whether the retrieved "system calls" by our NLP tool (S2) consist with S1. For a system call in S2 but not in S1, we treat it as a false positive. If a system call is correctly identified but with wrong parameter value, we also treat it as a false positive. On the other hand, if a system call is in S1 but not in S2, we treat it as a false negative. Also, if a system call is correctly identified but the parameter value specified in S1 is not retrieved in S2, we treat it as a false negative. Based on this, the precision is 84.34% and recall is 95.89%. After further analyzing the descriptions, we found that the main reason of false positive is that the names of some system calls are ambiguous. For example, CVE-2016-7915 says "by connecting a device". SemFuzz mistakes the word "connecting" as a `connect` system call. However, the verb "connect" here actually does not indicate the `connect` system call since the object of the verb is "a device", which is irrelevant with network communication. The main reason of false negative is the missing of connection between the value of parameter and its description. For example, CVE-2015-8539 says "via crafted keyctl commands that negatively instantiate a key". From the semantics of "negatively instantiate a key", an experienced analyst is able to infer that the value of the parameter `operation` in the system



(a) Impact of "vulnerable functions". (b) Impact of "critical variables". (c) Impact of "system calls".

Figure 7: Impact of different guidances.

call `keyctl` is `KEYCTL_NEGATE`. SemFuzz does not build the connection between the keyword `KEYCTL_NEGATE` and its description "negatively instantiate a key", which could be identified once the description is parsed in the future.

6.3 Performance

We measured the performance of SemFuzz and compared it with the official version of Syzkaller [4], a state-of-art Linux system call fuzzer. The default configuration of Syzkaller is to randomly select system calls to test. To be fair, for each vulnerability, we configure Syzkaller to test the system calls correlated to the vulnerable subsystem. For example, if the vulnerability is in the networking subsystem, we enable Syzkaller to test all network-related system calls.

Table 3 gives the results. For the 18 CVEs, the average time for SemFuzz to generate a PoC exploit is 13.2 hours. As a compassion, Syzkaller can only generate exploits for 7 CVEs. Further, we analyzed the time consumption in each stage in the fuzzing process. The time for SemFuzz to extract information and prepare for the first seed input is less than 0.1 seconds, which is negligible compared to the later fuzzing process. For Syzkaller, it does not need this step. Then we measured the time needed to generate the first input that could let the execution reach the vulnerable function. On average, it takes SemFuzz 1.8 hours in this step. Syzkaller can only generate such inputs for 14 CVEs within the time limit, and the average time is 5.2 hours, which is 1.9 times slower than SemFuzz. The last step is to mutate the input to trigger the vulnerabilities, which takes SemFuzz 13.2 hours. Still, Syzkaller can only handle 7 vulnerabilities with the average time of 33.9 hours (1.6 times slower than SemFuzz). From the analysis, we found our coarse-level and fine-grained mutation strategies are very helpful, mainly due to the semantic information from CVEs and git logs.

6.4 Findings

We explored to understand the relationship between guidances and the PoC exploits. To this end, we studied the existence of guidances and its impact on the automatic PoC exploit generation of SemFuzz. This understanding helps to release a CVE and git log without facilitating potential attackers to automatically generate exploits. We also found two *unknown* vulnerabilities in the fuzzing process, which have already been confirmed by the Linux kernel developer group.

Guidances and the exploits. For the five guidances, "affected version" and "vulnerability type" can always be found in CVE and git log. Hence, we focus on the rest three guidances (i.e., the vulnerable functions, critical variables and system calls). Figure 7 (a)-(c) shows the impact of each guidance.

Figure 7(a) shows how the "vulnerable functions" affects the exploit generation. Interestingly, more vulnerable functions decrease the possibility to generate a vulnerability. Note that vulnerable functions are those touched by a patch. However, likely only one of them is exploitable. In other cases, these functions may need to be invoked in a certain order to reach the vulnerability. Either way, the problem here is more complicated (than a single vulnerable function), making the underlying vulnerability harder to trigger. For the 18 CVEs from which SemFuzz generates PoC exploits, 13 of them have only one vulnerable function. The mutation processes of them are more concentrated.

