Industry Practice of Coverage-Guided Enterprise Linux Kernel Fuzzing

Heyuan Shi
KLISS, BNRist, Tsinghua University
China

Runzhe Wang
KLISS, BNRist, Tsinghua University
China

Ying Fu
KLISS, BNRist, Tsinghua University
China

Mingzhe Wang
KLISS, BNRist, Tsinghua University
China

Xiaohai Shi
Operating System Group, Alibaba Inc
China

Xun Jiao
Villanova University
USA

Houbing Song
Embry-Riddle Aeronautical University
USA

Yu Jiang*
KLISS, BNRist, Tsinghua University
China

Jiaguang Sun
KLISS, BNRist, Tsinghua University
China

ABSTRACT

Coverage-guided kernel fuzzing is a widely-used technique that has helped kernel developers and testers discover numerous vulnerabilities. However, due to the high complexity of application and hardware environment, there is little study on deploying fuzzing to the enterprise-level Linux kernel. In this paper, collaborating with the enterprise developers, we present the industry practice to deploy kernel fuzzing on four different enterprise Linux distributions that are responsible for internal business and external services of the company. We have addressed the following outstanding challenges when deploying a popular kernel fuzzer, syzkaller, to these enterprise Linux distributions: coverage support absence, kernel configuration inconsistency, bugs in shallow paths, and continuous fuzzing complexity. This leads to a vulnerability detection of 41 reproducible bugs which are previous unknown in these enterprise Linux kernel and 6 bugs with CVE IDs in U.S. National Vulnerability Database, including flaws that cause general protection fault, deadlock, and use-after-free.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Kernel fuzzing, enterprise Linux, bug detection

ACM Reference Format:
Heyuan Shi, Runzhe Wang, Ying Fu, Mingzhe Wang, Xiaohai Shi, Xun Jiao, Houbing Song, Yu Jiang, and Jiaguang Sun. 2019. Industry Practice of Coverage-Guided Enterprise Linux Kernel Fuzzing. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3338906.3340460

*Yu Jiang is the correspondence author.

1 INTRODUCTION

Coverage-guided kernel fuzzing is a widely-used approach to ensure the correctness of the operating system (OS), which exploits coverage information to guide the generation and mutation of system call (syscall) series with their input parameters, and catch the abnormal kernel behavior during the execution. A number of kernel fuzzing tools have been released, e.g., TriforceLinuxSyscallFuzzer [2, 4], kAFL[31] and syzkaller [33]. As for academic research, many researchers focus on improving the efficiency of kernel fuzzing on specific fuzzing targets, e.g., file systems [37] and drivers [17, 32], with domain knowledge. Moreover, many other validation techniques are also integrated into the fuzzing process, such as symbolic execution [21, 36] and static analysis [19, 29].

The effectiveness of kernel fuzzers has attracted many industrial users for real practice. Unfortunately, it is challenging and very difficult to apply kernel fuzzing on enterprise Linux distributions. The gap mainly locates in two aspects. First, most academic kernel fuzzers are implemented as a prototype without the support for complex industrial applications and hardware environment, which is not acceptable for enterprise uses. Moreover, the current kernel fuzzing tools such as syzkaller in industry try to keep up with the latest mainline Linux, without considering the adaptation of enterprise Linux kernel, which usually consists of many customized features or inconsistent configurations.

In particular, for most industrial users, they usually maintain their own Linux distribution (distro) based on a long-term support (LTS) version of Linux kernel, which is stable but old. However, the old versions of kernel lead to the absence of some features and components required by the state-of-the-art kernel fuzzers. For example, the feature of coverage support is missed in many enterprise Linux distributions based on old version of kernel, e.g., Linux-3.10 used in CentOS 7. As a result, even with those kernel fuzzers in hand, they may still fail to fuzz enterprise Linux kernel with the guidance of coverage. Besides that, the continuous kernel fuzzing integrated into existing continuous integration (CI) system is highly appreciated by engineers in practice, but it is difficult to implement, with the increasing complexity in kernel updates and host environment.

In this paper, we adapt the syzkaller, one of the widely-used kernel fuzzers, to the enterprise Linux kernel used in various real business services, and address several main challenges, including the absence of coverage support, kernel boot fails in VM, shallow bugs in running syzkaller, and continuous fuzzing complexity. For each challenge, we provide corresponding solutions and develop supporting tools. For example, we implement a framework to mitigate the continuous fuzzing complexity. We also propose a method to add coverage support in the old version of enterprise Linux kernel by backporting the KCOV module in the latest Linux kernel. By successfully using the syzkaller on the development of enterprise Linux kernel and the accompanied patches to the existing test platform in companies, we detect 41 reproducible bugs in the enterprise Linux kernel, e.g., memory leak, general protection fault, task hung or kernel panic. We also detect 6 vulnerabilities that are assigned with unique common vulnerabilities and exposures (CVE) IDs in U.S. National Vulnerability Database (NVD). The contributions are summarized as follows.

- We identify the typical challenges in deploying kernel fuzzing tools on four enterprise Linux distributions with different business scenarios, i.e., Linux-3.10, Linux-4.9, Linux-4.14-rt and Linux-4.19.
- We address these challenges by developing corresponding solutions including backporting kernel modules, configurations adaption in kernel and virtualization tool, shallow bugs repair, and continuous fuzzing implementation.
- Experimental results show that our proposed solutions can lead to the vulnerability detection of 41 unknown bugs in the enterprise Linux kernel and 6 bugs with CVE IDs in U.S. National Vulnerability Database, including flaws that cause general protection fault, deadlock, and use-after-free.

The rest of this paper is organized as follows. The background of coverage-guided kernel fuzzing and the selected kernel fuzzer are first introduced in Section 2. The fuzzing procedures and the to-be-fuzzed enterprise Linux distributions are presented in Section 3. We show the typical challenges and solutions for each stage of kernel fuzzing in Section 4. Then, the testing results and the bug statistics are given in Section 5. Finally, the discussions related to the observation in kernel fuzzing deployment and conclusion are presented in Section 6 and Section 7, respectively.

2 BACKGROUND

In this section, we first give a brief introduction to coverage-guided kernel fuzzing techniques. Then, we introduce syzkaller, a widely-used kernel fuzzing tool in industry practice.

2.1 Coverage-Guided Kernel Fuzzing

Generally, OS kernels are fuzzed by executing the sequences of syscalls with randomly generated parameter values [18, 24, 25, 28]. Though the basic methodologies of kernel fuzzing are similar, an increasing number of enhanced mechanisms are proposed. For example, the coverage-guided kernel fuzzers integrate coverage feedback to guide the process of fuzzing. In this way, we can reserve test cases which can maximize the code coverage, which shows the superior performance than non-coverage-guided random fuzzers in practice.

