# FastSyzkaller: Improving Fuzz Efficiency for Linux Kernel Fuzzing

IOP Conf. Series: Journal of Physics: Conf. Series **1176** (2019) 022013 doi:10.1088/1742-6596/1176/2/022013

# FastSyzkaller: Improving Fuzz Efficiency for Linux Kernel Fuzzing

**Dan Li[1,*], Hua Chen[2]**

[1] National Key Laboratory of Science and Technology on Information System Security, Beijing, China
[2] National Key Laboratory of Science and Technology on Information System Security, Beijing, China

*Corresponding author e-mail: 13681216412@139.com

**Abstract**. Linux is one of the most popular operating systems and enjoys a large number of subscribers on a world-wide scale. To ensure its stability and reliability, fuzzing is an effective method to trigger crashes hidden in the Linux kernel. In this paper, we apply N-Gram model to extract vulnerable program behaviours and dig out vulnerable patterns to guide the test case generation phase of the traditional fuzzing technique so as to improve the fuzzing efficiency from the Linux system call aspect. The experiment is implemented on the basis of a Google well-known open-source project Syzkaller, a fuzzer that targeted to the Linux kernel and other OS kernels as well. It is an unsupervised coverage-guided kernel fuzzer, set up with a manager, a fuzzer and an executor. This paper proposes FastSyzkaller which combines Syzkaller with N-Gram model in its fuzzer to optimize the test case generation process to improve the fuzzing efficiency. In 4 weeks, FastSyzkaller exposes 29 different crash types. However, Syzkaller only exposes 12 types. FastSyzkaller also produces crashes 3x faster than Syzkaller, also with 3x more unique crashes from the quantity perspective.

## 1. Introduction

Linux systems are widely used in home and enterprise desktops, embedded systems, supercomputers, servers and smart phones. The Linux kernel is an open-source monolithic computer operating system kernel. It is also a piece of software that is exposed to untrusted user input, developed by contributors from worldwide, which is an important target for fuzzing.

Fuzzing feeds huge numbers of random inputs to the targeted programs and watches for crashes[1]. For the Linux kernel fuzzing, Trinity[2] is a template-based fuzzer which tests system calls in an intelligent way that is driven by per-system call templates. The Linux kernel is highly-profiled such as its file systems so that it is also worth of writing specific, template-based fuzzers which can generate inputs from built-in knowledge about the templates for the kernel. However, Trinity lacks of coverage feedback guidance due to the slow speed of the coverage tools available at that time.

Around 2016, Dmitry Vyukov and a Google team brought coverage-guided fuzzing into Linux kernel and published a tool named Syzkaller[3]. Syzkaller is an unsupervised coverage-guided kernel fuzzer targeted to the Linux kernel at the very beginning. It uses a hybrid approach that both relies on templates which indicate the argument domains for each system call and the feedback from code

coverage information to guide the fuzzing procedure[4]. Syzkaller is now still under active development and contributors continue to subscribe to it to help improve its performance.

Syzkaller uses declarative description of syscalls to present programs that it executes. It generates, mutates, minimizes, serializes and de-serializes programs which are sequences of system calls. The syscalls are translated into runnable code by syz-extract function. By collecting the sequential syscalls of all crashed and non-crashed test cases that Syzkaller executes and provides, vulnerable program behaviors can be partly represented by a sequence of system calls, which is also called vulnerable patterns in this paper.

This paper applies N-Gram model to extract vulnerable system call patterns in the Linux kernel. Moreover, this paper further proposes FastSyzkaller which leverages the learnt vulnerable system call patterns to guide the test case generation phase of the open-source fuzzer Syzkaller to improve the fuzzing efficiency targeted to the Linux kernel.

This paper content as follows. Sec.2 introduces the background on the Linux kernel fuzzing as well as the fuzzing tool Syzkaller. We dedicate Sec.3 to explain the experimental designs and evaluation. Experimental setup is detailed in Sec.4, including collecting vulnerable patterns, guiding the test case generation phase of Syzkaller and evaluating the performance of FastSyzkaller. In Sec.5, we apply FastSyzkaller to crash reproduction and compare it with Syzkaller. What's more, Sec.5 also includes the results and conclusion.