Figure 7(b) demonstrates how the "critical variables" affects the exploit generation. In the five cases (i.e., CVE-2015-5706, CVE-2016-0728, CVE-2016-6213, CVE-2016-8646, and CVE-2016-9555), when the critical variable is missing, the PoC exploits can still be generated. When there is only one critical variable, the number of generated PoC exploits is maximized. More critical variables will decrease the number of PoC exploits, similar to our observations of "vulnerable functions".

Figure 7(c) gives the ratio of successful exploits using only system calls and using both system calls and the values of their parameters. From the figure, we find that the success rate is higher by using both system calls and the values of their parameters than by only using the system calls alone. We further compare the time to generate PoC exploits between the two situations, and find that using system calls together with values of parameters are faster, mainly due to that there is no need to mutate the parameters to find the correct parameter values.

To summarize, the three guidances ("vulnerable functions", "critical variables" and "system calls") are necessary for generating an exploit. To release a CVE in a way which is less likely for attackers to generate exploits, one idea is to decrease the number of guidances, such as only disclosing the related system calls but not the specific values for triggering the vulnerability. Another idea is, on the contrary, to increase the number of "vulnerable functions" and "critical variables" by mentioning more related functions and variables in the description, which can confuse attackers and further to impede the automatic exploit generation. Considering CVE is

```
1 syscall(__NR_mmap, 0x20000000ul, 0xf000ul, 0x3ul,
2         0x32ul, 0xffffffffffffffff, 0x0ul);
3 *(uint32_t *)0x20000000 = (uint32_t)0x6;
4 *(uint32_t *)0x20000004 = (uint32_t)0x4;
5 *(uint32_t *)0x20000008 = (uint32_t)0x54d1;
6 *(uint32_t *)0x2000000c = (uint32_t)0xc93;
7 syscall(__NR_bpf, 0x0ul, 0x20000000, 0x10ul,
8         0, 0, 0);
```

(a) The sequence of system calls to trigger CVE-2016-4794.

```
1 syscall(__NR_mmap, 0x20000000ul, 0x2000ul, 0x3ul,
2         0x32ul, 0xffffffffffffffff, 0x0ul);
3 *(uint32_t *)0x20000000 = (uint32_t)0x6);
4 *(uint32_t *)0x20000004 = (uint32_t)0x4);
5 *(uint32_t *)0x20000008 = (uint32_t)0x7);
6 *(uint32_t *)0x2000000c = (uint32_t)0x1001000);
7 *(uint32_t *)0x20000010 = (uint32_t)0x0);
8 syscall(__NR_bpf, 0x0ul, 0x20000000ul, 0x14ul);
```

(b) The sequence of system calls to trigger the zero-day vulnerability.

Figure 8: The proof-of-concept exploits of CVE-2016-4794 and the zero-day vulnerability.

designed to identify and catalog vulnerabilities in software [23], such mitigation may not affect the usefulness of CVE descriptions.

Unknown vulnerabilities. SemFuzz found one zero-day vulnerability and one undisclosed vulnerability when fuzzing related ones. For the zero-day vulnerability, it appears around the known flaws. For the undisclosed vulnerability, we verified that they are similar problems inside equivalent components. We show them in the case study (Section 6.5). This demonstrates the possibility of using SemFuzz to help find similar but unknown vulnerabilities in the process of fuzzing known vulnerabilities.

6.5 Case Study

We have demonstrated how SemFuzz works using CVE-2017-6347 as an example in previous sections. In this subsection, we will demonstrate the generation of PoC exploits for a zero-day and an undisclosed vulnerabilities, and their connections with the known vulnerabilities.

The zero-day vulnerability. We discovered a zero-day vulnerability in the fuzzing process of CVE-2016-4794, a use-after-free vulnerability in the Berkeley Packet Filter (bpf) subsystem. From the CVE description, we can learn that the vulnerability could be triggered using the `mmap` and `bpf` system calls. From the corresponding patch code, we found multiple functions that are patched. By filtering out those functions not mentioned in CVE or git log, we got two vulnerable functions: *pcpu_need_to_extend* and *pcpu_alloc*.