Many coverage-guided kernel fuzzers are implemented by porting the AFL [16, 26, 27, 36], a fuzzer for user-space applications, to the kernel-space. Typical coverage-guided kernel fuzzer includes TriforceLinuxSyscallFuzzer [4], which was presented by the nccgroup to perform fuzzing on Linux x86_64 kernels. It is based on TriforceAFL [2], which is a patched version of AFL that supports full-system fuzzing using the virtualization tool of QEMU. The engineers in Oracle also proposed a similar kernel fuzzer based on QEMU and AFL called kernel-fuzzing [3]. Moreover, benefiting from the hardware feature of Intel processor trace, kAFL [31] was proposed to fuzz OS kernels which can trace the execution accurately with low overhead. Xu et. al. [37] presented a feedback-driven fuzzer called JANUS specific to file systems, which explores the two-dimensional input space by mutating meta-data on a large image.

Another widely used coverage-guided kernel fuzzers is syzkaller [33], which is an unsupervised fuzzer developed by Google. Besides collecting code coverage information as the feedback, syzkaller exploits predefined templates to manipulate sequences of syscall based on the syntax of syscall interfaces [14, 15]. Syzkaller has detected thousands of bugs in the Linux kernel [1], and shows the potential to find more kernel bugs along with the mainline Linux kernel development.

Moreover, many academic researchers try to integrate other kernel validation techniques and optimizations with syzkaller, e.g., static analysis and dynamic testing approaches, to further improve the fuzzing performance. For example, Pailoor et. al. [29] proposed a tool integrating static analysis called MoonShine to distill the initial seeds of syzkaller and make corpus close to the practical testing scenario at the beginning of fuzzing. In this way, both explicit and implicit dependencies between sequences of syscall are explored in real-world programs in many Linux test suites, based on the modified kernel syscall trace tool of strace and the static analysis framework of smatch. By the seed distillation, the achieved code coverage for the Linux kernel was improved and several unknown bugs were found. Kim et. al. [21] introduced ALEXKIDD-FUZZER, which integrates syzkaller with concolic analysis and uses symbolic information to guide the fuzzing process. Jeong et. al. [19] proposed RAZZER to detect race bugs in kernels by guiding fuzzing to focus on the potential data race spots. In particular, static analysis is exploited to find the over-approximated potential data race spots, which are implemented by LLVM and the modified version of the points-to analysis framework of SVF. Then, both the single- and multi- thread dynamic analysis are performed to trigger a race bug, by syzkaller and deterministic thread interleaving implemented on the hypervisor in QEMU.

Besides the traditional validation techniques, other popular techniques such as machine learning are also integrated to improve the performance of kernel fuzzing. For example, You et. al. [38] proposed a semantics-based fuzzer called SemFuzz to generate Proof-of-Concept Exploits automatically, which integrates the natural language processing (NLP) and syzkaller. In this way, the NLP

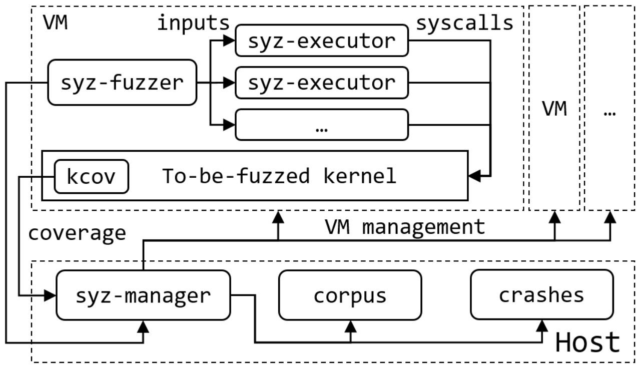

Figure 1: The architecture of syzkaller.

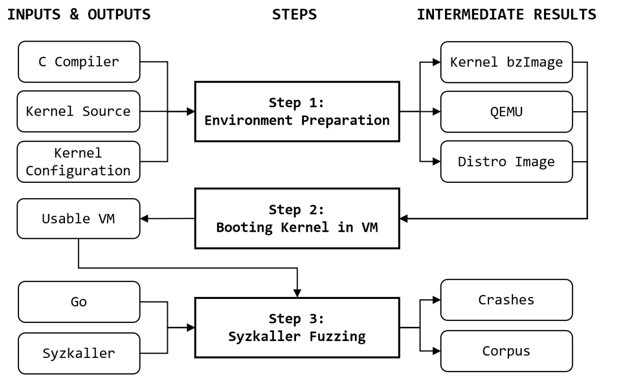

Figure 2: Steps of kernel fuzzing plan.

approach is first exploited to retrieve the semantic information from vulnerability-related patches and text sources. Then, the seed generation and mutation of syzkaller can be guided by the collected semantic information. As a result, it enhances the ability of syzkaller to trigger the known flaws.

2.2 Why Syzkaller?

As shown above, there are many related kernel fuzzers proposed, and their performance seems to be attractive. However, when deploying a kernel fuzzer on enterprise Linux used in the industry scenario, the components required by each fuzzer are not at all satisfied. In the following, we introduce the design of syzkaller, which is supposed to be one of the most industry-friendly kernel fuzzer, and reasons for selecting it as our kernel fuzzer.

The design of syzkaller is presented in Fig. 1. There are mainly three components in syzkaller: syz-manager, syz-fuzzer, and syz-executor. syz-manager is running on the host machine with a stable kernel. It starts, monitors and restarts several virtual machine(VM) instances, and starts a syz-fuzzer process inside of the VMs. Moreover, it is responsible for updating corpus and crashes. The syz-fuzzer process runs inside of presumably unstable VMs. It guides the fuzzing process in each VM instance, in charge of input generation, mutation, minimization, and sending input corpus that trigger new coverage back to the syz-manager process. The syz-fuzzer also starts transient syz-executor processes, and each syz-executor process executes a single test case from the syz-fuzzer, i.e., a sequence of syscalls, and sends the result back.

Syzkaller is selected as the fuzzer to fuzz enterprise Linux kernel based on the following reasons.

- **Availability**: Syzkaller is an open-source software without the restriction for enterprise use, which can reduce the risk as well as the costs of kernel fuzzing in practice. Indeed, syzkaller is released on GitHub with the Apache License 2.0 [33], which is suitable for the commercial use, distribution, and modification.
- **Efficiency**: The performance of syzkaller is considerable and bugs have been found not only on the experimental scenario with the recommended configurations but also in the industry practice. In particular, syzkaller has detected thousands of kernel bugs and over 1200 bugs are fixed by developers while hundreds of bugs are under open, moderation or fix pending status, according to the statistics on syzbot [1].
- **Portability**: Syzkaller is easy to deploy on enterprise test machine. Specifically, syzkaller can perform kernel fuzzing on VM of QEMU or GCE [5], without any extra hardware feature requirements, e.g., Intel-PT needed by kAFL.
- **Maintenance**: Syzkaller is under good maintenance and is updated regularly, to follow the developments of Linux kernel branches and learn from other vulnerability detection tools. In particular, syzkaller performs fuzzing on the mainline Linux kernel branch with the latest compiler [12], e.g., Linux 4.19 with gcc 7.3, Linux 4.20 with clang 8.0 and Linux 5.0 with gcc 9.0, while other fuzzers were last updated in several years ago, e.g., TriforceLinuxSyscallFuzzer.

