Automated Bug Detection and Replay for COTS Linux Kernel Modules with Concolic Execution

Bo Chen*, Zhenkun Yang*, Li Lei†, Kai Cong†, Fei Xie†
*Intel Corporation, Hillsboro, OR 97124, USA, {chenbo, zhenkun.yang}@intel.com
†Portland State University, Portland, OR 97201, USA, {leil, congkai, xie}@cs.pdx.edu

Abstract—**Linux kernel is pervasive in the cloud, on mobile platforms, and on supercomputers. To support these diverse computing environments, the Linux kernel provides extensibility and modularity through Loadable Kernel Modules (LKM), while featuring a monolithic architecture for execution efficiency. This architecture design brings a major challenge to the security of Linux kernel. Having LKMs run in the same memory space as the base kernel on Ring 0, a single flaw from LKMs may compromise the entire system, e.g., gaining root access. However, validation and debugging of LKMs are inherently challenging, because of its special interface buried deeply in the kernel, and non-determinism from interrupts. Also, LKMs are shipped by various vendors and the public may not have access to their source code, making the validation even harder.**

In this paper, we propose a framework for efficient bug detection and replay of commercial off-the-shelf (COTS) Linux kernel modules based on concolic execution. Our framework automatically generates compact sets of test cases for COTS LKMs, proactively checks for common kernel bugs, and allows to reproduce reported bugs repeatedly with actionable test cases. We evaluate our approach on over 20 LKMs covering major modules from the network and sound subsystems of Linux kernel. The results show that our approach can effectively detect various kernel bugs, and reports 5 new vulnerabilities including an unknown flaw that allows non-privileged users to trigger a kernel panic. By leveraging the replay capability of our framework, we patched all the reported bugs in the Linux kernel upstream, including 3 patches that were selected to the stable release of Linux kernel and back-ported to numerous production kernel versions. We also compare our prototype with kAFL, the state-of-the-art kernel fuzzer, and demonstrate the effectiveness of concolic execution over fuzzing on the kernel level.

I. Introduction

Linux kernel is widely used, e.g., 90 percent of the public cloud workloads were running on Linux in 2017 [1]; in the first quarter of 2019, 75 percent of smart-phones were equipped with Android which uses Linux as its core [2]; all of the top 500 supercomputers use Linux at the end of 2018 [3]. To support these diverse computing environments, the size of the Linux kernel has been steadily growing, reaching over 24.7 million LOC [1], and is continually changing to improve security, performance or maintainability, as well as to support new devices, file systems, and hardware architectures [4].

Linux kernel is typically split into two parts, e.g., the base kernel and Loadable Kernel Modules (LKM) [5]. The base kernel provides essential services for user applications and LKMs, such as process management, memory management, and inter-process communication. Other functionalities are offloaded into separate LKMs, such as supporting a new device or file system. The use of LKMs significantly improves the extensibility and modularity of Linux kernel and reduces the memory usage of Linux kernel, by allowing dynamic loading and unloading of LKMs on demand. The security and reliability of LKMs are critical to the entire computer system, as they are part of the trusted computing base of many systems [6]. Bugs and vulnerabilities in LKMs can easily lead to system crashes, and some can be further exploited by adversaries with normal privilege to bypass kernel-enforced protections and gain root privilege eventually. A study by Arnold et al. [7] argues that every kernel bug should be treated as security-critical, and must be patched as soon as possible. As a result, systematic and thorough validation and testing for LKMs are highly desired.

Nevertheless, LKM validation (both functional and security) and debugging are inherently difficult. First, LKMs are buried deeply inside the Linux kernel, interacting only with hardware and base kernel directly. Isolating LKMs for runtime validation is difficult and labor intensive. Testing LKMs through the kernel interface, e.g., system calls, is also not effective, as different inputs issued to the kernel interface need to cross multiple layers or modules to reach target LKM interfaces. Second, Linux kernel employs a number of kernel threads, intensively interacting with hardware and user-level applications, leading to high concurrency and non-determinism. It remains a challenge to efficiently reproduce discovered kernel bugs. Furthermore, LKMs are shipped by various vendors which may not have access to their source code, and interactions between multiple LKMs are even harder to validate.