## 2. Background

### *2.1. Fuzzing the Linux Kernel*

System call fuzzers can be traced as far back as 1991[5]. Via writing apps to bomb system call inputs with garbage data, people succeeded in crashing a variety of operating systems[2]. For the Linux kernel, the template-based fuzzer Trinity is one of the most effective fuzzers which is driven by per-system call templates. Although Trinity has considered feedback-guided fuzzing, the coverage tools are too slow to support the function at that time.

With the upgrading of the compiler functions, coverage-guided fuzzing has appeared, which can operate without specific templates towards the targets under test. American Fuzzy Lop[6] (also known as AFL) and LLVM's LibFuzzer[7] are two of the most representative coverage-guided fuzzers. The coverage information is exposed via an instrumented build of the binary under test. These fuzzers mutate existing test cases randomly and save new inputs to the corpus if it can enlarge the code coverage.

Syzkaller both relies on templates to indicate the argument domains for each system call and uses coverage information to guide the fuzzing procedure. With the hybrid approach, Syzkaller shows pretty good results. As a consequence, we choose Syzkaller as our fundamental Linux kernel fuzzing tool.

### *2.2. Syzkaller and its Basic Methodology*

Syzkaller is currently one of the most widely-used system call fuzzers towards the Linux kernel. It was firstly proposed and published as an open-source project by Dmitry Vyukov and a Google team in 2016. The early results look pretty promising that it covers over 150 different kinds of bugs in the Linux mainline kernel in several months[4]. Nowadays, Syzkaller has also been extended to support other OS such as Akaros, FreeBSD, NetBSD, Windows and so forth.

Syzkaller is an unsupervised and coverage-guided kernel fuzzer. With the rapidly development of the compiler and the kernel support, the kernel module KCOV provides Syzkaller with code coverage collection for coverage-guided fuzzing. The complimentary compiler support was added in gcc version 231296. KCOV collects coverage that is function of syscall inputs and exposes kernel code coverage in a form which is suitable for coverage-guided fuzzing[8].

The complete structure for Syzkaller is shown in Figure 1. Syzkaller is made up of by three main processes: a syz-manager, a syz-fuzzer and a syz-executor. The instrumented kernel under test runs over a set of QEMU virtual machines, with syz-fuzzer and syz-executor running inside the virtual machines.

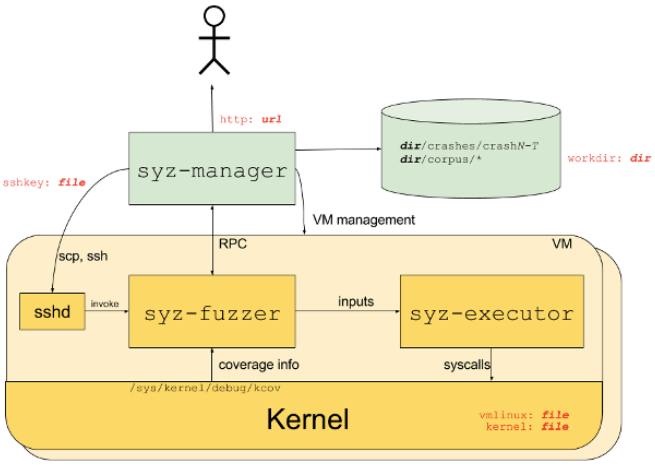


**Figure 1.** Process Structure for Syzkaller System.

When Syzkaller starts to work, the syz-manager firstly interactive with the operator and starts, monitors several virtual machines instances. Syz-manager starts the syz-fuzzer process inside of the VM via RPC communication. The configuration file and the existing corpus (if it exists) are transported to the syz-fuzzer to guide the fuzzing procedure.