Besides the advantages above, there are two additional features making syzkaller attractive in industry practice: troubleshooting support and academic future. The chosen fuzzer is expected to have troubleshooting support. Actually, syzkaller owns an active community of Google group that can provide technical help for enterprise engineers. Moreover, it would greatly benefit the research community because there are active research works around it, and industrial users may reduce the potential costs for maintaining the modified version of syzkaller with those academic optimizations.

3 FUZZING PROCEDURES AND TARGETS

In this section, the procedures of deploying syzkaller on fuzz enterprise Linux kernel and the target versions of kernel are introduced.

3.1 Fuzzing Procedures

The traditional kernel fuzzing procedure of syzkaller is presented in Fig. 2. There are 3 steps to deploy syzkaller to fuzz the target enterprise Linux: (1) Environment preparation; (2) Booting kernel in VM; and (3) Syzkaller fuzzing.

In the first step, we need to prepare the following components required by syzkaller, based on which, we can compile the to-be-fuzzed kernel and derive the kernel image, i.e., bzImage. [5]:

- C compiler with coverage support. syzkaller is a coverage-guided fuzzer and therefore it needs the kernel to be built with coverage support, which requires a recent gcc version.

Table 1: Summary of target enterprise Linux distributions.

Distros	Kernel	Provider	Targeted Services
CentOS 7	Linux-3.10	RedHat	General businesses services
D-Linux-4.9	Linux-4.9	Alibaba	Internal enterprise services
RTLinux	Linux-4.14-rt	CRRC	Real-time Ethernet data services on railway
D-Linux-4.19	Linux-4.19	Alibaba Cloud	Cloud elastic compute service

Table 2: Challenges overall.

Step	Challenge	Linux kernel version			
		3.10	4.9	4.14-rt	4.19
1	No support for code coverage in compiler	x			
1	No support for code coverage in kernel	x			
2	Improper configurations of kernel and VM	x	x	x	x
3	Shallow bugs	x	x	x	x
3	Continuous fuzzing	x	x	x	x

- Linux kernel with coverage additions. Besides coverage support in gcc, it also requires the coverage support on the kernel side.
- Virtual environments. Syzkaller performs kernel fuzzing on several slave VMs and therefore it needs enterprise Linux image and virtualization tools such as KVM and QEMU.
- Go toolchain. Syzkaller is written in Go, so the recent Go toolchain is required to build syzkaller.

In the second step, the to-be-fuzzed kernel is booted with prepared distro image in the VM to validate the usability of VM instance prepared for syzkaller. Finally, in the third step of syzkaller fuzzing, syzkaller is deployed on a test machine and performs kernel fuzzing, which outputs crash reports and the corresponding corpus. In summary, step 1 is to prepare the required components, step 2 is to validate the usability of VM instance, and step 3 is to run syzkaller and detect bugs in enterprise Linux kernel.

3.2 Fuzzing Targets

Collaborated with our industrial partners, we deploy syzkaller to fuzz the following enterprise Linux distributions which use the LTS Linux version of 3.10, 4.9, 4.14 and 4.19, as summarized in Table 1.

CentOS [11] is a community supported Linux distribution derived from the Red Hat Enterprise Linux (RHEL), whose current version is CentOS 7. As such, CentOS Linux is a compatible rebuild of the Red Hat Enterprise Linux, in functional compliance with Red Hat's redistribution requirements. CentOS 7 uses the Linux-3.10 as its kernel, and because of its stable, predictable, manageable and reproducible feature, CentOS Linux is widely popular among Linux users for web hosts and small businesses.

D-Linux-4.9 is the enterprise Linux distribution provided by Alibaba, which is responsible for millions of business applications with a very strict requirement on reliability, e.g., search services, running on thousands of servers in Alibaba. Its default kernel is based on the LTS version of Linux-4.9 [7], which adds some enhanced features and optimization.

RTLinux is one of the enterprise Linux distributions used in CRRC Corp. Ltd. (CRRC). The used kernel Linux-4.14-rt in RTLinux is based on the Linux-4.14 with the PREEMPT_RT patches, which is to fulfill the requirements of a real-time system and minimize the amount of kernel code that is non-preemptive [13]. RTLinux is deployed on the embedded devices of real-time railway Ethernet switches provided by CRRC. In this way, there are many safety-critical applications, e.g., critical data transmission of railways, running on the RTLinux with kernel Linux-4.14-rt.

D-Linux-4.19 is an open-source Linux distribution originated by Alibaba operating system team, aiming to deliver OS services with various functionality, high performance and stability to numerous cloud elastic compute service (ECS) customers [10]. Its default kernel is based on the LTS version of Linux-4.19 [9], which adds the features and enhancements specific to the cloud infrastructure and products.

We select these enterprise Linux distributions because of the following reasons. Firstly, they are responsible for the core business of the industrial partners, e.g., real-time data transmission and search services. Secondly, they are widely used on the enterprise devices in practical industrial environments, e.g., thousands of servers running enterprise services and hundreds of switches running railway applications. Thirdly, the running environment of these enterprise Linux distributions is typical and various, e.g., physical machines, virtual machines on the cloud and embedded devices.

4 TYPICAL CHALLENGES AND SOLUTIONS

Though enterprise Linux distributions are all based on the Linux LTS version and syzkaller is one of the most widely used and industry-friendly kernel fuzzer maintained by Google, several challenges appear when we try to perform kernel fuzzing on the target enterprise Linux kernel, as summarized in Table 2. In the following, we present the details of challenges as well as solutions in each step of kernel fuzzing deployment.

4.1 Environment Preparation

In this step, the components of kernel image, i.e., bzImage, distribution image, and virtualization tools are prepared. To derive the kernel image, we initially try to compile the to-be-fuzzed kernel by the default compiler and the kernel configuration file. However, the enterprise Linux providers and users heavily customized enterprise Linux distributions to ensure the stability of applications and services in both domain-specific production environment and development process, e.g., specific gcc compiler, kernel configurations, and software collections. Furthermore, the standard components provided by enterprise Linux are usually stable but outdated, will not satisfy the requirements of syzkaller.