In the fuzzing process, SemFuzz continuously generates system call sequences that can let the execution run towards *pcpu_alloc* or *pcpu_need_to_extend*, and finally trigger the vulnerability. In a fuzzing instance, we found that a mutated system call sequence enters the *pcpu_need_to_extend* function and calls the *free_percpu* function, which makes CPU stall (in dead lock status). We discovered this new vulnerability in Linux kernel version 4.6. We further checked other versions of Linux kernel and found this new vulnerability still exists in the latest Linux kernel version 4.11. We reported this vulnerability to the Linux kernel developer group and they confirmed our finding [7].

Figure 8(a) presents the PoC exploit of CVE-2016-4794 and Figure 8(b) presents the PoC exploit of the zero-day vulnerability. We can see that they share the same system call sequence (i.e., `mmap` and `bpf`) with different parameter values (e.g., the 2nd parameter of `bpf` is filled with different values). From this case study, we find that new bugs tend to appear around the known flaws and can be triggered with similar inputs after some mutations.

The undisclosed vulnerability. We discovered an undisclosed vulnerability when analyzing the flaw reported by CVE-2016-3841, a use-after-free vulnerability in the networking subsystem. From the CVE description, we can learn that the vulnerability could be triggered by using the `socket` and `sendmsg` system calls. From its corresponding patch code, we found multiple functions that are patched. Even after filtering using CVE and git log descriptions, there are still 18 vulnerable functions. After successfully generating a PoC exploit, SemFuzz finds that the vulnerability is triggered in the function *udpv6_sendmsg*.

In the fuzzing process, SemFuzz mutates parameters values of the system call `socket` (i.e., `domain` and `protocol`). In a certain fuzzing instance, the `domain` is set to AF_INET (standing for ipv4) and the `protocol` is set to PROT_ICMP (standing for the ICMP protocol). Then another memory-related vulnerability (i.e., null pointer deference) is triggered in another function *ping_v4_sendmsg*. We have reported this vulnerability to the Linux kernel developer group [8] and they told us that this bug has been patched in the latest Linux kernel before we reported to them. However, we could not find any publicly released report referring to this bug. So it is considered to be an undisclosed vulnerability.

7 DISCUSSION

Vulnerability type. In this study, although SemFuzz is able to handle 16 types of vulnerabilities including double free, use-after-free and memory corruption, etc. However, we do not consider the vulnerabilities that require specific devices to trigger and the logical vulnerabilities whose abnormal behaviors cannot be directly observed by SemFuzz. Actually, such limitations are mostly due to the fuzzer, but not the semantics-based approach. On the contrary, the semantic information is really helpful (e.g., for a human analyst to generate exploits). In particular, to fuzz the device-specific vulnerabilities, one can emulate the required device under the support of visualization. The information retrieved from the vulnerability description can be used to guide the emulation of the required device. For example, CVE-2016-2782 requires a specific USB device that lacks a bulk-in or interrupt-in endpoint. With the description about the target USB device, we can leverage vUSBf [21], a USB device fuzzer to emulate various USB devices that meet these requirements. For the logical vulnerabilities, the description about the violations can be used as guidances for constructing detector under the basic understanding of the target subsystem. For example, CVE-2015-8660 describes a logical vulnerability which allows an attacker to bypass intended access restrictions and to modify the attributes of arbitrary overlay files. With such description, as

well as the knowledge of the file system, we can build a detector to check the behavior of file attribution modification in the overlay file system.

