IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received October 12, 2021, accepted November 9, 2021, date of publication November 17, 2021,
date of current version December 2, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3129062

Evaluating Code Coverage for Kernel Fuzzers via Function Call Graph

MINGI CHO, HOYONG JIN, DOHYEON AN, AND TAEKYOUNG KWON, (Member, IEEE)
Graduate School of Information, Yonsei University, Seoul 03722, South Korea

Corresponding author: Taekyoung Kwon (taekyoung@yonsei.ac.kr)

This work was supported by the Institute for Information and Communications Technology Planning and Evaluation (IITP) funded by the Government of Korea, Ministry of Science & Information Technology (MSIT), under Grant 2018-0-00513 (Machine Learning-Based Automation of Vulnerability Detection on Unix-Based Kernel).

ABSTRACT The OS kernel, which has full system privileges, is an attractive attack surface. A kernel fuzzer that targets system calls in fuzzing is a popular tool for discovering kernel bugs that can induce kernel privilege escalation attacks. To the best of our knowledge, the relevance of code coverage, which is obtained by fuzzing, to the system call has not been studied yet. For instance, modern coverage-guided kernel fuzzers, such as Syzkaller, estimate code coverage by comparing the entire set of executed basic blocks (or edges) regardless of the system call relevancy. Our insight is that the system call relevancy could be an essential performance indicator for realizing kernel fuzzing. In this regard, this study aims to assess the system call-related code coverage of kernel fuzzers. For this purpose, we have developed a practical assessment system that leverages the Intel PT and KCOV and assessed the Linux kernel fuzzers, such as Syzkaller, Trinity, and ext4 fuzzer. The experiments on different kernel versions demonstrated that approximately 32,000–47,000 functions are implemented in the Linux kernel, and approximately 9.7–15.2% are related to the system call. Our finding is that fuzzers that achieve higher code coverage in conventional metrics do not execute more basic blocks related to system calls. Thus, we recommend that kernel fuzzers use both system call-related functions and regular basic blocks in coverage metrics to assess fuzzing performance or to improve coverage feedback.

INDEX TERMS Fuzzing, kernel fuzzing, evaluation, system call, code coverage.

I. INTRODUCTION

Software bugs can cause incorrect and unintended states in a computer system, which can then be exploited as a path for intrusion by attackers. The OS kernel is an attractive attack surface because of its entire system privileges. Hence, it is important to discover and fix such bugs in the OS kernel ahead of attacks to prevent attackers from obtaining complete control over the system by exploiting its vulnerabilities. Over the past five years, more than 1,000 CVEs have been assigned to the Linux kernel, further indicating the presence of several latent vulnerabilities in OS kernels [1].

Various analysis methods (static, dynamic, and combined) have been used to discover latent bugs in the software. Fuzzing is a dynamic bug-finding technique that automatically generates or mutates inputs to trigger bugs, and it has successfully discovered critical bugs in real-world software applications [2]. Fuzzing is also often used for finding bugs in OS kernels by focusing on interfaces that handle user-mode and kernel-mode communications. Most kernel fuzzers target system calls that request a service from the kernel, including but not limited to process control, file management, and device management.

In this context, various kernel fuzzers have been proposed [3]–[6], and are widely deployed for finding kernel bugs. In particular, Syzkaller and its derivatives (and also AFL-derivatives) are coverage-guided fuzzers. To assess the performance of fuzzers, we generally consider the number of distinct bugs that are discovered during fuzzing. However, for a more rigorous assessment, we must also consider the number of code blocks that are explored during fuzzing [7]. Although the primary goal of a fuzzer is to find more bugs, a typical list of bugs discovered by a fuzzer (e.g., CVE ID) is not sufficient for understanding the performance of a fuzzer. Instead, code coverage can be used as an indicator to determine whether a fuzzer is adequately operated.

The associate editor coordinating the review of this manuscript and approving it for publication was Tawfik Al-Hadhrami.

Most kernel bugs discovered by the kernel fuzzer appear in the system call relevant functions. For example, Syzkaller applied KASAN to the Linux kernel and found 701 bugs (83.1%) out of the 844 fixed bugs found via system calls. Therefore, to assess the kernel fuzzer, we need to consider the system call relevancy. However, most kernel fuzzers estimate code coverage by comparing the entire set of executed basic blocks (or edges) regardless of the system call relevancy. This motivated us to assess the performance of kernel fuzzers by measuring the code coverage of system call-related functions and gaining insight for extending the existing fuzzers (e.g., to find an uncovered kernel component).

In this study,[1] we present a methodology to assess the code coverage performance of kernel fuzzers with system call-related functions, which can be executed through a system call. To assess kernel fuzzers, we first extracted a list of functions and basic blocks related to system calls (Section III-A). Thereafter, we built a guest OS environment on the virtual machine (e.g., KVM) and performed fuzzing using the kernel fuzzer that had to be assessed in the built guest OS environment. During the fuzzing process, we used the Intel PT [9] to record the execution trace of the change of flow instruction (CoFI) that affects the control flow, and we used the KCOV [10] to record the executed basic blocks (Section III-B). Finally, we assessed the performance of a kernel fuzzer by comparing the list of functions and basic blocks related to system calls and those that were executed by the fuzzer (Section III-C). We implemented and evaluated the proposed method using seven different Linux kernels and three different fuzzers: Syzkaller [4], Trinity [3], and ext4 fuzzer [11]. The evaluation results demonstrated that 32,000–47,000 functions were implemented in the Linux kernel and 9.7–15.2% of them were related to the system call. Moreover, we demonstrated that the results obtained while assessing fuzzers using system call-related basic blocks were different from those obtained using all the executed basic blocks.

The following are the contributions of this study:

- **Novel indicator for coverage metrics in kernel fuzzers.** We have proposed a new method for measuring the code coverage for kernel fuzzers regarding system call relevancy. When measuring code coverage, our method prioritizes the basic blocks executed by system calls rather than the entire set of basic blocks. Thus, it is easy to check which function the kernel fuzzer was stuck in and could not explore a new execution path. This method provides a new indicator for coverage metrics in kernel fuzzers, which can be used to assess the fuzzers in terms of code coverage performance, and further improve the fuzzers.
- **Engineering work to generate function call graphs of Linux kernel binaries.** When generating function call graphs of Linux kernel binaries, the problems caused by tail call optimization, retpoline, and dynamically registered functions (e.g., kernel driver) are challenging from an engineering perspective. We analyzed these problems in detail and resolved them in parts to implement our assessment system. Our results can be used for further studies.
- **Concrete evaluation of open-source kernel fuzzers.** We performed coverage evaluation on the Linux kernel fuzzers, such as Syzkaller, Trinity, and ext4 fuzzer, using the proposed approach. The evaluation results support the significance of system call relevancy in coverage metrics for kernel fuzzers.

[1] A preliminary version of this paper was presented as a poster [8] at ACM CCS 2019.