There have been many recent approaches to verifying and testing the Linux kernel and LKMs [8]–[21]. Static analysis is widely used [8]–[11], yet it faces major challenges, such as high false positive rates, not capable to detect runtime defects, and not applicable to COTS LKMs. Symbolic or concolic execution has been applied to Linux kernel and drivers [12]–[14]. However, they either need to instrument and recompile the kernel [13], or do not produce actionable test cases [12] which are essential for reproducing and debugging detected kernel bugs. Recently, fuzzing has been trending in detecting security vulnerabilities in OS kernels [15]–[21]. Most of the work [18]–[20] focuses on fuzzing through system call interfaces of Linux kernel which is often far away the target LKMs and cannot effectively analyze target LKM behaviors. Many of other work is not applicable to COTS LKMs [16], [17]. In summary, existing approaches have two major limitations: (1) lack of

978-1-7281-5143-4/20/$31.00 © 2020 IEEE

effective analysis over COTS LKMs by manipulating LKM interfaces directly; (2) lack of infrastructures to generate and replay test cases that can steadily reproduce detected kernel vulnerabilities, under the kernel non-determinism.

We present a novel approach to thoroughly testing COTS LKMs and steadily reproducing discovered bugs. Our approach includes two major techniques: (1) automated test case generation from LKM interfaces with concolic execution; (2) automated test case replay that repeatedly reproduces detected bugs. Our approach starts with a concrete execution of target LKMs triggered by a test harness that is a sequence of user-level application commands. Along this concrete execution, we inject symbolic values to the LKM interface and perform concolic execution to exercise different paths of target LKMs and generate test cases for each explored path. A generated test case is a sequence of LKM interface invocations that contains inputs or outputs values of LKM entry functions and kernel APIs invoked from target LKM. To minimize the non-determinism of the sequence of LKM interface invocations under the same test harness, for test case generation and replay, we exclude LKM interface invocations if the kernel is handling interrupts, and only include LKM invocations triggered by non-concurrent user-level commands from the test harness. Together with the capability of detecting and tolerating inconsistencies of LKM invocations while test case replay, we achieve high replayable rate of generated test cases, and enable automated reproduction of detected bugs.

We have implemented a prototype of our approach in COD, based on an open-source concolic engine CRETE [22]. Together with kernel dynamic instrumentation via Kprobe [23], COD automatically generates compact sets of test cases from COTS LKMs, proactively checks for common kernel bugs with embedded checkers, and provides facility to repeatedly replay detected vulnerabilities with actionable test cases. We have evaluated COD on over 20 LKMs which cover major modules from the network and sound subsystems of Linux kernel. The results show that our approach can effectively identify various kinds of kernel bugs, and reports 5 previously unreported vulnerabilities including an unknown flaw that allows non-privileged user to trigger kernel panic. By leveraging COD's test case replay capability, we were able to fix all the detected flaws in a short time without any domain knowledge, and patched all these bugs in the Linux kernel upstream, including 3 patches were selected to the stable release of Linux kernel and back-ported to numerous production kernel versions.

In summary, our paper makes three key contributions:

- We proposed an approach to automatically generating compact sets of test cases for COTS LKMs. The generated test cases can thoroughly exercise the target LKMs by manipulating LKM interfaces directly and precisely.
- We designed a system to automatically replay test cases of COTS LKMs and proactively check for common kernel bugs, which allows to repeatedly reproduce detected kernel vulnerabilities. We believe this system has major potential in helping LKM debugging and patching.

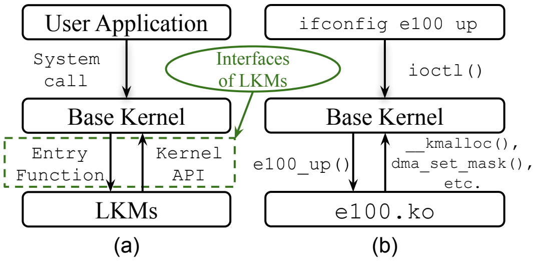

Fig. 1. The interface of LKMs: (a) interactions between user application, base kernel and LKMs, and (b) a concrete example based on `ifconfig` and `e1000`. The interactions between hardware and LKMs are omitted.

- We implemented a prototype of our approach in COD, and evaluated it with over 20 COTS LKMs covering network and sound subsystems of Linux kernel. COD discovered various kernel vulnerabilities, including null-pointer de-reference and resource leak. By leveraging the replay facility of COD, we also patched all the detected bugs in the Linux kernel upstream.