Vulnerable targets. In this study, we choose Linux kernel as the target. Although the approach presented in this paper is specific to the Linux kernel, the basic idea of leveraging semantic information to guide fuzzing can be applied to other programs (even without source code), as long as we can obtain enough guidances. For example, CVE-2017-3053 (Adobe Reader) discloses affected versions (e.g., 11.0.19 and earlier), vulnerability type (i.e., memory address leak), vulnerable component (i.e., image conversion engine), and trigger conditions (i.e., parsing the APP13 segment in JPEG files). These information can be leveraged to guide a file format fuzzer to trigger the vulnerability by mutating the JPEG APP13 segment and monitoring whether memory address leak occurs in the image conversion engine. Besides CVE, we can also retrieve more information from other sources of vulnerability descriptions.

Sources of vulnerability descriptions. In this study, we retrieved guidances from both the CVE description and the Linux git logs. In our experiments, sometimes, we found that only one source of vulnerability descriptions is enough to support the necessary guidances. We take the vulnerability description with the commit ID "2b95fda2c4fcb6d6625963f889247538f247fce0" in the Linux git log as an example. It is a kernel crash report of a double free vulnerability. From the log, we can retrieve the vulnerable function (i.e., *x509_free_certificate*) and the critical variable (i.e., *cert->pub->key*). Also, the log presents the call trace of the vulnerable function, from which we can retrieve the triggering system call (i.e., `add_key`). Based on the observation, we found that the semantic information may be obtained from one source, and can also be combined with the other sources for verification, which could help to increase the performance of fuzzing. In addition, we also found that besides CVE and git log, there may be other sources of vulnerability descriptions, which is also helpful for guidance generation. For example, the source from FullDisclosure [2] contains the vulnerability information of CVE-2016-8655, which helps SemFuzz to get extra guidances that are not disclosed by CVE and git log, such as "we can reach packet_set_ring() by calling setsockopt() on the socket using the PACKET_RX_RING option". We will try to crawl more descriptions from other sources in future work.

8 RELATED WORK

Automatic exploit generation. Brumley et al. [28] proposed an automatic approach to construct proof-of-concept exploits for input-validation vulnerabilities by seeking an input that fails the newly added checks in the corresponding patches. Avgerinos et al. [26] developed a technique to generate exploits for control flow hijacking attacks by modeling them as formal verification problems on the inputs. Hu et al. [34] developed a data-flow stitching technique for systematically finding ways to join data flows in the program to generate data-oriented exploits. In addition, there are some other studies, such as Chainsaw [24] and CraxWeb [35], targeting on automatically generating exploits for SQLI and XSS attacks on web applications. On the mobile platform, Centaur [39] leverages symbolic execution of Android framework for vulnerability discovery and exploit generation. All of them rely on generating constraints on inputs and leverage symbolic execution to further solve the constraints which let the program run to the vulnerable functions. By solving the constrains, an exploit could be generated. However, as mentioned in the introduction, symbolic execution and constrains solving have difficulties in exploiting deep program flaws on complicated target programs (e.g., non-linear constraints, multiple threads). Our study leverages semantics-based fuzzing to support more types of vulnerabilities. Also, using the semantic information retrieved from non-code text, the fuzzing process can be well guided for achieving high performance.

Guided fuzzing. Traditional fuzzing without guiding suffers from redundant executions [44]. That is, an execution instance lets a program run to the same status as previous instances do, which greatly reduces the performance of fuzzing. To solve this problem, previous studies guide the fuzzing process using the running status of a program and the format of inputs. For example, Skyfire [56] uses a data-driven seed generation approach, which leverages the knowledge learned from the vast amount of existing samples to generate well-distributed seed inputs for fuzzing. VUzzer [49] and AFLFast [27] leverage static control-flow and data-flow analysis to prioritize deep paths and de-prioritize frequent paths when mutating the inputs. ArtFuzz [32] dynamically discovers likely memory layouts to guide the fuzzing process. Our study demonstrates that, in addition to the running status of a program, the non-code descriptions in CVE and git log also help to avoid redundant runs.