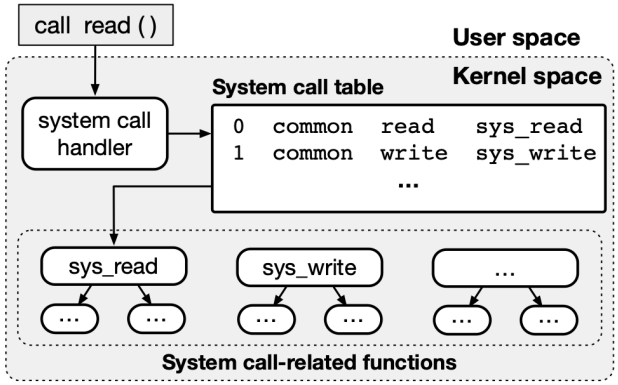


FIGURE 1. Steps to invoke a system call.

A. ORGANIZATION

Section II describes the technical background of this study. Section III presents the design and implementation of the proposed system. Section IV presents the experimental results. Section V discusses the limitations of our approach, and Section VI presents the related work. Finally, we present our conclusions in Section VII.

II. BACKGROUND

A. SYSTEM CALL

A system call is used by user-level programs to request privileged services to the OS kernel. For example, a user program uses a system call to create a process, read/write a file, or create a socket. The system call works as follows: At first, the user program executes an int 0×80 or a sysenter instruction to generate interrupts, after which context switching to the kernel space occurs. In the kernel space, a system call handler is executed to handle the exception; the system call handler then finds the system call function in the system call table according to the syscall number requested by the user program and executes it. We defined the system call function executed in the system call handler as the *system call entry function*. Furthermore, we defined this system call entry function and all the functions called by it as *system call-related functions*.

Figure 1 shows the processing steps of the kernel when *read()* is called from a user process. When *read()* is called,

TABLE 1. Number of total functions and system call-related functions in the Linux kernel extracted by previous study [8].

Version	Total	Related	Ratio
4.4.0-87	20,503	3,990	19.5%
4.9.0	20,963	3,797	18.1%
4.14.0	22,503	4,183	18.6%

the system call handler finds its entry function from the system call table and executes it. As the *read* system call has number 0, the user program sets the *rax* register to zero when requesting a *read()*. Finally, the entry function of *read*, the first function of the system call table, is executed.

B. KERNEL FUZZING

Fuzzing is a widely used method for finding software bugs. To find bugs in the OS kernels, the kernel fuzzing, which generates a random sequence of system calls with random arguments, is used. Trinity [3] is a widely used system call fuzzer that targets the Linux system. It creates well-defined structured inputs using a predefined system call template. Because the number of arguments and the type of argument used for each system call vary, fuzzing can be effectively performed using a predefined template. Recently, Syzkaller [4] has emerged as the most widely used kernel fuzzer targeting the Linux kernel. Syzkaller is a coverage-guided fuzzer that not only leverages template-based fuzzing, as in the case of Trinity, but also measures the code coverage of the kernel and feeds it to the fuzzer to discover a new code path. Coverage-guided fuzzing has already exhibited its effectiveness in finding bugs in user-level programs [12], [13], and recently, Syzkaller has been successful in finding bugs in the Linux kernel. Among the Linux kernel bugs reported by Syzkaller, more than 3,000 bugs have been patched [14], and out of the 844 bugs included in the KASAN report, 83.1% were discovered through system calls. Many recent kernel fuzzers have been researched based on Syzkaller [15].

Table 1 lists the number of system call-related functions extracted from our previous study [8]. Note that the previous study is limited in that only the Radare command *aaa* was used to analyze the kernel binary. In contrast, in this study, we additionally used the *af* command to analyze functions that Radare could not automatically analyze. The system call-related functions account for less than 20% of the functions implemented in the entire Linux kernel. Although system call-related functions account for a relatively small percentage of the total functions, more than 80% of the bugs found in the Linux kernel have been discovered in these functions. Therefore, we extracted system call-related functions and used them to assess kernel fuzzers.

C. CODE COVERAGE MEASUREMENT

Coverage-guided fuzzing, which generates a new input by feeding back code coverage information, is used to test the code in the deep path. Because AFL and LibFuzzer successfully found a large number of bugs in user-space programs, coverage-guided fuzzers are also widely used in kernel fuzzing. To implement a coverage-guided fuzzer, it is necessary to collect the coverage information. The primary requirement of the code coverage measurement method used for fuzzing (especially in kernel fuzzing) is a low performance overhead. As fuzzing is a time-consuming task, if a high overhead occurs while measuring code coverage, fewer code blocks will be executed within a limited time, and as a result, fewer bugs can be found. The following are the details of Intel processor trace (PT) and KCOV used for the coverage measurement in the Linux kernel.

1) INTEL PROCESSOR TRACE

Intel PT records software execution traces in a packet format using the processor's hardware device [9]. It records the packets when instructions that change the control flow of an execution, called change of flow instructions (CoFIs), are executed. The types of packets are taken not-taken (TNT) packets, target IP (TIP) packets, and flow update packets (FUP). The TNT packet records the result (true or false) of the branch condition. The TIP packet records the address of the next instruction to be executed when an indirect jump instruction that uses the relative address is executed. The FUP records information about asynchronous events such as interrupts and traps. Intel PT has been utilized for binary-only fuzzing in both user and kernel space [11], [16]–[18].

2) KCOV

KCOV is a tool that records code coverage information of the Linux kernel tailored for coverage-guided fuzzing, which has been supported since Linux kernel 4.6 [10]. KCOV aims to measure basic block level code coverage for the functions of system call inputs rather than measuring the whole functions executed in the kernel. To achieve this, code coverage of soft/hard interrupts and some non-deterministic parts of the kernel is not measured by default; only the code coverage of user threads that handle system calls is recorded. As a result, the background processes and irrelevant code are ignored to enable a more efficient fuzzing performance. To measure code coverage, KCOV inserts the function *__sanitizer_cov_trace_pc()*, which tracks the process counter of the executed basic block in every basic block. The process counters are then stored in a coverage buffer.