Challenge 1: No support for code coverage in compiler. Syzkaller is a coverage-guided fuzzer which requires the coverage support on the compiler side. However, the default compiler for building enterprise Linux kernel is too old to support coverage feature. Specifically, the feature of code coverage in compiler is supported by gcc 6.1.0 or later [35], via its "-fsanitize-coverage=trace-pc" flag. Though the default compiler supports code coverage for the Linux-4.9, Linux-4.14-rt, and Linux-4.19, the provided compiler in the software repository, i.e., redhat gcc-4.8.5, are too old to support the code coverage for Linux-3.10 used as the default kernel in CentOS 7.

Solution 1: Update gcc to add coverage support. We make gcc support coverage feature by updating its version later than 6.1.0. In particular, gcc is updated from 4.8.5 to 8.2, to compile Linux-3.10. It seems relatively easy but we should notice that the version of compiler cannot be changed in reality due to the stability and maintenance consideration. In this case, we use the currently used gcc (version from 4.8 to 6.0) with the sancov plugin to achieve coverage tracking [34], which inserts a tracing call at the beginning of each basic block of code.

Challenge 2: No support for code coverage in kernel. Besides the compiler, code coverage support is also required on the kernel side for syzkaller. More specifically, Linux kernel requires KCOV module to provide the code coverage feature, which is committed by upstream in the Linux kernel version of 4.6. However, the KCOV module is still absent in Linux-3.10. The same situation appears when performing kernel fuzzing on other old versions of enterprise Linux kernel, e.g., Linux-2.6.33 which is still used in some embedded devices due to the historical burden.

Solution 2: Backporting KCOV to add coverage support. As for the kernel side, the problem becomes relative complex. We have to backport the entire KCOV module to add coverage support into the Linux-3.10 correctly, by changing the related files, e.g., git commits related to KCOV and Makefile, and fixing the conflicts between the latest and old version of the kernel source. The details of backporting the KCOV module to Linux-3.10 are introduced as follows.

(1) The commits related to the KCOV module are backported. In this way, all the commits related to KCOV are derived from the git repository of the latest stable kernel by "git log –grep kcov", and merged into the git repository of Linux-3.10 by using the command "git cherry-pick".
(2) The parameters required to build KCOV module in Linux-3.10 are backported. In this way, we add the parameter of "KBUILD_CFLAGS += -fno-pic" in the file of MAKEFILE to enable the default build option required by KCOV. We add "ARCH_HAS_KCOV if X86_64" in the file of Kconfig to enable the kernel option of "CONFIG_HAS_KCOV" when configuring Linux-3.10. After that, the flags to support code coverage in Makefile.lib are added, e.g., CFLAGS_KCOV = -fsanitize-coverage=trace-pc.
(3) The data structures and the preprocessor directive in the kernel headers required by KCOV are backported. In this way, we first add kcov_mode, kcov_size, *kcov_area and *kcov, which is defined in the latest kernel but not in Linux-3.10, to the structure task_struct in the header of sched.h. We add the static functions of __read_once_size and __write_once_size to enable the macro of READ_ONCE() and WRITE_ONCE() in the compiler.h.
(4) The new functions introduced by the KCOV commits which are not usable for Linux-3.10 are backported or replaced. For example, we replace the function of in_task() in the latest kernel by the function of preempt_count() in the Linux-3.10 because their functionalities are identical.
(5) The coverage support is disabled in the architecture-specific files and low-level components to boot the enterprise Linux kernel with the coverage support successfully in our prepared environment. More specifically, by adding the instruction "KCOV_INSTRUMENT := n" in the Makefile of these kernel module, we disable the coverage support in the kernel modules related to the architecture, e.g., cpu and ia32, and the platform-specific files.

With the steps above, the coverage support is backported into the Linux-3.10. Other enterprise Linux without coverage support can also backport KCOV in terms of these steps. Finally, both compiler and kernel with coverage support are derived by updating and backport respectively.

4.2 Booting Kernel in the VM

In this step, we use the derived kernel bzImage and Linux image to boot the enterprise Linux in QEMU, to test the usability of VM instance of the target kernel. Unfortunately, for all the versions of to-be-tested enterprise Linux kernel, the prepared kernel bzImage using the initial default kernel configurations cannot boot correctly.

Table 3: Errors appear on kernel boot in the VM.

Type	Logs
System reboot	VFS: Unable to mount root fs on unknown-block
Emergency mode	Give root password for maintenance:
Service boot failed	Failed to start Raise network interfaces
Service boot failed	Failed to start LSB: Bring up/down networking
Service boot failed	Failed to start Crash recover kernel arming

Challenge 3: VM instance boot failure. Though syzkaller provides many recommended kernel configurations [8], the kernel is still failed to boot with these recommended configurations. It is because of the improper kernel configuration and QEMU boot command. As a result, the boot process in the VM instance ended with the kernel panic, system reboot, emergency mode or abnormal status of system service, and the corresponding failure logs is presented in Table 3. With manual analysis, we found that the main reasons for kernel boot failed in VM is the absence of a number of kernel configurations and QEMU boot parameters.

Solution 3: Adapt kernel and VM configurations. The enterprise Linux kernel must boot successfully in the VM before performing the kernel fuzzing. We collaborate with distribution image providers to fix the improper kernel configurations and VM boot parameters, and enabling the additional kernel configurations.

Table 4: Kernel configurations adaption for Linux-4.9.

Configuration Descriptions	Enabled Configuration
Enabled additional options for guest kernel support	CONFIG_IP_PNP CONFIG_IP_PNP_DHCP CONFIG_VIRTIO_BLK CONFIG_SCSI_VIRTIO CONFIG_VIRTIO_NET CONFIG_VIRTIO_CONSOLE CONFIG_VIRTIO CONFIG_VIRTIO_PCI CONFIG_VIRTIO_INPUT
SCSI disk support	CONFIG_BLK_DEV_SD
Serial ATA and Parallel ATA drivers	CONFIG_ATA
The Extended 4 (ext4) filesystem	CONFIG_EXT4_FS
Intel(R) PRO/1000 Gigabit Ethernet support	CONFIG_E1000

Listing 1: QEMU command with additional parameters to boot kernel in the VM.

```
qemu-system-x86_64 \
-drive if=none,file=image/img.qcow2,id=hd \
-device virtio-scsi-pci,id=scsi \
-device scsi-hd,drive=hd \
-net user,host=10.0.2.10,hostfwd=tcp::10021-:22
-net nic -nographic \
-kernel arch/x86/boot/bzImage \
-append "crashkernel=auto net.ifnames=0 root=/dev/
    sda1 kvm-intel.nested=1 kvm-intel.
    unrestricted_guest=1 kvm-intel.ept=1 kvm-intel
    .flexpriority=1 kvm-intel.vpid=1 kvm-intel.
    emulate_invalid_guest_state=1 kvm-intel.eptad
    =1 kvm-intel.enable_shadow_vmcs=1 kvm-intel.
    pml=1 kvm-intel.enable_apicv=1 console=ttyS0
    rw earlyprintk=serial slub_debug=UZ vsyscall=
    native rodata=n oops=panic panic_on_warn=1
    panic=600 ima_policy=tcb" \
-enable-kvm -m 2G -smp 4 -cpu host -snapshot
```