The syz-fuzzer guides the fuzzing process including test case generation, mutation and minimization. It continuously receives coverage information from the kernel and sends inputs to syz-executor for real execution. Syz-fuzzer also sends inputs which triggers new coverage back to the syz-manager to update the corpus.

Syz-executor is a simple process which is designed not to interfere with the fuzzing procedure. The fuzzer and the executor communicates by IPC. It only receives inputs from the fuzzer, executes every syscall in the 'Prog' and sends the results back to the fuzzer.

### *2.3. Test Inputs of Syzkaller*

Syzkaller uses declarative description of syscalls to present executable test cases. An executable sequence of syscalls can form the structure 'Prog' in Syzkaller which is a single input for one execution. Numbers of Progs pack together and form the test case corpus. An example of the executable Prog structure is shown in Figure 2. Textual syscall descriptions are translated into executable code by syz-executor.

```
mmap(&(0x7f0000000000/0x860000)=nil, 0x860000, 0x3, 0x44010, 0xffffffffffffffff, 0x0)
r0 = socket$inet_tcp(0x2, 0x1, 0x0)
bind$inet(r0, &(0x7f00006c8000)={0x2, 0x0, @multicast1=0xc0000001, [0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0]}, 0x10)
connect$inet(r0, &(0x7f0000857000-0x10)={0x2, 0x0, @loopback=0x7f000001, [0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0]}, 0x10)
getsockopt$sock_cred(r0, 0x1, 0x11, &(0x7f000085e000)={<r1=>0x0, 0x0, 0x0}, &(0x7f000085e000)=0xc)
mmap(&(0x7f0000860000/0x1000)=nil, 0x1000, 0x3, 0x32, 0xffffffffffffffff, 0x0)
ioctl$int_in(r0, 0x5452, &(0x7f0000861000-0x8)=0x676)
fcntl$setownex(r0, 0xf, &(0x7f0000353000+0xe1c)={0x0, r1})
```

**Figure 2:** An Example of the Prog Structure.

As for the syscall order in the syscall sequence, Syzkaller provides a mechanism to add new syscalls into a Prog aiming to trigger new coverage. The syz-fuzzer maintains a priority table to calculate call-to-call priorities. For a given pair of calls X and Y, if the additional syscall Y be added into the program after X and can give new coverage, Y is given a high priority.

The priority calculation contains both static and dynamic algorithms. The static component is based on analysis of argument types. The dynamic is based on the frequency of occurrence of a particular pair of syscalls in a single program in corpus.

According to the priority table and enabled syscall list, the syz-fuzzer will establish a choice table to do a weighted choice of a syscall for a given syscall based on call-to-call priorities and a set of enabled syscalls. This is how the order of syscalls is decided.

### *2.4. Mutation Strategy for Syzkaller Test Cases*

Syzkaller mutates the existing test case every time it executes a Prog in the following strategy. In one percent probability, the test case is spliced with another Prog from the corpus. In 20 out of 31 probability, Syzkaller might insert a new syscall into the test case which is chosen from the choice table according to the priorities. In 10 out of 11 probability, it may change arguments of a system call. The default action is to remove a random call from the Prog.

What's more, Syzkaller also provides a minimization strategy after the test case is executed to minimize the Prog into an equivalent program using the equivalence predicate function pred. It tries to remove all system calls one-by-one for simplification attempt.

### *2.5. N-Gram Model*

Syzkaller executes and provides test cases which are sequences of syscalls from the corpus. Some of them can lead to crashes while some of them can't. Vulnerable program behaviors may hide inside the sequences of syscalls in crashed test cases which is the real execution data from the Syzkaller corpus.

N-Gram model is widely and mostly used in Natural Language Toolkit, in the fields of computational linguistics and probability. By collecting from a text or a corpus, we own a continuous sequence of items which can be divided into n-gram tuples.

In this paper, the N-Gram model is used to extract vulnerable system call patterns. We aim to grub N-Gram sequential system call patterns in the real execution traces which may be potentially vulnerable so as to guide the test case generation phase in the fuzzing procedure and further improve the kernel fuzzing efficiency.