III. SYSTEM DESIGN AND IMPLEMENTATION

In this section, we explain the construction of our assessment system (Figure 2). At first, we used the system call list, `System.map`, and kernel images to retrieve the memory addresses of the functions related to the system call. Thereafter, we performed kernel fuzzing with a target fuzzer on the guest OS environment, and the addresses of the executed functions were recorded by Intel PT and KCOV. Finally, we compared the system call-related functions with the executed functions to assess the code coverage performance of the kernel fuzzer. In this paper, code coverage performance

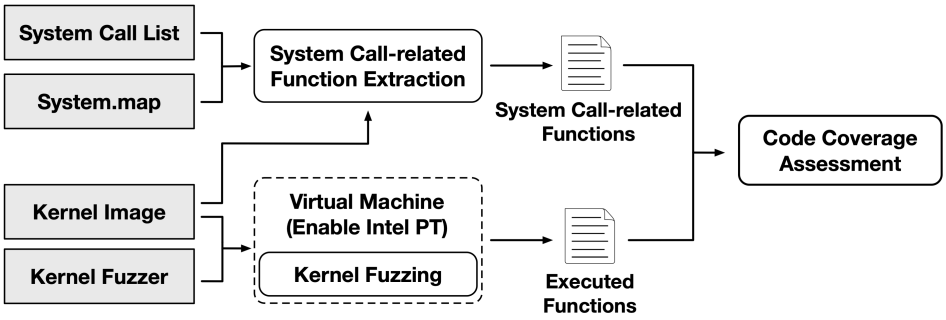


FIGURE 2. System architecture for fuzzer assessment.

assessment compares the number of functions and basic blocks executed on fuzzing.

A. EXTRACT SYSTEM CALL-RELATED FUNCTIONS

To assess the code coverage performance of the kernel fuzzers, we first need to extract the system call-related functions among the functions implemented in the kernel and determine the address loaded onto the memory. System call-related functions refer to the kernel functions that construct a function call graph (FCG) of the entry function that can be executed when the system call is called. We extracted only system call-related functions from all functions and focused on their coverage because not all functions in the kernel could be executed by requesting system calls. The kernel fuzzer aims to discover bugs triggered through system calls that can be exploited by attackers with user privileges. Therefore, it is reasonable to focus on system call-related functions rather than on the entire set of functions implemented in the kernel to assess the kernel fuzzers. There could exist some kernel bugs which are caused by worker threads that use shared resources that could be controlled through system calls. However, it is difficult to identify and analyze functions that can be indirectly executed through system calls. Therefore, we first assessed the kernel fuzzer using system call-related functions and we shall consider the identification of functions that could trigger bugs through worker threads in the future work.

System call-related functions are extracted in four steps: 1) The list of system call entry functions from the system call table is obtained. 2) The address of each entry function is extracted by referring to the System.map. 3) The FCG of the functions implemented in the kernel binary are generated. 4) A list of all functions generated by the system call entry function is obtained. Because code coverage in the basic block level is used as a de facto standard for kernel fuzzing, we also extracted the basic block address of the system call-related functions. The details of each step are as follows.

1) EXTRACTION OF SYSTEM CALL ENTRY LIST

We used the *arch/x86/entry/syscalls/syscall_64.tbl* to obtain the names of the system call entry functions, which manages system calls from the Linux source code. The *syscall_64.tbl* lists the system call number, ABI, system call name, and entry function. We extracted the names of the entry functions from *syscall_64.tbl* for use in the subsequent step. From Linux kernel v4.17, *ksys_xxx*, which functions similarly with respect to the system call, was added. The existing system call entry functions (*sys_xxx*) were changed to simply call *ksys_xxx*. The problem caused by this change is that the FCG was not completely generated because of the application of compiler optimization in entry functions. Therefore, from kernel version v4.17, we added the functions named *ksys_xxx* to the system call entry list. From Linux kernel v5.7, the entry function name in the tbl file omits the __x64_ prefix, and hence to extract the entry functions properly, we appended __x64_ prefix to the entry function.

2) EXTRACTION OF SYSTEM CALL ENTRY FUNCTION ADDRESS

We used the *System.map*, which contains kernel image symbols and corresponding addresses, to find the kernel memory address of the entry function of the system call from the loaded kernel image. Because *System.map* also contains functions that cannot be executed via a system call (i.e., functions not related to the system call), we obtained the address of entry functions obtained in 1) and collected other system call-related functions through the following steps. *System.map* is located in the root of the source directory after building the Linux kernel; for some Linux distributions, it is located in the */boot* directory.

3) FUNCTION CALL GRAPH GENERATION

To obtain all of the system call-related functions, we created an FCG for each entry function obtained in 2). We generated the FCG from the *vmlinux*, a Linux kernel image, which can be obtained by building a Linux kernel from the source code or by decompressing the *vmlinuz*. In general, a FCG generated from source code is more accurate than a FCG generated from binary. However, numerous macros are implemented in the Linux kernel. Different execution paths are created according to build configurations; thereby, many engineering efforts are required to extract a precise FCG from the source code. We decided to generate FCG from binary, considering the application to the COTS OSes

and device drivers in the future. To decompress vmlinuz, we used *extract-vmlinux* provided by the Linux project [19]. We used *Radare2* [20], a reverse-engineering framework, to generate an FCG.[2] We first ran the *aaa* command to analyze the binary kernel, and thereafter ran the *af* command to analyze functions listed in the System.map. Later, we ran the *agCj* command to generate the global FCG in the JSON format. The JSON format of the FCG is as follows:

```
[“name”:“func1”,“size”:100,“imports”:
                    [“sub1”,“sub2”], . . .]
```

The *name*, *size*, and *imports* represent the function name, size of the function implemented in a binary, and symbols used in the function, respectively. Because not all symbols in the *imports* are functions, only symbols in a range of text areas of the kernel are considered as functions. The text area ranges from *_stext* to *_etext*, and these addresses are listed in the System.map.

4) EXTRACTION OF SYSTEM CALL-RELATED FUNCTION ADDRESSES

Finally, we obtained the address of all system call-related functions in the FCG that was generated from the *System.map*. For coverage assessment at the basic block level, we extracted the addresses of the basic blocks for each function. The *afb* command of Radare2 extracts the start and end, and the size of the basic blocks.

From the abovementioned process, the address list of system call-related functions and a list of their basic blocks were generated.

B. CODE COVERAGE MEASUREMENT

The code coverage of the Linux kernel can be measured using both static and dynamic instrumentation. To measure code coverage statically, the code for coverage measurement was inserted into the original code during the compilation process. However, the static method requires the source code for instrumentation, and thus, it is difficult to apply it to closed-source OS kernels such as Windows. For example, *KCOV*, which is a coverage sanitizer in the kernel, only supports open-source OS kernels such as Linux. In this section, we have demonstrated the dynamic instrumentation method using Intel PT, which can be applied regardless of the environment of the kernel fuzzer to be assessed, and the static instrumentation method using KCOV for Syzkaller-based fuzzers. After Intel PT and KCOV recorded the address of the executed basic blocks, we extracted the address of the executed functions from the recorded basic block addresses.

1) INTEL PT

Dynamic instrumentation uses a virtual machine to handle the instructions executed on the guest OS. However, this method is software-intensive; therefore, it incurs a significant performance overhead. To address this issue, we used the Intel PT (supported by CPU-level features) and a modified version of kAFL [11], a coverage-guided kernel fuzzing framework that uses Intel PT.

kAFL consists of a VM infrastructure, a user-mode agent, and fuzzing logic. The VM infrastructure tracks the control flow of the guest OS by leveraging VT-x for hardware-assisted virtualization and Intel PT for tracking execution records. The user agent manages the entire process, including the initiation and termination of the fuzzer. We modified the user agent to trace the coverage of the target kernel fuzzer. To handle asynchronous events such as interrupts, the kAFL filters out the TIP that is marked with FUP from the Intel PT trace.