Semantics-based program analysis. Semantic information is not the first use in program analysis. iComment [54] leverages NLP to analyze program comments and compares the semantics of comments with the code to determine the inconsistency, which indicates a bug or a bad comment. aComment [55] extracts annotations from both code and comments written in natural language to detect interrupt-related concurrency bugs. In addition to code comments, developer documents are also a good source for mining knowledge to assist program analysis (e.g., constructing API models [59], extracting security policies [57] and inferring resource specifications [60]). On the mobile platform, some recent studies apply NLP to understand app descriptions for checking whether the app requests unnecessary permissions [47] or performs unexpected behaviors [33]. Also, some API's semantics are analyzed for mapping libraries cross platforms [31]. Different from these semantics-based program analysis studies, our work is the first to utilize third-party vulnerability descriptions to guide the generation of PoC exploits.

Repository analysis. In recent years, various code mining approaches [37, 38] have been proposed to automatically extract implicit programming rules from source code repositories. In addition, some studies mine software repositories to predict vulnerabilities. Neuhaus et al. [46] use the vulnerability database of the Mozilla project to predict the vulnerable software components. Meneely et al. [41–43] conduct the research on the correlation between meta data in code repositories (e.g., code churn, lines of code, number of reviewers, etc.) and the reported vulnerabilities. The studies of [48, 51] mine software repositories to discover the bug-introducing or fix-inducing commits. Different from them, we use the code repositories as an information source for generating guidances for fuzzing.

9 CONCLUSIONS

In this paper, we designed and implemented a semantics-based approach for automatic generation of proof-of-concept exploits. Making this end-to-end approach feasible is the intelligent fuzzing technique guided by the automatically recovered vulnerability-related knowledge from non-code text reports. Running SemFuzz over 112 Linux kernel vulnerabilities, 18 of them have been automatically triggered. Interestingly, SemFuzz also found a zero-day vulnerability and an undisclosed one. Our research extends the automatic exploit generation for simple input-validation vulnerability proposed 10 years ago to much more complicated vulnerabilities including uncontrolled resource consumption, dead lock, memory corruption, etc. More importantly, our research gives new insights on the way that vulnerability-related information is shared today.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their constructive comments. Also, the authors would like to express their thanks to Dmitry Vyukov for his Syzkaller tool and his help in solving the issues regarding Syzkaller. IU authors were supported in part by NSF CNS-1223477, 1223495, 1527141, 1618493 and ARO W911NF1610127. IIE authors were supported in part by NSFC U1536106 and 61728209, National Key Research and Development Program of China (Grant No.2016QY04W0805), Youth Innovation Promotion Association CAS, and strategic priority research program of CAS (XDA06010701). RUC authors were supported in part by NSFC 91418206, 61170240 and 61472429.

APPENDIX

Table 4 gives a detailed description of the 18 CVEs that can be successfully auto-exploited by SemFuzz. We present the semantic information retrieved from their CVE descriptions and git logs as guidance used for fuzzing, including the affected version, vulnerability type, vulnerable functions, critical variables and system calls.