## 3. Implementation and Evaluation

### *3.1.* Implementation Overview

We set up and run the original Syzkaller to fuzz the Linux kernel, gather the test cases separately which trigger different kinds of crashes as well as those test cases which do not trigger crash. With all the test cases, we obtain thousands of sequential syscalls including the vulnerable system call patterns.

With a variety of test cases collected especially those crashed, we further apply N-Gram model to the test inputs, collect a set of vulnerable sequential system call patterns with length of n system calls. We tried setting n from three to ten and finally we decide to choose n as five. In FastSyzkaller, we choose n as five in the N-Gram model to generate new test cases based on the given five-gram patterns and pack them into the corpus.

In the following fuzzing stages, we aim to generate new test cases which contains the sequential vulnerable system call patterns that we extracted by the N-Gram model and pack them together into a corpus. With an aim to keep the order of the vulnerable system call pattern in each test case, FastSyzkaller can only insert reasonable syscalls with high priorities after the given test inputs or change arguments of the system calls when mutating the existing test cases. What's more, it cannot remove or mutate any of the given syscalls in the syscall sequence, in order not to change the order of the vulnerable syscall patterns.

Via generating test cases in the way that we defined, we can make sure that the newly-generated test cases of FastSyzkaller contain the pre-learnt N-Gram vulnerable system call patterns to improve the Linux kernel fuzzing efficiency of Syzkaller. What's more, FastSyzkaller can still insert new system calls to the test cases and change arguments to improve its syscall diversity and coverage.

### *3.2.* Evaluation for FastSyzkaller Performance

In order to evaluate the performance of our fuzzing tool FastSyzkaller, we carry out experiments to compare it with Syzkaller targeted to the same version of the Linux kernel.

For a fuzzer, one of the most important criteria is the number of crashes that it exposes. During the same time budget, the number of crashes also indicates the speed that it triggers crashes[9].Despite of the amount of crashes that it triggers, the crash type is also a very significant evaluation factor to show the diversity of all the crashes that the tool exposes.

## 4. Experimental Setup

### *4.1.* Test cases Pre-collection

In order to collect enough amount of sequential system calls in the real execution traces for further extracting N-Gram vulnerable system call patterns, we need to execute Syzkaller for test cases pre-collection especially those crashed. In our experiment, we execute Syzkaller targeted to the Linux kernel for 2 weeks and in total we collect 5429 test cases including 652 crashed and 4777 non-crashed.

### *4.2.* N-Gram Vulnerable Pattern Extraction

We apply N-Gram model to those test cases which are executed and stored by Syzkaller because each test case is a sequence of system calls from the corpus.

In this paper, we tried setting n from three to ten, aiming to find out the best choice for vulnerable pattern length. If the length is too short, it is too weak to figure out the vulnerability mechanism. However, if the length n is set too long, the long syscall patterns may lose its universality.

To be more specific, we extracted trigram, four-gram and five to ten-gram system call patterns differently. We separately do statistical analysis on these N-Gram patterns and we found that when n is set as five, the data distribution of the vulnerable patterns is fairly well-distributed of all choices.

### *4.3.* Generation of Corpus

With the vulnerable five-gram system call patterns, we aim to guide the test case generation phase of FastSyzkaller so that the newly-generated test cases must contain the sequential vulnerable patterns and new system calls can still be inserted after the sequences.

According to the specific features of Syzkaller, it can splice Progs with each other, insert new syscall, change arguments of a syscall and remove a random call from the test case every time it mutates the test case. There is also a minimization strategy in Syzkaller to minimize the Prog into equivalent program.