To measure the code coverage of the kernel fuzzer, we ran a target kernel fuzzer on a guest OS. A large number of system calls and their related functions were executed during fuzzing, and the Intel PT logged the address of the executed kernel basic blocks.

2) KCOV

The KCOV measures code coverage by inserting an instrument code for coverage measurement into the source code while compiling. Syzkaller is the most representative coverage-guided fuzzer using KCOV, and many recent kernel fuzzers, such as HFL [21], Agamotto [22], Moonshine [23], and Charm [24], were implemented along with Syzkaller. Because these fuzzers already measure code coverage using KCOV, the use of Intel PT is not always required. Because Syzkaller manages a coverage buffer that contains de-duplicated addresses of the executed basic blocks, the raw addresses can be obtained by running the following command:

wget http://localhost:<syz-manager port>/rawcover

The *syz-manage port* is an HTTP port number, as described in the Syzkaller configuration. To assess the fuzzer, we extracted coverage information at regular intervals (e.g., 1 min).

C. ASSESSMENT OF KERNEL FUZZERS

To assess the performance of kernel fuzzers, we compared the list of system call-related functions generated in Section III-A with the list of executed functions generated in Section III-B. Because functions that can be executed through the system call are listed in the system call-related function list, it is possible to assess the performance of the kernel fuzzer by analyzing the number of functions executed during fuzzing. We evaluated the validity of this strategy in the section that follows.

IV. EVALUATION

To evaluate our assessment system, we analyzed the applicability of system call-related functions and basic blocks as fuzzer assessment indicators. Thereafter, we directly assessed several kernel fuzzers using our system. For this purpose, we posed the following questions:

[2]Radare2 v5.3.1 is used in our system.

- **RQ1:** Can the proposed system extract a larger number of system call-related functions and basic blocks compared to the previous study? (Section IV-B)
- **RQ2:** Can system call-related functions assist in the improvement of the fuzzer? (Section IV-C)
- **RQ3:** Can system call-related functions and basic blocks be used as metrics for assessing kernel fuzzers? (Section IV-D)

A. SETUP

To answer these research questions, we evaluated several fuzzers with different Linux kernels.

1) KERNELS

We used seven different versions of Linux kernels, four of which were compiled from the source code, and the remaining were downloaded Ubuntu distributions. The four compiled versions, 4.14, 4.19, 5.4, and 5.10, are the LTS version of the Linux kernel, and the downloaded kernels, 4.4.0-87-generic, 4.15.0-88-generic, and 5.4.0-72-generic are used in the LTS version of the Ubuntu. We compiled the kernels with GCC 6.5, and used the default kernel configuration for Syzkaller.

2) FUZZERS

We used three different fuzzers: Syzkaller, Trinity, and ext4 fuzzer. Syzkaller and Trinity were compiled from git commit *fdb2bb* and *03f10b*, respectively. We used the ext4 fuzzer included in kAFL v0.1. Syzkaller fuzzes the kernels that we compiled, and other fuzzers fuzz the downloaded kernels.

3) EXPERIMENT ENVIRONMENT

We tested our system on 384GB RAM, and Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz that supports VT-x and Intel PT. Ubuntu 16.04, with Linux kernel 4.6.2, ran on the machine as the host OS.

B. EXTRACTION OF SYSTEM CALL-RELATED FUNCTIONS AND BASIC BLOCKS (RQ1)

To answer RQ1, we extracted system call-related functions from seven different versions of Linux kernel binaries and compared the number of extracted functions with those mentioned in a previous study [8]. This indicates that our system can successfully extract system call-related functions and basic blocks from various Linux kernel binaries and that the related functions occupy a small proportion in the Linux kernel binary. In addition, it indicates whether the process improved in Section III-A helps in the extraction of the system call-related functions that the previous work could not extract. Because most kernel bugs are discovered in system call-related functions, it is possible to perform more efficient fuzzing by assigning a high priority to the related functions.

Table 2 lists the results of the extracted system call-related functions according to the Linux kernel version. The first column indicates the Linux kernel version. The second and third columns list the number of total functions extracted from the Linux kernel image and the number of system call-related functions, respectively. As listed in the Table 2, 32,000–47,000 functions were implemented in the Linux kernel, and 9.7–15.2% of them were system call-related functions. Table 3 lists the number of system call-related basic blocks. We can see that 11–18% of the total basic blocks are related to the system call.

TABLE 2. Number of total functions and system call-related functions in the Linux kernel.

Version	Total	Related	Ratio
4.14	39,159	4,818	12.3%
4.19	41,892	4,083	9.7%
5.4	43,769	4,842	11.1%
5.10	46,807	5,147	11.0%
4.4.0-87-generic	32,323	4,756	14.7%
4.15.0-88-generic	37,658	5,605	14.9%
5.4.0-72-generic	42,397	6,454	15.2%

TABLE 3. Number of total basic blocks and system call-related basic blocks in the Linux kernel.

Version	Total	Related	Ratio
4.14	424,618	64,786	15.3%
4.19	458,048	51,726	11.3%
5.4	488,946	66,565	13.6%
5.10	519,620	72,193	13.9%
4.4.0-87-generic	390,117	61,024	15.6%
4.15.0-88-generic	468,249	76,417	16.3%
5.4.0-72-generic	531,850	95,461	18.0%

From Table 2, we can see that Linux kernel 4.19 has a lower ratio of system call-related functions compared to other versions. According to our analysis, this is due to the mitigation technique added to the Linux kernel since v4.15. *Retpoline* [25] was applied in the Linux kernel to prevent side-channel attacks using the Specter V2 [26] vulnerability. It prevents Specter V2, which can occur in the indirect branch predictor, by not using an indirect jump/call. An example of an indirect jump is a switch statement. When the switch statement is compiled, an indirect jump is executed by referring to the jump table generated by the compiler. The size of the jump table can be obtained by referring to the operand of the *cmp* instruction of the previous basic block of the basic block where the indirect jump instruction exists.

However, if *retpoline* is applied, an *__x86_indirect_thunk_rax* symbol can be added as a trampoline to avoid an indirect jump. As a result, Radare2 cannot accurately analyze the switch statement in this case and therefore restores only the control flow corresponding to one of the several cases. In Linux kernels 5.4 and 5.10, this problem did not occur because the jump table was not used for the switch statement.