REFERENCES

[1] 2016. 2016 Financial Industry Cybersecurity Report. https://cdn2.hubspot.net/hubfs/533449/SecurityScorecard_2016_Financial_Report.pdf. (2016).
[2] 2016. FullDisclosure: CVE-2016-8655 Linux af_packet.c race condition (local root). http://seclists.org/oss-sec/2016/q4/607. (2016).
[3] 2016. Kernel: Add KCOV Code Coverage. https://lwn.net/Articles/671640/. (2016).
[4] 2016. Syzkaller. https://github.com/google/syzkaller. (2016).
[5] 2016. Yahoo: Hackers Stole Data On Another Billion Accounts. https://www.forbes.com/sites/thomasbrewster/2016/12/14/yahoo-admits-another-billion-user-accounts-were-leaked-in-2013. (2016).
[6] 2017. Application Vulnerability: Trend Analysis and Correlation of Coding Patterns across Industries. https://www.cognizant.com/whitepapers/Application-Vulnerability-Trend-Analysis-and-Correlation-of-Coding-Patterns-Across-Industries.pdf. (2017).
[7] 2017. Bug 195709. https://bugzilla.kernel.org/show_bug.cgi?id=195709. (2017).
[8] 2017. Bug 195807. https://bugzilla.kernel.org/show_bug.cgi?id=195807. (2017).
[9] 2017. Common Vulnerabilities and Exposures. https://cve.mitre.org. (2017).
[10] 2017. Common Weakness Enumeration. https://cwe.mitre.org. (2017).
[11] 2017. CWE: Improper Input Validation. https://cwe.mitre.org/data/definitions/20.html. (2017).
[12] 2017. FullDisclosure Mailing List. http://seclists.org/fulldisclosure. (2017).
[13] 2017. Information Security Resources. https://www.sans.org/security-resources/blogs. (2017).
[14] 2017. Krebs on Security. https://krebsonsecurity.com. (2017).
[15] 2017. Linux Kernel Git Repositories. https://git.kernel.org. (2017).
[16] 2017. Linux man pages online. http://man7.org/linux/man-pages/index.html. (2017).
[17] 2017. National Vulnerability Database. https://nvd.nist.gov. (2017).
[18] 2017. pyStatParser. https://github.com/emilmont/pyStatParser. (2017).
[19] 2017. STP Constraint Solver. http://stp.github.io. (2017).
[20] 2017. Vulnerability. https://en.wikipedia.org/wiki/Vulnerability_(computing). (2017).
[21] 2017. vUSBf. https://github.com/schumilo/vUSBf. (2017).
[22] 2017. WannaCry Ransomware Attack. https://en.wikipedia.org/wiki/WannaCry_ransomware_attack. (2017).
[23] 2017. What is CVE and How Does It Work? http://www.csoonline.com/article/3204884/application-security/what-is-the-cve-and-how-does-it-work.html. (2017).
[24] Abeer Alhuzali, Birhanu Eshete, Rigel Gjomemo, and VN Venkatakrishnan. 2016. Chainsaw: Chained Automated Workflow-Based Exploit Generation. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS 2016)*. ACM, 641–652.
[25] Frances E Allen. 1970. Control Flow Analysis. In *ACM SIGPLAN Notices*, Vol. 5. ACM, 1–19.
[26] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J Schwartz, Maverick Woo, and David Brumley. 2014. Automatic Exploit Generation. *Commun. ACM* 57, 2 (2014), 74–84.
[27] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-Based Greybox Fuzzing as Markov Chain. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS 2016)*. ACM, 1032–1043.
[28] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. 2008. Automatic Patch-Based Exploit Generation is possible: Techniques and Implications. In *Proceedings of the 29th IEEE Symposium on Security & Privacy (S&P 2008)*. IEEE, 143–157.
[29] Yan Cai and Lingwei Cao. 2015. Effective and Precise Dynamic Detection of Hidden Races for Java Programs. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering, (FSE 2015)*. 450–461.
[30] Eugene Charniak. 1996. Tree-Bank Grammars. In *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI 1996)*. 1031–1036.
[31] Kai Chen, Xueqiang Wang, Yi Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Bin Ma, Aohui Wang, Yingjun Zhang, and Wei Zou. 2016. Following Devil's Footprints: Cross-Platform Analysis of Potentially Harmful Libraries on Android and iOS. In *Proceedings of the 37th IEEE Symposium on Security & Privacy (S&P 2016)*. 357–376.
[32] Kai Chen, Yingjun Zhang, and Peng Liu. 2016. Dynamically Discovering Likely Memory Layout to Perform Accurate Fuzzing. *IEEE Trans. Reliability* 65, 3 (2016), 1180–1194.
[33] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. 2014. Checking App Behavior Against App Descriptions. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, 1025–1035.
[34] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. 2015. Automatic Generation of Data-Oriented Exploits. In *Proceedings of the 24th USENIX Security Symposium (Security 2015)*. 177–192.
[35] Shih-Kun Huang, Han-Lin Lu, Wai-Meng Leong, and Huan Liu. 2013. Craxweb: Automatic Web Application Testing and Attack Generation. In *Proceedings of the 7th IEEE International Conference on Software Security and Reliability (SERE 2013)*. IEEE, 208–217.
[36] James C King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394.
[37] Zhenmin Li and Yuanyuan Zhou. 2005. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2005)*. 306–315.
[38] Bin Liang, Pan Bian, Yan Zhang, Wenchang Shi, Wei You, and Yan Cai. 2016. AntMiner: mining more bugs by reducing noise interference. In *Proceedings of the 38th International Conference on Software Engineering (ICSE 2016)*. 333–344.
[39] Lannan Luo, Qiang Zeng, Chen Cao, Kai Chen, Jian Liu, Limin Liu, Neng Gao, Min Yang, Xinyu Xing, and Peng Liu. 2017. System Service Call-oriented Symbolic Execution of Android Framework with Applications to Vulnerability Discovery and Exploit Generation. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys 2017)*. 225–238.
[40] Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. 1993. Building a Large Annotated Corpus of English: The Penn Treebank. *Computational Linguistics* 19, 2 (1993), 313–330.
[41] Andrew Meneely, Harshavardhan Srinivasan, Ayemi Musa, Alberto Rodriguez Tejeda, Matthew Mokary, and Brian Spates. 2013. When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits. In *Proceedings of the 7th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, (ESEM 2013)*. IEEE, 65–74.
[42] Andrew Meneely, Alberto C Rodriguez Tejeda, Brian Spates, Shannon Trudeau, Danielle Neuberger, Katherine Whitlock, Christopher Ketant, and Kayla Davis. 2014. An Empirical Investigation of Socio-Technical Code Review Metrics and Security Vulnerabilities. In *Proceedings of the 6th International Workshop on Social*