With the attempt to generate new test cases which include the exactly vulnerable syscall patterns that we extract by N-Gram model, we firstly leverage each vulnerable five-gram pattern to form one test case and pack all of the different test cases into the corpus before we start FastSyzkaller. Once FastSyzkaller starts to work, it checks if the corpus exists and it loads the test cases from the corpus which we previously packed. With the given test cases which include vulnerable system call patterns, FastSyzkaller can insert new syscalls after the sequential vulnerable patterns or mutate the arguments of any system call. Unlike Syzkaller, it cannot remove any random syscall or splice Progs with each other to avoid disorganizing the order of the given vulnerable pattern. Minimization strategy is also forbidden for maintaining the proper order of the system calls in the test cases.

### *4.4.* FastSyzkaller Performance Evaluation

To evaluate the fuzzing efficiency of FastSyzkaller, we carry out experiments to compare FastSyzkaller with the original Syzkaller. We set the same time budget as 4 weeks and we run Syzkaller and FastSyzkaller under the same environment with the same configurations targeted to the same version of Linux kernel.

## 5. Results and Conclusion

We have collected all executed test cases including those crashed and those not crashed from the Syzkaller corpus. We apply N-Gram model to extract vulnerable system call patterns and further illustrate these patterns. We also show evaluation results for FastSyzkaller and compare it with Syzkaller. Finally, we draw conclusions about all the work that this paper proposed.

### *5.1.* Execution traces

Via fuzzing the Linux kernel using Syzkaller, we collected 4777 test cases that do not trigger any crash during execution and 652 test cases that lead to crashes when executed by Syzkaller. Among the 652 crashes test cases, the total useful crash type that they exposed is 14.

In the next step, we apply N-Gram model to the above two kinds of test cases separately and extract five-gram system call patterns from those traces.

In brief, we totally collect 652 crashed traces, extract 11575 five-gram patterns from the vulnerable test cases. In the meanwhile, 4777 test cases which do not lead to any crash also provides 179617 five-gram patterns.

*5.2.* Analysis on Vulnerable Patterns

We calculated the number of occurrences of all different five-gram patterns. We also rank all the five-gram occurrence times differently. We show the top 10 vulnerable five-gram patterns in crashed programs in Table 1.

**Table 1:** Top 10 Vulnerable Five-gram Patterns.

| **Vulnerable Five-gram Pattern** |
|---|
| (‘setsockopt$inet_tcp_int’, ‘bind$int’, ‘connect$inet’, ‘setsockopt$inet_tcp_TCP_CONGESTION’, ‘sendto’) |
| (‘getsockopt$SO_PEERCRED’, ‘rt_tgsigqueueinfo’, ‘socket$inet_tcp’, ‘ioctl$EVIOCSFF’, ‘setsockopt$SO_ATTACH_FILTER’) |
| (‘rt_tgsigqueueinfo’, ‘socket$inte_tcp’, ‘ioctl$EVIOCSFF’, ‘setsockopt$SO_ATTACH_FILTER’, ‘setsockopt$inet_tcp_int’ ) |
| (‘mmap’, ‘mmap’, ‘mmap’, ‘mmap’, ‘recvmmsg’) |
| (‘setsockopt$SO_ATTACH_FILTER’, ‘setsockopt$inet_tcp_int’, ‘setsockopt$inet_tcp_int’, ‘bind$inet’, ‘connect$inet’) |
| (‘ioctl$sock_SIOCOUTQ’, ‘iotcl$sock_inet6_tcp_SIOCOUTQNSD’, ‘sendmsg’, ‘getsockopt$inet_sctp_SCTP_GET_LOCAL_ADDRS’, ‘getsockopt$inet_sctp6_SCTP_DEFAULT_SNDINFO’) |
| (‘getgrp’, ‘getsockopt$SO_PEERCRED’, ‘rt_tgsigqueueinfo’, ‘socket$inet_tcp’, ‘iotcl$EVIOCSFF’) |
| (‘mmap’, ‘getgrp’, ‘getsockopt$SO_PEERCRED’, ‘rt_tgsigqueueinfo’, ‘socket$inet_tcp’) |
| (‘sendto’, ‘openat$vga_arbiter’, ‘ioctl$sock_SIOCOUTQ’, ‘ioctl$sock_inte6_tcp_SIOCOUTQNSD’, ‘sendmsg’) |
| (‘sendto’, ‘ioctl$sock_SIOCOUTQ’, ‘ioctl$sock_inte6_tcp_SIOCOUTQNSD’, ‘sendmsg’, ‘shutdown’) |

We also present top 10 normal five-gram patterns in non-crashed programs in Table 2.