Figure 3 shows the number of system call-related functions extracted by previous [8] and current studies. As shown in the figure, this study extracted more related functions than those in a previous study in all target Linux

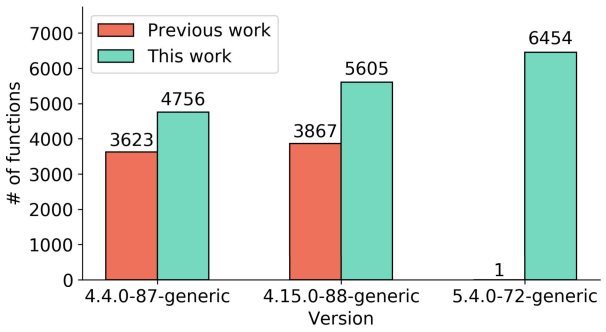


FIGURE 3. Comparison of the number of system call-related functions extracted by previous study [8] and this study.

kernel versions. In 5.4.0-72-generic, the previous study failed to extract related functions because of compiler optimization caused by the use of *ksys_xxx*. Our system treats *ksys_xxx* as system call entry functions, so the system call-related functions can be extracted even in the latest version of the Linux kernel. In addition, because we analyzed the functions manually using the Radare2 command by referring to the System.map for functions that were not automatically analyzed by Radare2, we could extract more than 30% of system call-related functions compared to the previous study. Because the previous study was tested on different environments (different compilers and Radare2 versions), the number of extracted related functions depicted in Figure 1 is slightly different from those listed in Table 1.

From Tables 2 and 3, we can see that, our system successfully extract the system call-related functions and basic blocks from the various versions of the Linux kernel. From Figure 3, we can see that our system extracts more than 30% of related functions compared to the previous study. These observations allowed us to positively answer RQ1.

C. ANALYZING KERNEL FUZZERS (RQ2)

To answer RQ2, we performed 24h fuzzing using Syzkaller, Trinity, and ext4 fuzzer. Thereafter, we compared the number of total related functions and basic blocks with the number of executions among them. Because system call-related functions can be executed directly by the kernel fuzzer, all related functions and basic blocks should be tested by the fuzzer. Therefore, if we can identify a system call-related function that has not been tested by the fuzzer, the reason for not executing the function would be determined to improve the fuzzer.

Figure 4 shows the number of executed system call-related functions and basic blocks as a result of the 24h fuzzing by Syzkaller. The ratios of executed related functions were 53.5%, 57.8%, 58.5%, and 59.3% for kernels 4.14, 4.19, 5.4, and 5.10, respectively. The ratios of the executed related basic blocks were 31.3%, 35.0%, 35.4%, and 35.1%, respectively. We investigated why more than 40% of the system call-related functions were not tested by Syzkaller. According

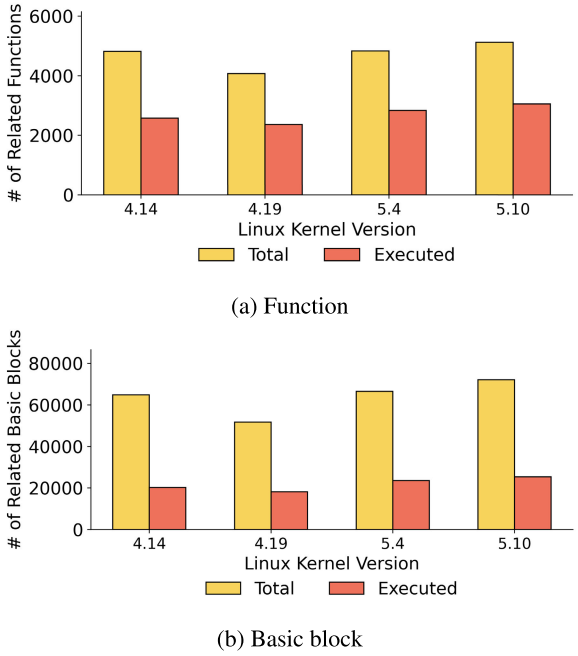


FIGURE 4. Number of system call-related functions and basic blocks executed by 24h fuzzing of Syzkaller.

TABLE 4. Number of unexecuted functions by 24h fuzzing of Syzkaller.

Version	# Unexecuted functions		Ratio
	Total	Related	
4.14	31,987	2,242	7.01%
4.19	33,210	1,723	5.19%
5.4	34,893	2,008	5.75%
5.10	37,520	2,094	5.58%

to our analysis, these functions failed to execute for the following two reasons. The first is that the fuzzer overlooked a large portion of the related functions and basic blocks while fuzzing. In this case, the reason for Syzkaller failing to execute the functions could be manually analyzed and used to improve the related fuzzers, including Syzkaller. The second reason is that many of the executed functions, classified as a system call-related function by our system, were not recorded by KCOV for coverage measurement. In this case, the related functions that are actually executed were classified as unexecuted functions in the experiment. Because KCOV is designed for fuzzing, it does not measure the coverage of the entire set of functions, but only the coverage of the functions related to the syscall input.

Table 4 lists the number of unexecuted functions as a result of 24h fuzzing by Syzkaller. To improve the fuzzers, it is necessary to identify the unexecuted functions and induce them to execute these functions. For example, if a target program needs to satisfy a complex conditional statement to execute a specific function, we can update the mutation strategy of the fuzzer to satisfy the statement. However, because there are a large number of functions implemented in the kernel,

TABLE 5. Number of system call-related functions and basic blocks by 24h fuzzing of Trinity and ext4 fuzzer.

Fuzzer	#Related functions		#Related basic blocks	
	Total	Executed	Total	Executed
Trinity	4,756	1,737 (36.5%)	61,024	11,665 (19.1%)
Ext4 fuzzer		1,533 (32.2%)		10,337 (16.9%)

it is difficult to know which functions the fuzzer can test but not tested. In Table 4, we can see that Syzkaller failed to execute more than 30,000 functions, which is about 60% of the total functions implemented in the Linux kernel listed in Table 2. Analyzing all 30,000 functions to improve Syzkaller takes a very long time, so it is more efficient to improve the fuzzer by first analyzing unexecuted system call-related functions, which are considerably smaller in number. There are approximately 2,000 related functions that have not been executed, which occupy 5–7% of the total functions.

Table 5 lists the results of the 24h fuzzing by Trinity and ext4 fuzzer. The Trinity and ext4 fuzzer executed 36.5% and 32.2% of the related functions and 19.1% and 16.9% of the related basic blocks, respectively. Because the Trinity and ext4 fuzzers are relatively simple fuzzers compared to Syzkaller, they executed a lower percentage of related functions and basic blocks compared to Syzkaller, although they fuzzed different kernel binaries. Because we can easily identify functions that are not executed among system call-related functions, it can be used to improve the fuzzer efficiently by analyzing these functions first.