II. Background

A. *Interfaces of LKMs*

A program communicates and interacts with users or other programs through interfaces. With different interface inputs, a program exhibits different behaviors and exercises different paths. The purpose of test case generation is to produce a set of interface inputs that covers as many program paths as possible. User applications normally have clean interfaces, e.g., strings for command-line programs, and files for editors.

LKMs have a more complex interface than user applications, because they are buried in kernel and only works with the base kernel directly. As shown in the green box of Figure 1 (a), LKMs interacts with base kernel through entry functions and kernel APIs, which are the LKM interfaces. Entry functions are defined in LKMs and are exposed to base kernel as interfaces to fulfill requests from user applications, while LKMs utilize kernel functionalities by calling kernel APIs. Different paths of LKMs can be exercised with either different entry function calls with different arguments from base kernel, or different side effects from kernel APIs, e.g., return values of kernel APIs, and data exchanged with pointer arguments passed to kernel APIs. For example, Figure 1 (b) shows the interactions between LKM `e100.ko` and the base kernel, triggered by the user application `ifconfig`. Note that LKMs also interact directly with hardware, e.g., reading and writing hardware interface registers. It is also a part of the LKM interface, but is omitted as it is not the focus of this paper.

B. *Concolic Execution and the CRETE Engine*

Concolic Execution [24], which combines concrete and symbolic execution, has become an important technique for automated software analysis [25], [26]. It has advantages over concrete execution [27] since it explores each execution path based on path constraints, while it is more scalable than symbolic execution [28] because it leverages information from concrete execution to augment symbolic execution.

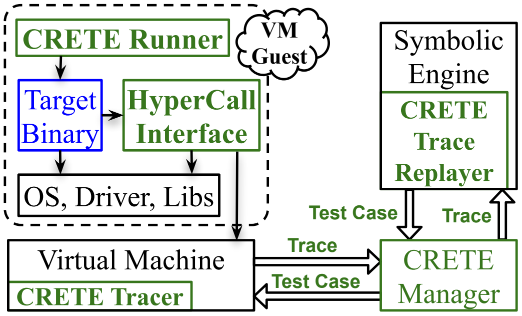

Fig. 2. The architecture of CRETE.

We adopt the CRETE engine [22] in this paper, as it is open-source, works on unmodified x86 binaries, and features a modular design for easy extension. As shown in Figure 2, CRETE consists of three components. (1) Its front-ends are Virtual Machines (VMs), namely QEMU [29], augmented with tracing capabilities, the CRETE `Tracer`, which produce replayable traces for the target binary running inside the VM guest OS. To scale the engine to real word applications, CRETE employs a Dynamic Taint Analysis (DTA) [30] engine in the `Tracer` for reducing the size of captured runtime traces. Its DTA engine tracks the propagation of tainted values, normally specified by users, during the execution of a program, and only captures basic blocks that operate on tainted values [22]. (2) Its back-end is a symbolic execution engine, namely KLEE [28], augmented with the CRETE `Trace Replayer`, which takes captured traces as inputs and generates test cases to cover new paths of the target software. (3) Its manager is a coordinator that maintains pools of test cases and traces, and communicates with both front-end and back-end through sockets to exchange test cases and traces.

C. Kernel Dynamic Instrumentation

We leverage Kprobe [23], a debugging mechanism provided by Linux kernel, to perform kernel dynamic instrumentation. Kprobe allows users to insert a set of handlers on a certain instruction address. By using Kprobe, we introduce concolic values at the interface of LKMs for test case generation, replay generated test cases repeatedly, and collects run-time information for detecting kernel bugs.

III. Approach

A. Methodology

While designing the COD framework for analyzing LKMs, we have identified the following design goals:

- **Binary-level In-vivo Analysis.** It should be applicable to COTS LKMs, and require no recompilation or modification to the rest of kernel stack.
- **Effective Bug Detection.** It should detect various types of kernel bugs with minimal false alarms.
- **Automated Bug Replay.** It should enable developers to reproduce bugs easily, which helps locate and fix the reported bugs.
- **Multiple LKMs.** It should be capable of analyzing multiple LKMs and their interactions at the same time.

To achieve the goals above, we adopt and extend the versatile concolic testing approach of CRETE in the design of the COD framework as follows.