For the example of Linux-4.9, we enable the extra configurations which are summarized in Table 4. In particular, we first enable additional options for general support of KVM guest kernel by "make kvmconfig". It is noticed that there is no rule to target 'kvmconfig' for Linux-3.10, because kvmconfig was added into kernel since Linux-3.11 [30]. In this case, we enable the configurations appeared in the kvmconfig by manually editing the kernel configuration file, i.e., .config. Then, we enable the drivers required by the kernel boot in VM. In this way, we enable the SCSI disk support, i.e., CONFIG_BLK_DEV_SD, to mount the root file system which is located on a SCSI disk. Moreover, the file system used for kernel is enabled, e.g., ext4. Finally, the network driver is enabled, e.g., CONFIG_E1000, because the syzkaller requires the secure shell (ssh) and secure copy (scp) services. What's more, we follow the syzkaller tutorial to add more kernel configurations [8] refer to support coverage tracing, syzkaller features and bug detection abilities. Besides the changes on the kernel configuration, the boot parameters of QEMU are also updated to the specific enterprise Linux distribution, e.g., mounted path of the root file system. As shown in Listing 1, we need to change the QEMU command to boot kernel in VM successfully.

In summary, the kernel configurations related to KVM, drivers for mounting the root file system and network driver have to be enabled firstly. The other configurations referred to kernel hacking are expected to be enabled for the kernel fuzzing, e.g., KASAN. During the configuration adaption process, a powerful machine is helpful, since we need to recompile the kernel for each change on kernel configuration, and boot the updated bzImage in the VM to verify whether the kernel configuration is proper or not. By the iterative adaption of kernel configuration, the kernel finally boots in the VM successfully, and a usable VM instance specific to the enterprise Linux is prepared for syzkaller.

4.3 Syzkaller Fuzzing

After the verification of VM boot, we build syzkaller by the Go compiler and a test run of syzkaller is required. We should ensure the correct and stable operations of kernel fuzzing process. For this purpose, syzkaller is deployed on a test machine and perform kernel fuzzing, to verify whether syzkaller can work correctly and generate the initial corpus. However, several detected bugs interrupt the syzkaller test run and limit the performance of kernel fuzzing. And the practical requirement of continuous fuzzing increases the deployment complexity of syzkaller.

Table 5: Part of the shallow bugs in enterprise Linux kernel found by syzkaller.

ID	Shallow bug descriptions	Count
1	WARNING: kmalloc bug in vga_arb_write	100
2	BUG: unable to handle kernel NULL pointer dereference in sidtab_search_core	100
3	WARNING: kmalloc bug in relay_open_buf	22
4	general protection fault in tcp_sk_exit	12

Challenge 4: Shallow bugs in target enterprise Linux kernel limit the fuzzing of deep path. When we start the syzkaller for several hours, it succeeds to detect some bugs in the target enterprise Linux kernel. However, there are several bugs detected appearing quickly and too frequently, and a part of them as summarized in Table 5. For example, bug 1 and 2 appear more than 100 times when we fuzz Linux-3.10 for the first time, which causes wastes of resource. It is quite common for syzkaller to detect many shallow bugs when performing fuzzing on target enterprise Linux kernel for the first time, which prevents syzkaller to reach the deep path of the kernel effectively.

Solution 4: Patches for shallow bugs repair with developers. The shallow bugs must be fixed or worked around to enable syzkaller to reach deep paths of the kernel before long-time fuzzing. Therefore, the shallow bugs which appear too frequently are fixed

Listing 2: Patch snippnets of shallow bug in Linux-4.19.

```
static void __net_exit tcp_sk_exit(struct net *net
    ){
...
- module_put(net->ipv4.tcp_congestion_control->
    owner);
+ if (net->ipv4.tcp_congestion_control)
+    module_put(net->ipv4.tcp_congestion_control->
    owner);
...
}
```

by the related patches. For example, the bug 4 is fixed with the collaboration of developers in Alibaba Inc., as shown in Listing 2. In particular, this bug is caused by the function of tcp_sk_exit(). If the initialization function tcp_sk_init() failed in inet_ctl_sock_create(), the parameter 'net->ipv4.tcp_congestion_control' will be left uninitialized. However, there is no check for that in the function tcp_sk_exit() when invoking module_put() to remove the usage count. This bug leads to a NULL pointer dereference in the end. Moreover, the bug also exists in the upstream of the latest Linux kernel and is not fixed when we found this bug. We fix this bug by adding a judgment before module_put() to check whether net->ipv4.tcp_congestion_control has been initialized, and thus prevent the NULL pointer dereference [23].

In general, the shallow bugs in the enterprise Linux kernel can be fixed not only by the enterprise developers theirselves, but also can utilize the upstream patches because syzbot or other syzkaller users may find the same bug in the main Linux kernel branches, e.g., the bug 3 is fixed by the patch from the community [20]. As a result, the patches are used to fix shallow bugs in the enterprise Linux kernel, which enable syzkaller to reach the deeper path in the kernel.

Challenge 5: Requirements of continuous fuzzing and its complexity. It is quite common for industrial users to adapt the continuous test framework in their daily developments. For example, Alibaba Inc. has developed a kernel test system (KTS) integrating many open source test suites specific to the Linux kernel, as a part of continuous integration (CI) when a new version of enterprise Linux is released. In this case, we need to integrate kernel fuzzing in the continuous integration. Furthermore, testers also want to deploy syzkaller on any free test machine in the test pool and fuzz the latest built enterprise Linux kernel automatically, which brings more challenges to the deployment of syzkaller due to the environment complexity and frequent enterprise Linux updates. Because of the historical burden, the environment of test machines in the test pool is complex, which is different from the predefined environment of the machine in syzkaller test run. Therefore, the problem of packages missing still happens. Moreover, the continuous fuzzing should follow the updates on the enterprise Linux kernel, e.g., kernel configurations. The enterprise developers usually just maintain a standard enterprise Linux configuration, but no more kernel configuration files specific to fuzzing. This situation is quite common in real industry projects, and reduces the usability and acceptance of fuzzing techniques in practice, because it is tedious to manually update kernel configuration and other required packages.