**Table 2:** Top 10 Normal Five-gram Patterns.

| **Normal Five-gram Pattern** |
|---|
| (‘mmap’, ‘mmap’, ‘mmap’, ‘mmap’, ‘mmap’) |
| (‘mmap’, ‘open’, ‘perf_event_open’, ‘openat$hidraw0’, ‘perf_event_open’) |
| (‘unshare’, ‘mmap’, ‘open’, ‘perf_event_open’, ‘openat$hidraw0’ ) |
| (‘open’, ‘perf_event_open’, ‘openat$hidraw0’, ‘perf_event_open’, ‘iotcl$PERF_EVENT_IOC_PERIOD’) |
| (‘ioctl$EVIOCGKEY’, ‘perf_event_open’, ‘openat$hidraw0’, ‘perf_event_open’, ‘iotcl$PERF_EVENT_IOC_PERIOD’) |
| (‘unshare’, ‘mmap’, ‘getsockopt$inet_stcp_SCTP_RESET_STREAMS’, ‘open’, ‘mmap’) |
| (‘openat$vga_arbiter’, ‘mmap’, ‘open’, ‘perf_event_open’, ‘openat$hidraw0’) |
| (‘open’, ‘perf_event_open’, ‘socket$inet_tcp’, ‘openat$hidraw0’, ‘perf_event_open’) |
| (‘getsockopt$inet_stcp_SCTP_RESET_STREAMS’, ‘open’, ‘mmap’, ‘ioctl$EVIOCGKEY’, ‘perf_event_open’) |
| (‘unshare’, ‘openat$rfkill’, ‘openat$vga_arbiter’, ‘mmap’, ‘open’) |

Via ranking the occurrence times of both the vulnerable and normal patterns, we found that the vulnerable patterns and normal patterns vary a lot from each other.

By carefully observing the ranked five-gram lists, we can see that the majority of vulnerable patterns contain system calls that are relevant to the setsockopt( ) function. When sockets connect and transport messages between each other, system calls such as send, sendto, sendmsg, recvmmsg are

frequently used and are more likely to trigger unpretended program behaviors such as lost connection or even crashes.

Although it seems that the ioctl( ) function has some relationship with the vulnerable patterns. However, the ioctl( ) system call function also appears for many times in normal five-gram patterns. This function is used to manipulate the underlying device parameters of special files and control the I/O devices. It is not one of the main factors that leads to a crash.

### *5.3.* Evaluation on FastSyzkaller Performance

For a fuzzer, one of the most obvious criteria to judge whether it works well is the number of crashes that it exposes. For FastSyzkaller, during 4 weeks' execution, it totally triggers 4813 crashes, with an average speed of 7 crashes per hour. For Syzkaller, it only exposes 1446 crashes in 4 weeks with 2 crashes per hour. FastSyzkaller performs 3x faster than Syzkaller, also with 3x more unique crashes from the quantity perspective.

Another main factor to evaluate the performance of a fuzzer is the crash types that it exposes. During 4 weeks' execution, FastSyzkaller exposes 29 useful crash types which are listed in Table 3 including use-after-free crashes, kernel bug, general protection fault and so force.

However, Syzkaller only exposes 12 crash types during the same time budget. The crash types that Syzkaller triggers are listed in Table 3 as Number 1, 5, 6, 7, 12, 14, 15, 22, 23, 24,27 and 29.