- We introduce a `kernel shim` to intercept interactions between base kernel and target LKMs, and use it along with a `kernel hypercall interface` to dynamically inject concolic values at LKM interfaces while capturing runtime traces (Section IV-C). Also, we build `COD tracer` by augmenting CRETE tracer to support multiple applications and kernel modules, through which we capture run-time execution traces of target LKMs from unmodified guest OS stack (Section IV-D).
- We build `COD Trace Replayer` for symbolic analysis and test case generation over the captured traces, by extending the CRETE trace replayer with `trace checkers` and `constraint editors` for checking common kernel bugs and imposing constrains on generated test cases (Section IV-E).
- We provide `COD TC Replayer`, which allow users to replay generated test cases repeatedly, out of test generation environment and on both virtual and physical platforms (Section V). It is embedded with `kAPI checkers` (Section V-B) to detect common kernel bugs and produce informative reports to boost bug analysis.

B. Test Cases for LKMs

The core of our approach is to generate effective test cases from LKM interfaces for bug detection and replay. We now introduce the definition of the test case for LKMs.

Let an LKM entry function or a kernel API be function $f : \vec{\alpha} \to \tau$, where $\vec{\alpha}$ represents the inputs and τ represents the return of function f. An invocation of function f is denoted by a triple $k \triangleq \langle f_i, \vec{A}, t \rangle$, where f_i is one instance of the invocation of function f, $\vec{A}$ contains the concrete values for the inputs, and t is the concrete return value. Note that we treat the invocations of f at different locations in the LKM as different instances. A test case $\pi \triangleq (k_0, k_1, \cdots, k_n)$ is a sequence of entry function or kernel API invocations.

Informally, when we run a test harness with multiple user commands, it triggers a sequence of LKM entry functions or kernel APIs. A test case is defined as the observed behavior on the interfaces, *i.e.* inputs and return values of LKM entry functions and kernel APIs, upon running the test harness. COD distinguishes each instance of function invocations with a `TC Identifier` that consist of function name, invocation site, LKM name, and index of user commands in the test harness.

C. Handling Kernel Non-determinism

Our approach involves tracing the execution of LKMs and replay of LKM test cases, both of which face major challenges from kernel's non-determinism. The major cause of kernel's non-determinism is from interrupts and the concurrent nature of the kernel itself [31].

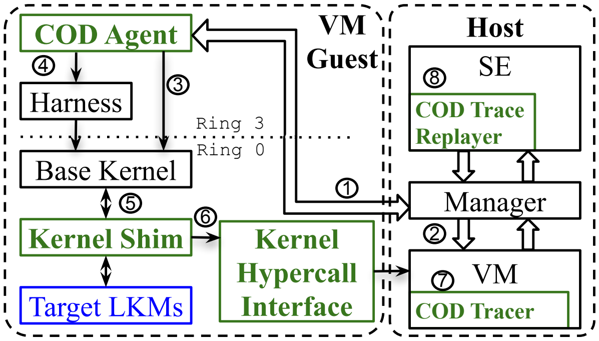

Fig. 3. The architecture of COD for automated test case generation.

To handle this challenge, we first monitor the start and end of interrupts and exclude the execution of interrupt handler from the tracing and test case generation process (details in Section IV-D). Second, we require the user-level test harness only contains commands with no concurrency, and all commands are executed sequentially. Meanwhile, in test case generation and replay, we only include LKM interface invocations from the test harness, while excluding other invocations from other process or interrupts. Third, in test case replay with the given test harness, we detect and tolerate the inconsistencies of interface invocations from target LKMs (details in Section V).

IV. Test Case Generation

A. Architecture and Workflow

As shown in figure 3, the COD architecture for test case generation is split into two domains, VM guest OS and host OS. A user-land `Agent` and two custom kernel modules, `kernel shim` and `kernel hypercall interface`, together with target LKMs and native OS stack are running within VM guest OS. A virtual machine augmented with `COD Tracer`, a symbolic engine augmented with `COD Trace Replayer`, and a Manager are running on host machine.