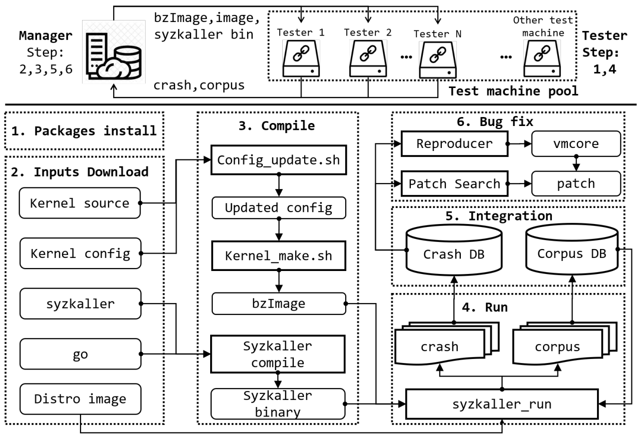

Figure 3: Continuous fuzzing framework.

Listing 3: Scripts snippets to update kernel configurations for continuous fuzzing.

```
bash $KERNEL_PATH/scripts/config\
-e CONFIG_VIRTIO\          # KVM
-e CONFIG_BLK_DEV_SD\      # Device drivers
-e CONFIG_KCOV\            # Coverage collection
-e CONFIG_KALLSYMS\        # Syscalls detection
-e CONFIG_KASAN\           # Bug detection tools
...
make olddefconfig
```

Solution 5: Develop continuous fuzzing framework with automatic scripts for syzkaller deployment. To solve the problems in continuous fuzzing, we design the supporting modules to integrate the syzkaller with the existing continuous testing system, including corpus updates and bug information reporting. As shown in Fig. 3, there are two types of test machines in the proposed framework: one manager and many testers. The manager builds the required components and derives the latest syzkaller binary for kernel fuzzing. It is also responsible for the management of corpus and providing the final bug reports for bug fixing to developers. The testers in the test pool are responsible for performing kernel fuzzing. In this way, the required kernel and packages are firstly installed on the tester, to ensure a usable environment of host OS. Components required by syzkaller are downloaded and compiled by the manager, e.g., the latest version of enterprise Linux kernel source and syzkaller from the code repository. It is noted that the kernel configuration specific to fuzzing is updated based on the standard enterprise kernel configuration in this phase. The script of updating kernel configuration is based on the script to manipulate .config files on the command line [22], as shown in Listing 3.

Then, syzkaller deployed on testers starts kernel fuzzing. When the daily fuzzing is over, the corpus and crashes on each tester are integrated by the manager. After that, the possible patches and discussions related to bugs are searched and provided to the enterprise developers, as presented in the bug fix module. Moreover, the VM instance with the vulnerable kernel for bug reproduction are also prepared for developers to fix bugs. With these modules,

Table 6: Summary of bugs in enterprise Linux distributions.

Kernel	# Found Bugs	# Reproducible Bugs
Linux-3.10	28	6
Linux-4.9	27	7
Linux-4.14-rt	46	18
Linux-4.19	31	10

kernel fuzzing can be efficiently integrated into the daily continuous testing architecture. With the help of the C reproduce code, VM instance, core dump and relevant patches from the Linux mainline branches, enterprise developers can detect and fix the bugs with syzkaller more easily.

5 RESULTS

After overcoming the challenges, we successfully perform fuzzing on enterprise Linux kernel distributions, and integrate syzkaller into existing CI system of our industrial partners. After running kernel fuzzing for several weeks, a number of bugs have been detected in the test version of enterprise Linux kernel, and a part of them can be reproduced based on the C code generated by syzkaller. Table 6 summarizes the number of bugs which are detected and reproduced by syzkaller in each enterprise Linux kernel. The results indicate that syzkaller is able to detect a number of kernel bugs. In particular, we detect 6, 7, 18, 10 reproducible bugs in Linux-3.10, Linux-4.9, Linux-4.14-rt and Linux-4.19, respectively. The number of reproducible bugs show the considerable performance of syzkaller in bug reproduction which can help developers to fix bugs. In general, without considering the cost and risk to overcome the challenges, kernel fuzzing is a powerful approach to find bugs in enterprise Linux kernel.

To show the effectiveness of kernel fuzzing with more details, we conduct a case study of kernel fuzzing on the Linux-4.19. Firstly, we summarize the reproduced bugs of Linux-4.19 found by syzkaller and their status, as shown in Table 7, where "en." and "up." denotes a bug is fixed by enterprise developers and upstream, respectively. There are 10 bugs have been reproduced in Linux-4.19, which are the previous unknown bugs for enterprise developers, and 5 bugs have been fixed by patches of upstream while developers are fixing other bugs. As for the approach of bug fixing, 2 bugs are fixed by the patches from upstream, while 3 bugs are fixed by the patches provided by enterprise developers based on the provided VM instance and the C reproducer for bug reproduce. For the example of "use-after-free Write in sanitize_ptr_alu", we give the trace of fault injection and the patch in Listing 4 and Listing 5, respectively. From the trace, we can observe that, when kzalloc() fails in push_stack(), the current verifier state will be released by free_verifier_state(). When push_stack() returns, dst_reg will be restored if ptr_is_dst_reg is false. However, in this case, dst_reg is also free as a member of cur_state. Therefore, the error of use after free occurs when dst_reg is de-referenced. Based on the provided VM instance with debug information, the enterprise developers fix it by checking the return value of push_stack() before restoring dst_reg. The patches related to bugs in the enterprise Linux kernel are also submitted to the

Table 7: Status of reproduced bugs found by syzkaller in Linux-4.19.

Bug	Status
general protection fault in tcp_sk_exit	en.
divide error in fb_var_to_videomode	fixing
possible deadlock in fifo_open	up.
possible deadlock in lock_trace	up.
possible deadlock in console_unlock	fixing
KASAN: use-after-free write in sanitize_ptr_alu	en.
INFO: task hung in exit_aio	fixing
WARNING in __alloc_pages_nodemask	en.
WARNING in __device_add_disk	fixing
unregister_netdevice: waiting for DEV to become free	fixing

Listing 4: Fault injection trace for KASAN: use-after-free Write in sanitize_ptr_alu.

```
Fault injection trace:
kfree+0xea/0x290
free_func_state+0x4a/0x60
free_verifier_state+0x61/0xe0
push_stack+0x216/0x2f0 <- inject failslab
sanitize_ptr_alu+0x2b1/0x8d0
...
```

Listing 5: Code snippets of kernel/bpf/verifier.c for KASAN: use-after-free write in sanitize_ptr_alu.

```
static int sanitize_ptr_alu(){
...
ret = push_stack(env, env->insn_idx + 1, env->
    insn_idx, true);
- if (!ptr_is_dst_reg)
+ if (!ptr_is_dst_reg && ret)
*dst_reg = tmp;
return !ret ? -EFAULT : 0;
}
```

Linux upstream [39]. In summary, the deployment of kernel fuzzing can truly find the enterprise concerned bugs in industry practice.