D. ASSESSING KERNEL FUZZERS (RQ3)

To answer RQ3, we compared the total number of functions and basic blocks executed as a result of the 24h fuzzing with the number of related functions. Figure 5 shows the result of code coverage as a result of the 24h fuzzing by Trinity and ext4 fuzzer. As depicted in Figure 5a, Trinity achieved a higher function coverage compared to the ext4 fuzzer for all the functions and related functions. Trinity achieved most of the function coverage at the start of fuzzing because it called the entire set of system calls randomly. However, because it called the system call without information about the dependency across them, the coverage did not increase during fuzzing. In contrast, the ext4 fuzzer found a new code block during fuzzing because the kAFL gives coverage feedback on coverage to the fuzzer.

In Figure 5b, the basic block coverage results for both fuzzers are slightly different from those depicted in Figure 5a. As a result of the 24h fuzzing, the basic block coverage of the ext4 fuzzer was higher than that of Trinity; however, the Trinity executed more system call-related basic blocks. It can be considered that ext4 fuzzer achieved higher code coverage than Trinity when it compared only the total number of basic blocks; however, while comparing system call-related basic blocks, Trinity achieved higher code coverage. Therefore, rather than comparing only the total number of executed

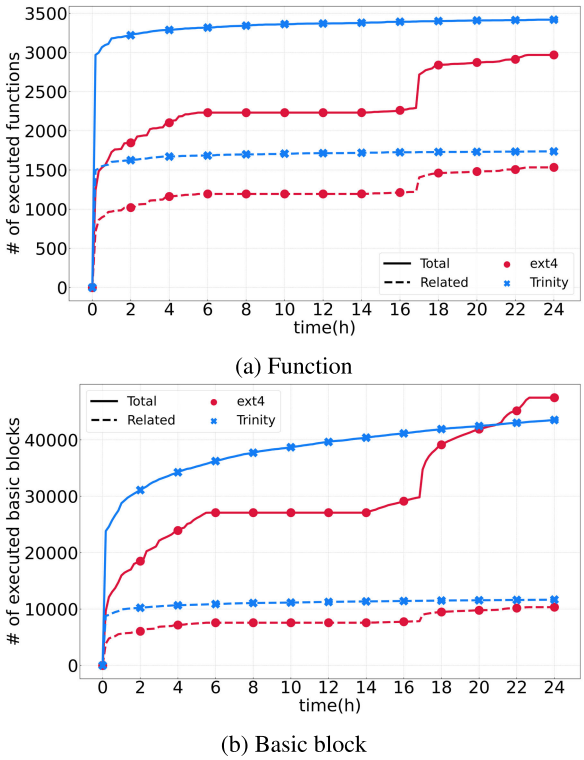


FIGURE 5. Number of executed functions and basic blocks by 24h fuzzing of Trinity and ext4 fuzzer.

functions and basic blocks, we can assess different aspects of fuzzers using system call-related functions and basic blocks. In the future study, we plan to experiment with whether our metric is helpful in terms of bug finding.

V. DISCUSSION

We have proposed a method to assess a kernel fuzzer using system call-related functions in this study. We address the limitations of this study and discuss further direction in this section.

A. KERNEL DRIVERS

System call-related functions were extracted using the FCG generated from the kernel binary. However, not all related functions implemented in the Linux kernel appear in the FCG. Some components register functions dynamically during kernel execution; therefore, these functions cannot be extracted through statically generated FCG. The kernel driver is one such component. Because the Linux kernel supports various devices, it is difficult to control them using general system calls. Therefore, each device uses a driver that includes an operation function that controls the device itself.

Among the operations used in the driver, *ioctl* supports device-specific input/output and other operations required by a device. Because *ioctl* enables direct access to the device driver from the user-level process, it is considered to be an important attack vector for attackers.

```
static const struct tty_operations con_ops = {
        .install = con_install,
        .open = con_open,
        .close = con_close,
        .write = con_write,
        .write_room = con_write_room,
        .put_char = con_put_char,
        .flush_chars = con_flush_chars,
        .chars_in_buffer = con_chars_in_buffer,
        .ioctl = vt_ioctl,
#ifdef CONFIG_COMPAT
        .compat_ioctl = vt_compat_ioctl,
#endif
        .stop = con_stop,
        .start = con_start,
        .throttle = con_throttle,
        .unthrottle = con_unthrottle,
        .resize = vt_resize,
        .shutdown = con_shutdown
};
```

FIGURE 6. The struct tty_operations in Linux kernel v4.15.

Therefore, while assessing kernel fuzzer, the code coverage of functions implemented in a device driver should be measured. However, because the current FCG does not cover dynamically registered driver functions, assessment of the driver may not be accurate.

Figure 6 shows the *tty_operations* of the *tty* device. As depicted in the figure, the driver has 16 operations (when CONFIG_COMPAT is defined). When *open* is requested on the device, *con_open* is executed, and when *write* is requested, *con_write* is executed. These driver operations are registered when the driver is initialized. To extract these functions as system call-related functions, it is necessary to identify the operation functions registered during the initialization of the device driver and thereafter add them as entry functions.

B. ACCURACY OF BINARY ANALYSIS

As we extract system call-related functions through the FCG, the accuracy of related functions depends on the accuracy of the generated FCG. Unfortunately, FCG extracted by binary analysis was not soundly generated for reasons such as compiler optimization and vulnerability mitigation. Figure 7 shows the disassembly code of the *__x64_sys_read* function of the Linux v4.19 kernel. This is the entry function of the *read* system call, and it executes *ksys_read*. As shown in the last line of the disassembly code, *ksys_read* is executed through the *jmp* instruction. For calling a function, *call* instruction is generally used, but *jmp* instruction is used instead of *call* because of the tail call optimization [27]. As a result, Radare2 failed to analyze the relationship between *__x64_sys_read* and *ksys_read*. The problem that occurs in the system call entry function can be solved by adding the *ksys_xxx* functions to the entry function, but other incorrectly extracted system call-related functions could be attributed to this.

```
__x64_sys_read:
dbg.__x64_sys_read (pt_regs const *regs);
; arg pt_regs const *regs @ rbx
0xffffffff81304f00      e82bc88f00      call sym.__fentry__
0xffffffff81304f05      53              push rbx
0xffffffff81304f06      4889fb          mov rbx, rdi
0xffffffff81304f09      e8c294e8ff      call dbg.__sanitizer_cov_trace_pc
0xffffffff81304f0e      488b5360        mov rdx, qword [rbx + 0x60]
0xffffffff81304f12      488b7368        mov rsi, qword [rbx + 0x68]
0xffffffff81304f16      488b7b70        mov rdi, qword [rbx + 0x70]
0xffffffff81304f1a      5b              pop rbx
0xffffffff81304f1b      e900ffffff      jmp dbg.ksys_read
```

FIGURE 7. Disassembly of __x64_sys_read in Linux kernel v4.19.Due to tail call optimization, ksys_read is called through *jmp* instruction rather than *call-ret* pair.

C. SELECTIVE ANALYSIS

We extracted system call-related functions that can be used in further kernel analyses. Because the kernel has a large number of implemented functions, kernel analysis methods incur a high performance overhead. The selective analysis of kernels using related functions makes it possible to perform efficient analysis.