We now outline the events and communications that take place during the test case generation process. When the manager is started, (1) it sends a message to `Agent` through sockets, and (2) sends an initial test case to the VM. The message contains a list of target LKMs, and a sequence of commands as test harness. (3) The `Agent` loads two custom kernel modules, `kernel shim` and `kernel hypercall interface`, and pass them the list of LKMs as parameters. (4) The `Agent` then executes the commands of the test harness sequentially to trigger functionalities of target LKMs through base kernel. (5) The custom kernel module `kernel shim` intercepts the interactions between base kernel and target LKMs. (6) It also communicates with the VM through the other module `kernel hypercall interface`, to add new tainted values to the taint analysis engine in the VM, report kernel panics to the VM, and retrieve values of test case from VM to modify the interactions between target LKMs and base kernel if needed. (7) When all commands in the test harness are finished, the `COD Tracer` captures the runtime execution trace into a file, and sends it to symbolic engine through the manager over sockets. (8) The `COD Trace Replayer` performs symbolic analysis over the captured trace, and sends the generated test cases back to the VM. The iteration of test case generation repeats from step (4) to step (8), and stops when user specified conditions are met, e.g., time limits.

B. COD Agent

The `Agent` is a user-mode application running in the VM guest OS, which receives commands from the Manager and sets up the guest OS for test generation. The `Agent` inserts the two custom kernel modules of COD along with the list of the target LKMs as parameters, and launches the commands from test harness one by one. It also monitors the crash and time-out of executed commands, and reports to VM when needed. The `Agent` also passes user-level information to the kernel through system calls, including the PID and index of the running command.

C. Kernel Shim and Hypercall Interface

COD provides two custom kernel modules running in Ring 0 to inject concolic values at the interface of target LKMs for capturing runtime traces. The module `kernel shim` defines a set of Kprobe handlers to intercept interactions between base kernel and LKMs, including calls to the entry functions of target LKMs from the base kernel and calls to kernel APIs from target LKMs. It also takes as input a list of target LKM names, and maintains the user-level information sent from the `Agent`, including the PID and index of the running command from the test harness. Based the PID and list of names, the Kprobe handlers modify the interactions between base kernel and target LKMs only if they are triggered by the command from the test harness. Each Kprobe handler defines where to inject concolic values to the current LKM entry function or kernel API. The module `hypercall interface` defines interface functions for VM guest kernel to communicate with the underlying VM. An important interface function is `cod_make_concolic()`, which is used to inform the VM to inject concolic values to the VM guest memory. This function takes as inputs a `TC Identifier`, the address and size of the piece of kernel memory in the VM guest for injecting concolic values. The `TC Identifier` is generated by each Kprobe handlers based on the information passed from the `Agent`. With the call to this function, the underline VM first marks the given range of guest memory as tainted values which is used for taint analysis and selective tracing, and then tries to retrieve values from a given test case by matching the `TC Identifier`. When a match is found, the values from test case overwrite the values in the given range of guest memory, modifying the current LKM interface invocation. Another important hypercall interface function is `cod_kernel_oops`, which reports the kernel panic to the VM for logging detected issues and rapid restarting for the next iteration of runtime tracing. For example, with these two custom kernel modules, COD introduces concolic values (and finally generates test cases) to the kernel memory whose value is from copy_from_user(), selected arguments of e1000_ioctl(), and return value of __kmalloc().

D. COD Tracer

The Tracer produces runtime traces of COTS LKMs. Like traces of CRETE, the captured trace is a self-contained LLVM [32] module with injected custom callbacks to inject symbolic values for test generation in Trace Replayer. We extend the Tracer of CRETE to capture COTS LKM runtime traces. CRETE is designed for user-level binaries and is not applicable to LKMs. First, CRETE is limited in injecting concolic values to the interface of user-level applications, e.g., command-line, file and stdin which are all statically known before the execution of given applications. COD extends the Tracer to support concolic values from LKM interfaces that are dynamically added on-the-fly during the execution of given test harness. Second, CRETE is designed to analyze a single application. COD extends the Tracer to capture traces from a sequence of applications. The COD Tracer turns on capturing when the PID is sent from the Agent, and turns off capturing when the process of the given PID exits. Also, COD extends the DTA engine in the Tracer to track the propagation of tainted values in the kernel across different processes, instead of only tracking a single process. As the design of the split virtual memory layout used in common x86 OS, Linux kernel is mapped to virtual address space of all processes, and is located always at the same virtual address. We pass the tainted memory in the virtual address of the kernel in the previous target process to the coming target process, and use it as the initial tainted values to start taint analysis. Third, COD added an interrupt monitor to the Tracer. COD intercepts the procedure of CPU transition from normal execution to interrupt handler in the VM, which covers both the synchronous interrupts raised from software, e.g., interrupt from a page fault, and the asynchronous interrupts raised from hardware, e.g., interrupt from network card. At the same time, the COD Tracer interleaves the procedure of handling iret instruction in the VM to detect the end of interrupts. By maintaining a call stack structure of interrupt starts and ends, the Tracer handles nested interrupts. When the CPU is executing code of interrupt handlers, the Tracer turns off tracing and ignores hypercalls of cod_make_concolic(). This alleviates the nondeterminism of the kernel for tracing and test case generation.