**Table 3:** Crash Types Exposed by FastSyzkaller.

| Number | Crash Types for FastSyzkaller |
|---|---|
| 1 | INFO: rcu detected stall |
| 2 | INFO: rcu detected stall on_each_cpu |
| 3 | INFO: recovery required on readonly filesystem |
| 4 | INFO: rcu_sched detected stalls on CPUs/tasks |
| 5 | KASAN: slab-out-of-bounds Read in prepare_signal |
| 6 | KASAN: use-after-free Read in cleanup_timers |
| 7 | KASAN: use-after-free Read in do_get_mompolicy |
| 8 | KASAN: use-after-free Read in ip_check_mc_rcu |
| 9 | KASAN: use-after-free Read in unix_stream_data_wait |
| 10 | KASAN: use-after-free Read in add_grec net/ipv4/igmp.c:473 |
| 11 | KASAN: use-after-free Write in move_expired_inodes |
| 12 | WARNING in dev_watchdog |
| 13 | WARNING in get_futex_key |
| 14 | WARNING in hrtimer_forward |
| 15 | WARNING in inet_sock_destruct |
| 16 | WARNING in mark_buffer dirty |
| 17 | WARNING in rcu_process_callbacks |
| 18 | WARNING in skb_try_coalesce |
| 19 | divide error in __tcp_select_window |
| 20 | general protection fault in __pskb_trim |
| 21 | general protection fault in _kmalloc_node_track_caller |
| 22 | general protection fault in check_timers_list |
| 23 | general protection fault in cleanup_timers |
| 24 | general protection fault in skb_release_data |
| 25 | Kernel BUG at net/core/skbuff.c:2241! |
| 26 | Kernel BUG at net/ipv4/tcp_output.c:2567! |
| 27 | Kernel BUG at net/ipv4/tcp_output.c:LINE! |
| 28 | Kernel panic – not syncing: panic_on_warn set |
| 29 | Kernel panic: Couldn't open N_TTY_ldisc |

## 6. Conclusion

In this paper, we leverage N-Gram model to extract vulnerable system call patterns and propose FastSyzkaller for improving the Linux kernel fuzzing efficiency. By using an open-source Linux-kernel targeted fuzzing tool Syzkaller, we collected sequential system call patterns in all test inputs and made use of N-Gram model to handle these data. We combined these vulnerable patterns into FastSyzkaller as the given corpus so that it can generate new test cases based on pre-defined vulnerable patterns. It improves fuzzing efficiency in triggering crashes. Furthermore, we set up experiments and carried out evaluations for comparing FastSyzkaller with Syzkaller. Results shows that FastSyzkaller performs 3x better in both crash types as well as total crash amount and fuzzing efficiency. FastSyzkaller also exposes more precious unique crash types than Syzkaller.

## Acknowledgments

We would like to thank the anonymous reviewers for their comments. We would also like to appreciate Dmitry Vyukov and the Google team for sharing their work Syzkaller and inspiring us to go on research of Linux kernel fuzzing and vulnerability discovery at a large scale.

This research is supported by the National Key Laboratory of Science and Technology on Information System Security.

## References

[1] Junjie Wang, Bihuan Chen, Lei Wei, Yang Liu, Skyfire: Data-Driven Seed Generation for Fuzzing[C], Security and Privacy (SP), 2017 IEEE Symposium on. 2017: 579–594.
[2] Information on http://github.com/kernelslacker/trinity.
[3] Information on http://github.com/google/syzkaller.
[4] Information on https:/lwn.net/Articles/677764/
[5] B. P. Miller, L. Fredriksen, and B. So, “An empirical study of the reliability of unix utilities,” Commun. ACM, vol 33, no. 12, pp. 32-44,1990.
[6] Information on https:/lcamtuf.coredump.cx/afl/
[7] Information on https:/github.com/Dor1s/libfuzzer-workshop/
[8] Information on https:/lwn.net/Articles/671640/
[9] Cha S K, Woo M, Brumley D, Program-adaptive mutational fuzzing[C], Security and Privacy (SP), 2015 IEEE Symposium on. 2015: 725–741.