Sanitizers such as KernelAddressSanitizer [28] (KASAN), UndefinedBehaviorSanitizer [29] (UBSan), and KernelMemorySanitizer [30] (KMSAN) were applied to the kernel to find vulnerabilities. While fuzzing with a sanitizer, we can efficiently fuzz by selectively sanitizing system call-related functions. Additionally, symbolic/concolic execution [31], [32] or dynamic taint analysis [33], [34] are also used for kernel analysis. However, such methods make kernel analysis impractical because the performance overhead ranges from tens to thousands of times as a result. Therefore, it would be more efficient to analyze the system call-related functions prior to analysis of other functions. We expect a fruitful result employing this direction in further research.

VI. RELATED WORK

Many kernel-fuzzing studies have been proposed to detect kernel vulnerabilities. Most kernel fuzzing techniques target system calls and are divided into cases with and without coverage information.

A. KERNEL FUZZING

Kernel fuzzers without coverage information include the Trinity [3] and IMF [6]. Trinity performs fuzzing by calling system calls randomly to the Linux targets. It stores information about the arguments and its type as a template so that the system call can be correctly executed. The IMF performs fuzzing on macOS by leveraging a model for learning the inferred dependencies among API function calls.

B. COVERAGE-GUIDED KERNEL FUZZING

Kernel fuzzers with coverage information include kAFL [11], Syzkaller [4], and TriforceAFL [35]. kAFL measures the kernel code coverage using virtualization and Intel PT to

provide coverage information to the fuzzer in a feedback loop. However, kAFL does not automatically generate valid system calls. The Syzkaller runs a target kernel on the QEMU virtual machine and requires the target kernel compiled with KCOV to measure coverage. TriforceAFL leverages QEMU to provide coverage information of the entire system, including the OS kernel. TriforceLinuxSyscallFuzzer [5] is a Linux system call fuzzer based on the TriforceAFL.

Recently, advanced fuzzing techniques based on Syzkaller have been studied [15]. DIFUZE [36] analyzed the ioctl interfaces using static analysis to discover vulnerabilities in the kernel driver. MoonShine [23] used a static analysis to detect dependencies across different system calls. RAZZER [37] leveraged static analysis and deterministic thread interleaving to detect race conditions in the Linux kernel. HFL [21] is a hybrid kernel fuzzer that combines fuzzing with symbolic execution. These fuzzers evaluate code coverage using the basic block or edge coverage measured by the KCOV. System call-related basic blocks can be used as another metric to assess the code coverage of these fuzzers. This will help to identify the missed basic blocks that can be executed by the fuzzer and thus, improve the fuzzer.

C. EVALUATING FUZZERS

Since the early 1990s, many fuzzing techniques have been studied to find software bugs [38]. Each fuzzing technique targets different types of programs, and each fuzzer has different strengths; consequently, it is very difficult to evaluate their performance. Recent studies have evaluated fuzzers based on the guidelines presented by Klees *et al.* [7].

The ground-truth metrics used to evaluate fuzzing techniques are discovered bugs and code coverage. Because the primary purpose of fuzzing is to find bugs in the target program, the fuzzer that detects a greater number of bugs is considered superior. However, the comparison of the number of bugs found alone is insufficient to evaluate the effectiveness of the fuzzing algorithm. Because the number of bugs used for the experiment is usually insufficient, a more efficient fuzzer may discover fewer bugs depending on the target program. To address this limitation, the automatic bug-injection frameworks LAVA and EvilCoder [39], [40] inserted a large number of artificial bugs into a real program. However, because these bugs have low diversity, they are still limited as a dataset for evaluating fuzzing. Magma [41] created high-quality ground-truth fuzzing benchmarks by considering diversity, verifiability, and usability, but these are still a small number of benchmarks for evaluating fuzzers.

There are still limitations in the evaluating of fuzzers by comparing the number of bugs found. Code coverage is used as a secondary metric to evaluate fuzzing techniques [7]. Because the kernel is larger than the user program, and various tasks are performed in the background, evaluating the kernel may not be an accurate approach when using the entire code coverage. To alleviate this problem, we extracted system call-related functions and basic blocks targeted by kernel fuzzers and used them for the code coverage assessment of kernel fuzzers.

VII. CONCLUSION

We have proposed a method to assess the code coverage of kernel fuzzers using system call-related functions and basic blocks. The evaluation results show that 32,000–47,000 functions were implemented in the Linux kernel, out of which 9.7–15.2% are related to the system call. Furthermore, as a result of the 24h fuzzing by Syzkaller, Trinity, and ext4 fuzzer, approximately 53.5% (v4.14), 36.5%, and 32.2% of system call-related functions were executed. Moreover, our evaluation results show that while assessing fuzzers with system call-related basic blocks, the evaluation result can be different from those obtained when assessing fuzzers with all the executed basic blocks. In future studies, we intend to analyze functions that are not covered by existing kernel fuzzers and extend the target for fuzzers that support other OS kernels such as Windows and macOS.

REFERENCES

[1] CVE Details. (2021). *Linux Kernel Vulnerability Statistics*. Accessed: Aug. 9, 2021. [Online]. Available: https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33

[2] K. Serebryany. (2017). *OSS-Fuzz-Google's Continuous Fuzzing Service for Open Source Software*. Accessed: Aug. 9, 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/serebryany

[3] (2019). *Trinity: Linux System Call Fuzzer*. Accessed: Aug. 9, 2021. [Online]. Available: https://github.com/kernelslacker/trinity

[4] (2019). *Syzkaller: Linux Syscall Fuzzer*. Accessed: Aug. 9, 2021. [Online]. Available: https://github.com/google/syzkaller

[5] J. Hertz and T. Newsham. (2019). *TriforceLinuxSyscallFuzzer*. Accessed: Aug. 9, 2021. [Online]. Available: https://github.com/nccgroup/TriforceLinuxSyscallFuzzer

[6] H. Han and S. K. Cha, "IMF: Inferred model-based fuzzer," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 2345–2358.

[7] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, Oct. 2018, pp. 2123–2138.

[8] S. Kim, S. Jeong, M. Cho, S. Chung, and T. Kwon, "Poster: Evaluating code coverage for system call fuzzers," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2019, pp. 2689–2691.

[9] PTD Library. (2013). *Intel Processor Tracing*. Accessed: Aug. 9, 2021. [Online]. Available: https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing

[10] Development Tools for the Kernel. (2021). *KCOV: Code Coverage for Fuzzing*. Accessed: Aug. 9, 2021. [Online]. Available: https://www.kernel.org/doc/html/latest/dev-tools/kcov.html

[11] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-assisted feedback fuzzing for OS kernels," in *Proc. USENIX Secur. Symp. (SEC)*, Aug. 2017, pp. 167–182.