Software Engineering (SSE 2014). ACM, 37–44.

[43] Andrew Meneely and Oluyinka Williams. 2012. Interactive Churn Metrics: Socio-Technical Variants of Code Churn. *ACM SIGSOFT Software Engineering Notes* 37, 6 (2012), 1–6.

[44] Barton P Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (1990), 32–44.

[45] Antonio Nappa, Richard Johnson, Leyla Bilge, Juan Caballero, and Tudor Dumitras. 2015. The Attack of the Clones: a Study of the Impact of Shared Code on Vulnerability Patching. In *Proceedings of the 36th IEEE Symposium on Security & Privacy (S&P 2015)*. IEEE, 692–708.

[46] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. 2007. Predicting vulnerable software components. In *Proceedings of the 14th ACM conference on Computer and Communications Security (CCS 2007)*. ACM, 529–540.

[47] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. 2013. WHYPER: Towards Automating Risk Assessment of Mobile Applications. In *Proceedings of the 22nd USENIX Security Symposium (Security 2013)*. 527–542.

[48] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. 2015. Vccfinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS 2015)*. ACM, 426–437.

[49] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-Aware Evolutionary Fuzzing. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS 2017)*. ISOC.

[50] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of the 31st IEEE Symposium on Security & Privacy (S&P 2010)*. IEEE, 317–331.

[51] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When Do Changes Induce Fixes?. In *ACM SIGSOFT Software Engineering Notes*, Vol. 30. ACM, 1–5.

[52] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the 23nd Annual Network and Distributed System Security Symposium (NDSS 2016)*.

[53] Michael Sutton, Adam Greene, and Pedram Amini. 2007. *Fuzzing: Brute Force Vulnerability Discovery*. Pearson Education.

[54] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. iComment: Bugs or Bad Comments?. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP 2007)*. ACM, 145–158.

[55] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. 2011. aComment: mining annotations from comments and code to detect interrupt related concurrency bugs. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*. IEEE, 11–20.

[56] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *Proceedings of the 38th IEEE Symposium on Security & Privacy (S&P 2017)*. IEEE.

[57] Xusheng Xiao, Amit Paradkar, Suresh Thummalapenta, and Tao Xie. 2012. Automated Extraction of Security Policies from Natural-Language Software Documents. In *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2012)*. ACM, 12.