Moreover, we compare the types of bugs detected by syzkaller and current test framework, to show the benefits of overcoming those challenges and deploying syzkaller on the current testing framework. In industry practice, most companies maintain a continuous integration system including their test suites. In the case of Linux-4.19, engineers use the kernel test system (KTS) as the continuous testing framework for Linux-4.19. We record the bugs found by fuzzing and KTS on the same daily testing period, and the results are summarized in Table 8. From the results we observed that the bugs detected by fuzzing is different from KTS, that the bugs found by fuzzing are related to the safety and security of

Table 8: Types of bugs found by fuzzing and KTS in Linux-4.19.

Type	Fuzzing	KTS
Soft lockup	8	0
Deadlock	6	0
WARNING	6	0
Task hung	3	0
Rcu detected stall	2	0
Divide error	1	0
General protection fault	1	0
Stack out of bounds	1	0
Use-after-free	1	0
Performance	0	14
Services usability	0	3
Others	2	0
Total	**31**	**17**

the enterprise Linux kernel, e.g., soft lockup, deadlock, hung and stall, while the current kernel testing framework mainly focus on performance and services usability. Therefore, combining kernel fuzzing tools with the existing testing frameworks are expected to find more previously unknown bugs in enterprise Linux kernel, and beneficial to ensure the correctness by integrating syzkaller into the current continuous integration framework.

6 LESSON AND LEARNED

In this section, we introduce some lessons learned on kernel fuzzing deployment, the analysis of detected bugs, and the process of fixing them in industry practice.

More efforts should be made to bridge the gap between the academic kernel fuzzers and the complex enterprise Linux kernel distributions. Most of the kernels used in enterprise Linux distributions are different from the latest Linux kernel. Specifically, enterprise Linux kernels may not satisfy the prerequisites for kernel fuzzing tools, e.g., the absence of kernel coverage support leads to the failure of syzkaller deployment on Linux-3.10. The support for fuzzing and bug detection features in enterprise Linux kernel is also insufficient and highly needed. More specifically, the fuzzing performance degradation appears when the useful kernel features related to bug detection in latest mainline Linux kernel are not all usable in enterprise Linux kernel, e.g., systematic fault injection support. Those features should be back-ported in industry practice.

Fuzzing on mainline kernel maybe not enough, and bugs detected in local enterprise Linux kernel distributions may also exist in the upstream. Though the syzbot system continuously fuzzes main Linux kernel branches, we still found bugs which are previously unknown in both main Linux kernel branches and enterprise Linux kernel, as presented in Fig. 4. For example, there are 10 bugs reproduced in Linux-4.19, but we reproduce all of them

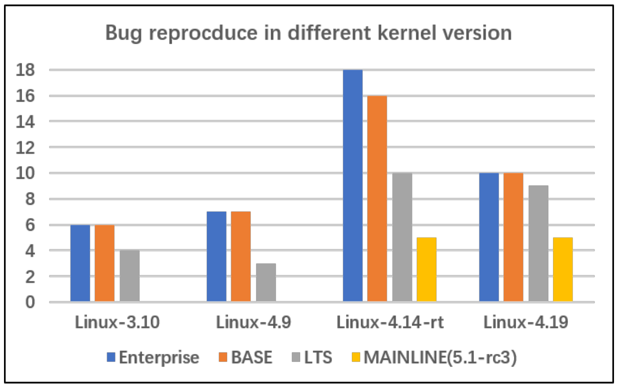

Figure 4: Bug reproduce in different kernel version.

in its base version of Linux-4.19.24, 9 of them in its LTS version of Linux-4.19.y, and 5 of them in the latest mainline branch of Linux-5.1-rc3. The reason is that the kernel configurations of enterprise Linux are different from that used for syzbot, which may trigger different paths during kernel fuzzing. Moreover, we focus on one enterprise Linux kernel specific to the particular version with all the computing power of the testing machine, while syzbot follows the daily development of mainline Linux kernel branches.

Patches from the upstream should be monitored and updated timely to the enterprise Linux kernel. During the experiments, we found there are many bug forks in the enterprise Linux kernel. We found many bugs which have been fixed in main Linux kernel branches but still appear in enterprise Linux kernel. For example, there are 5 bugs reproduced in Linux-4.19 but have been fixed in mainline branch of Linux-5.1-rc3. In this case, each bug fixed in mainline is still a new bug for the enterprise Linux kernel. Though it is not always possible for distributions, e.g., enterprise Linux, to track or fully monitor the commits of the upstream [6], it is expected to fix bugs caused by forking as soon as possible. Moreover, we also found that part of fixed bugs in mainline Linux kernel also exists in the LTS version of Linux. For example, there are 4 bugs reproduced in the LTS version of Linux-4.19.y which have been fixed in mainline branch of Linux-5.1-rc3. The results show that the work to the backport of bug fixes is still not enough, for both enterprise Linux kernel and LTS version.

More bug information should be provided to developers for bug fixing. Enterprise developers prefer fixing the bugs which can be reproduced. In the case of Linux-4.19, all bugs with C reproducer are fixed or fixing by the enterprise developers. However, though there are logs for bug location provided by bug detection tools, e.g., KASAN and KMEMLEAK, it is difficult to locate the buggy code and verify the patch, especially to reproduce bugs in the physical machine. Moreover, there are lots of crashes that are just reported but not reproduced. For these bugs, developers not only need the logs such as call stack, but also more debug information, e.g., vmcore. In practice, developers first evaluate the relevance between crash and kernel and the possibility of bug reproduction in physical machines. If a bug is not caused by kernel, rarely triggered or can only be triggered under an extremely unreasonable condition, engineers will reduce the priority to fix it.

7 CONCLUSION

In this paper, we conduct an industry practice of coverage-guided kernel fuzzing on four enterprise Linux distributions. We identify the main challenges in deploying the kernel fuzzer of syzkaller in the enterprise Linux kernel and develop corresponding solutions to address these challenges. In particular, we backport the kernel module of KCOV to support coverage collection. We perform the adaption on both kernel configurations and QEMU instruction to ensure the correct kernel booting in the VM. Then, we fix the bugs on the shallow paths as well as implementing the continuous fuzzing specific to the enterprise Linux kernel. The experimental results demonstrate the effectiveness of our proposed solutions, which have been integrated in the continuous integration system of our industrial partners.

This paper aims to bring more attention of researchers and developers to bridge the gap between the academic optimizations and the complex industrial environment in kernel fuzzing. Our future work mainly includes the following three aspects: 1) Customize the domain knowledge of different enterprise Linux kernel distributions such as configuration into the fuzzing of mainline kernel to help uncover more application related bugs. 2) Try to produce more debug information automatically to help developers facilitate the localization of kernel bugs. 3) Try to monitor and analyze those patches from the mainline kernels and automatically update those changes into the local enterprise Linux kernel distributions.