[12] M. Zalewski. (2014). *American Fuzzy Lop*. Accessed: Aug. 9, 2021. [Online]. Available: http://lcamtuf.coredump.cx/afl/

[13] LLVM. (2021). *Libfuzzer—A Library for Coverage-Guided Fuzz Testing*. Accessed: Aug. 9, 2021. [Online]. Available: https://llvm.org/docs/LibFuzzer.html

[14] (2021). *Syzbot*. Accessed: Aug. 9, 2021. [Online]. Available: https://syzkaller.appspot.com/upstream

[15] Syzkaller. (2021). *Research Work Based on Syzkaller*. Accessed: Aug. 9, 2021. [Online]. Available: https://github.com/google/syzkaller/blob/master/docs/research.md

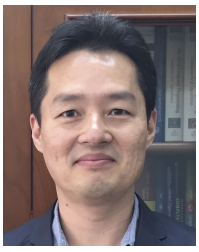
[16] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: Fuzzing with input-to-state correspondence," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, vol. 19, 2019, pp. 1–15.

[17] G. Zhang, X. Zhou, Y. Luo, X. Wu, and E. Min, "PTfuzz: Guided fuzzing with processor trace feedback," *IEEE Access*, vol. 6, pp. 37302–37313, 2018.
[18] Y. Wang, Z. Wu, Q. Wei, and Q. Wang, "NeuFuzz: Efficient fuzzing with deep neural network," *IEEE Access*, vol. 7, pp. 36340–36352, 2019.
[19] Linux. (2021). *Extract-Vmlinux*. Accessed: Aug. 9, 2021. [Online]. Available: https://github.com/torvalds/linux/blob/master/scripts/extract-vmlinux
[20] (2019). *Radare2: Framework for Reverse Engineering and Analysing Binaries*. Accessed: Aug. 9, 2021. [Online]. Available: https://www.radare.org/
[21] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "HFL: Hybrid fuzzing on the Linux kernel," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, Feb. 2020, pp. 1–17.
[22] D. Song, F. Hetzelt, J. Kim, B. B. Kang, J.-P. Seifert, and M. Franz, "Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints," in *Proc. USENIX Secur. Symp. (SEC)*, Aug. 2020, pp. 2541–2557.
[23] S. Pailoor, A. Aday, and S. Jana, "Moonshine: Optimizing OS fuzzer seed selection with trace distillation," in *Proc. USENIX Secur. Symp. (SEC)*, Aug. 2018, pp. 729–743.
[24] S. M. S. Talebi, H. Tavakoli, H. Zhang, Z. Zhang, A. A. Sani, and Z. Qian, "Charm: Facilitating dynamic analysis of device drivers of mobile systems," in *Proc. USENIX Secur. Symp. (SEC)*, Aug. 2018, pp. 291–307.
[25] (2018). *Avoid Speculative Indirect Calls in Kernel*. Accessed: Aug. 9, 2021. [Online]. Available: https://lwn.net/Articles/742756/
[26] J. Horn. (2018). *Reading Privileged Memory With a Side-Channel*. Accessed: Aug. 9, 2021. [Online]. Available: https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html
[27] (2021). *Tail Call*. Accessed: Aug. 9, 2021. [Online]. Available: https://en.wikipedia.org/wiki/Tail_call
[28] Google. (2021). *KernelAddressSanitizer (KASAN)*. Accessed: Aug. 9, 2021. [Online]. Available: https://github.com/google/kasan
[29] Development Tools for the Kernel. (2021). *The Undefined Behavior Sanitizer—UBSAN*. Accessed: Aug. 9, 2021. [Online]. Available: https://www.kernel.org/doc/html/latest/dev-tools/ubsan.html
[30] Google. (2021). *KMSAN (KernelMemorySanitizer)*. Accessed: Aug. 9, 2021. [Online]. Available: https://github.com/google/kmsan
[31] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for *in-vivo* multi-path analysis of software systems," *ACM SIGPLAN Notices*, vol. 46, no. 3, pp. 265–278, 2011.
[32] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 1–39, 2018.
[33] M. Jurczyk. (Jun. 2018). *Detecting Kernel Memory Disclosure With X86 Emulation and Taint Tracking*. Accessed: Aug. 9, 2021. [Online]. Available: http://pirate-network.com/pocorgtfo/pocorgtfo18/bochspwn_reloaded.pdf
[34] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, Feb. 2005, pp. 3–4.
[35] J. Hertz and T. Newsham. (2019). *TriforceAFL: AFL/QEMU Fuzzing With Full-System Emulation*. Accessed: Aug. 9, 2021. [Online]. Available: https://github.com/nccgroup/TriforceAFL
[36] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "DIFUZE: Interface aware fuzzing for kernel drivers," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, Oct. 2017, pp. 2123–2138.
[37] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 754–768.
[38] V. J. M. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Trans. Softw. Eng.*, vol. 47, no. 11, pp. 2312–2331, Nov. 2021.
[39] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "LAVA: Large-scale automated vulnerability addition," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2016, pp. 110–121.
[40] J. Pewny and T. Holz, "EvilCoder: Automated bug insertion," in *Proc. Annu. Conf. Comput. Secur. Appl. (ACSAC)*, Dec. 2016, pp. 214–225.
[41] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," *ACM Meas. Anal. Comput. Syst.*, vol. 4, no. 3, pp. 1–29, Dec. 2020.

MINGI CHO received the B.S. degree in computer engineering from Pusan National University, Busan, South Korea, in 2017. He is currently pursuing the Ph.D. degree with the Laboratory of Information Security, Yonsei University, Seoul. His research interests include software and system security, fuzzing, and memory safety.

HOYONG JIN received the B.S. degree in computer and information security from Sejong University, Seoul, South Korea, in 2020. He is currently pursuing the M.S. degree with the Laboratory of Information Security, Yonsei University, Seoul. His research interests include software and system security, fuzzing, and exploitation.

DOHYEON AN received the B.S. degree in computer science and engineering from Pusan National University, Busan, South Korea, in 2020. He is currently pursuing the M.S. degree with the Laboratory of Information Security, Yonsei University, Seoul. His research interests include software and system security, fuzzing, and memory safety.



TAEKYOUNG KWON (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer science from Yonsei University, Seoul, South Korea, in 1992, 1995, and 1999, respectively.

From 1999 to 2000, he was a Postdoctoral Research Fellow with the University of California, Berkeley, CA, USA. From 2001 to 2013, he was a Professor of computer engineering with Sejong University, Seoul. He is currently a Professor of information security with Yonsei University, where he is also the Director of the Information Security Laboratory. His research interests include authentication, cryptographic protocols, network security, software and system security, usable security, and adversarial machine learning. He is a member of ACM and USENIX. He serves on the Director Board for the Korea Institute of Information Security and Cryptology. He also serves on the Editorial Committee for the Korean Institute of Information Scientists and Engineers.