[58] Junfeng Yang, Ang Cui, Salvatore J Stolfo, and Simha Sethumadhavan. 2012. Concurrency Attacks. *HotPar* 12 (2012), 15.

[59] Juan Zhai, Jianjun Huang, Shiqing Ma, Xiangyu Zhang, Lin Tan, Jianhua Zhao, and Feng Qin. 2016. Automatic Model Generation from Documentation for Java API Functions. In *Proceedings of the 38th International Conference on Software Engineering (ICSE 2016)*. ACM, 380–391.

[60] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. 2009. Inferring Resource Specifications from Natural Language API Documentation. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009)*. IEEE, 307–318.

Table 4: Details of the 18 proof-of-concept exploits.

CVE	Version	Vulnerability Type	Vulnerable Function	Critical Variables	System Calls
CVE-2015-0275	4.0.9	Denial of service	ext4_zero_range	inode.i_size	fallocate(FALLOC_FL_ZERO_RANGE)
CVE-2015-1333	4.1.3	Uncontrolled resource consumption	__key_link_end	edit	add_key()
CVE-2015-5706	4.0.3	Use after free	path_openat		open(O_TMPFILE)
CVE-2015-6937	4.2.3	Null pointer dereference	__rds_conn_create	trans	socket() bind()
CVE-2015-7872	4.2.6	Denial of service	key_gc_unused_keys	keyring.type_data.link	keyctl()
CVE-2015-7990	4.3.2	Race conditions	rds_sendmsg	trans	socket() bind()
CVE-2016-0728	4.4	Use after free	join_session_keyring		keyctl(KEYCTL_JOIN_SESSION_KEYRING)
CVE-2016-10147	4.8.14	Null pointer dereference	mcryptd_check_internal mcryptd_create_hash	algt	socket(AF_ALG)
CVE-2016-3134	4.5.2	Memory corruption	unconditional get_chainname_rulenum mark_source_chains check_underflow check_entry_size_and_hooks	ipt_entry.next_offset	socket(AF_INET) setsockopt(IPT_SO_SET_REPLACE)
CVE-2016-3841	4.3.2	Use after free	dccp_v6_send_response dccp_v6_request_recv_sock dccp_v6_connect inet6_destroy_sock inet6_sk_rebuild_header __ip6_datagram_connect inet6_csk_route_req inet6_csk_route_socket inet6_csk_xmit do_ipv6_setsockopt do_ipv6_getsockopt raw6_sendmsg cookie_v6_check tcp_v6_connect tcp_v6_send_synack tcp_v6_syn_recv_sock udpv6_sendmsg l2tp_ip6_sendmsg	np.opt	socket(AF_INET6) sendmsg()
CVE-2016-4482	4.6	Information leak / disclosure	proc_connectinfo	ci	ioctl(USBDEVFS_CONNECTINFO)
CVE-2016-4794	4.6	Use after free	pcpu_need_to_extend pcpu_alloc	chunk.map_extend_work pcpu_lock	mmap() bpf()
CVE-2016-6213	4.8.17	Uncontrolled resource consumption	commit_tree umount_tree attach_recursive_mnt alloc_mnt_ns copy_mnt_ns create_mnt_ns propagate_one		mount(MS_BIND)
CVE-2016-8646	4.3.5	Null pointer dereference	hash_accept		socket()
CVE-2016-9555	4.8.7	Buffer over-read	sctp_sf_ootb		socket(IPPROTO_SCTP)
CVE-2016-9793	4.8.13	Memory corruption	sock_setsockopt	sk_sndbuf sk_rcvbuf	socket() setsockopt(SO_SNDBUFFORCE)
CVE-2017-6074	4.9.11	Double free	dccp_rcv_state_process	ireq.pktopts skb	socket(AF_INET6) setsockopt(IPV6_RECVPKTINFO)
CVE-2017-6347	4.10	Buffer over-read	ip_cmsg_recv_checksum	skb.len	socket(AF_INET, SOCK_DGRAM, 0) sendto(MSG_MORE, INADDR_LOOPBACK)