REFERENCES

[1] 2015. Syzbot dashboard. https://syzkaller.appspot.com/. Accessed April 26, 2019.

[2] 2016. AFL/QEMU fuzzing with full-system emulation. https://github.com/nccgroup/TriforceAFL. Accessed April 26, 2019.

[3] 2016. kernel-fuzzing: Fuzzers for the Linux kernel. https://github.com/oracle/kernel-fuzzing. Accessed April 26, 2019.

[4] 2016. TriforceLinuxSyscallFuzzer: A linux system call fuzzer using TriforceAFL. https://github.com/nccgroup/TriforceLinuxSyscallFuzzer. Accessed April 26, 2019.

[5] 2017. How to set up syzkaller. https://github.com/google/syzkaller/blob/master/docs/linux/setup.md. Accessed April 26, 2019.

[6] 2017. Syzbot and the tale of thousand kernel bugs. https://events.linuxfoundation.org/wp-content/uploads/2017/11/Syzbot-and-the-Tale-of-Thousand-Kernel-Bugs-Dmitry-Vyukov-Google.pdf. Accessed April 26, 2019.

[7] 2018. Alibaba Linux kernel tree. https://github.com/alibaba/alikernel. Accessed April 26, 2019.

[8] 2018. Linux kernel configs. https://github.com/google/syzkaller/blob/master/docs/linux/kernel_configs.md. Accessed April 26, 2019.

[9] 2019. Alibaba Cloud Linux Kernel - an open-source Linux kernel originated by Alibaba Operating System Team. https://github.com/alibaba/cloud-kernel. Accessed April 26, 2019.

[10] 2019. Alibaba Cloud Linux OS - An open-source Linux distribution powered by Alibaba Cloud. https://alibaba.github.io/cloud-kernel/os.html. Accessed April 26, 2019.

[11] 2019. The CentOS Project. https://www.centos.org. Accessed April 26, 2019.

[12] 2019. Kernel configs used by syzbot. https://github.com/google/syzkaller/blob/master/dashboard/config. Accessed April 26, 2019.

[13] 2019. The Real Time Linux collaborative project. https://wiki.linuxfoundation.org/realtime/start. Accessed April 26, 2019.

[14] Shuai Bai, Dan Li, Minhuan Huang, and Hua Chen. 2017. Synthesis of Linux Kernel Fuzzing Tools Based on Syscall. *DEStech Transactions on Computer Science and Engineering* (2017).

[15] Costin Carabas and Mihai Carabas. 2017. Fuzzing the Linux kernel. *2017 Computing Conference* (2017), 839–843.

[16] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Zhuo Su, and Xun Jiao. 2018. EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers.

[17] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Krügel, and Giovanni Vigna. 2017. DIFUZE: Interface Aware Fuzzing for Kernel Drivers. In *ACM Conference on Computer and Communications Security*.

[18] HyungSeok Han and Sang Kil Cha. 2017. IMF: Inferred Model-based Fuzzer. In *ACM Conference on Computer and Communications Security*.

[19] Dae R. Jeong, Kyung Tae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. 2018. RAZZER : Finding Kernel Race Bugs through Fuzzing.

[20] Dave Jiang. 2018. Kernel/relay.c: limit kmalloc size to KMALLOC_MAX_SIZE. https://lkml.org/lkml/2018/2/6/842. Accessed April 26, 2019.

[21] Kyungtae Kim and Byoungyoung Lee. 2018. ALEXKIDD-FUZZER: Kernel Fuzzing Guided by Symbolic Information. https://www.cerias.purdue.edu/assets/symposium/2018-posters/829-D1B.pdf. Accessed April 26, 2019.

[22] Andi Kleen. 2018. Manipulate options in a .config file from the command line. https://github.com/torvalds/linux/blob/master/scripts/config. Accessed April 26, 2019.

[23] Dust Li. 2019. Tcp: fix potential NULL pointer dereference in tcp_sk_exit. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=b506bc975f60f06e13e74adb35e708a23dc4e87c. Accessed April 26, 2019.

[24] Jun Li, Bodong Zhao, and Chao Zhang. 2018. Fuzzing: a survey. *Cybersecurity* 1 (2018), 6.

[25] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Guang Zhang. 2018. Fuzzing: State of the Art. *IEEE Transactions on Reliability* 67 (2018), 1199–1218.

[26] Jie Liang, Yu Jiang, Yuanliang Chen, Mingzhe Wang, Chijin Zhou, and Jiaguang Sun. 2018. Pafl: extend fuzzing optimizations of single mode to industrial parallel mode. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 809–814.

[27] Jie Liang, Mingzhe Wang, Yuanliang Chen, Yu Jiang, and Renwei Zhang. 2018. Fuzz testing in practice: Obstacles and solutions. *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2018), 562–566.

[28] Valentin J. M. Manès, H. Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2018. Fuzzing: Art, Science, and Engineering. *CoRR* abs/1812.00140 (2018).

[29] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *USENIX Security Symposium*.

[30] Borislav Petkov. 2013. X86, platform, kvm, kconfig: Turn existing .config's into KVM-capable configs. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=46ff53874bd935ab9955dee56d60212857e89bf3. Accessed April 26, 2019.

[31] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *USENIX Security Symposium*.

[32] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. 2018. Charm: Facilitating Dynamic Analysis of Device Drivers of Mobile Systems. In *USENIX Security Symposium*.

[33] Dmitry Vyukov. 2015. Syzkaller: an unsupervised, coverage-guided kernel fuzzer. https://github.com/google/syzkaller. Accessed April 26, 2019.

[34] Dmitry Vyukov. 2016. Documentation: note that KCOV is supported since gcc 4.5. https://lkml.org/lkml/2016/12/13/373. Accessed April 26, 2019.

[35] Dmitry Vyukov. 2019. kcov: code coverage for fuzzing. https://www.kernel.org/doc/html/latest/dev-tools/kcov.html. Accessed April 26, 2019.

[36] Mingzhe Wang, Jie Liang, Yuanliang Chen, Yu Jiang, Xun Jiao, Han Liu, Xibin Zhao, and Jiaguang Sun. 2018. SAFL: increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 61–64.

[37] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. 2019. Fuzzing File Systems via Two-Dimensional Input Space Exploration. In *IEEE Symposium on Security and Privacy (SP)*.

[38] Wei You, Peiyuan Zong, Kai Chen, Xiaofeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. 2017. SemFuzz: Semantics-based Automatic Generation of Proof-of-Concept Exploits. In *ACM Conference on Computer and Communications Security*.

[39] Xu Yu. 2019. Bpf: do not restore dst_reg when cur_state is freed. https://lkml.org/lkml/2019/3/21/202. Accessed April 26, 2019.