E. COD Trace Replayer

The Trace Replayer introduces symbolic values to the captured trace based on the callbacks embedded in the trace, replays the trace symbolically, and generates test cases by negating constraints of the branches encountered. We extend the Trace Replayer of CRETE with a Constraint Editor and a Trace Checker to generate more compact set of test cases, and detect more bugs with lower false alarm rates for LKMs. The Constraint Editor defines a set of rules to add predefined constraints to the symbolic values while they are introduced to the captured trace. These rules are to refine the symbolic values related to the kernel APIs and impose valid constraints to the test case generated. For example, the symbolic value of pci_enable_device's return is restricted to be in range $[-128, 0]$, which respects that this function returns 0 on success, returns negative on failure and never return positive values; the symbolic value of __kmalloc's return is restricted to be 0 (null), which respects that memory allocation functions either return 0 on failure or return non-zero on success. These rules are crucial to produce compact sets of test cases and reduce false alarms from generating test cases with invalid values of kernel APIs. What's more, the KAPI Checkers define a set of custom assertions to proactively check common bugs of LKMs. If an assertion failed, a test case is generated for users to reproduce the same assertion failure later, and the bug is reported to the Manager for logging. One example assertion is "__kmalloc() != 0". During the symbolic replay of captured traces, this assertion is checked for every memory operation (both read and write) whose operand address is composed of the return from __kmalloc().

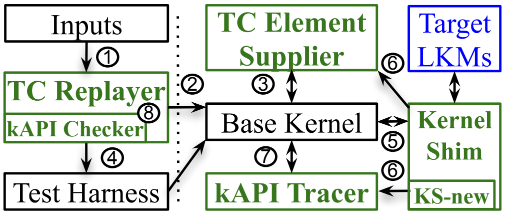

Fig. 4. The architecture of COD for automated test case replay.

V. Test Case Replay

A. Architecture and Workflow

COD allows user to reproduce generated test cases repeatedly on both physical and virtual machines, and generates crash log to assist developers to debug and fix reported bugs. As shown in Figure 4, the architecture of test case replay in COD is composed of a user-mode program TC Replayer with an extensible plugin kAPI Checker, and three custom kernel modules, namely Kernel Shim, TC Element Supplier, and kAPI Tracer.

We now illustrate the workflow of this design. (1) The TC Replayer is started by users with inputs of a set of test cases and a configuration file. The configuration file contains a list of target LKMs, and a sequence of commands as the test harness. Then the TC Replayer (2) loads the three custom kernel modules and passes them the list of target LKMs as parameters, (3) picks one test case and pass it to the custom kernel module TC Element Supplier, and (4) executes the commands in the test harness sequentially to trigger functionalities of target LKMs. (5) The custom kernel module Kernel Shim intercepts the interactions between base kernel and target LKMs. (6) The callbacks in Kernel Shim either call into TC Element Supplier to modify the interactions between kernel and target LKMs, or call into kAPI Tracer to capture kernel API usage information. When all commands in the test harness are executed, the TC Replayer (7) retrieves the kernel API usage information from the custom kernel module kAPI Tracer, and (8) checks for potential bugs with kAPI Checker. The loop repeats from (3) to (8) for all input test cases.

B. COD TC Replayer and kAPI Checker

The TC Replayer is a user-mode application that takes user inputs, and manages the test case replay loop. It leveraging Kdump [33] to collect system log and kernel dump image when kernel fails, such as kernel panic, oops or hang. TC Replayer also automatically retries on the same kernel failure, and reports to users only kernel failures that can be consistently reproduced. At the end, it outputs a set of detected bugs along with the corresponding test cases, system logs, and kernel dump images. Additionally, like the COD Agent, the TC Replayer passes user-level information, including PID and index of the running command from the test harness, to the kernel to assist the test case replay. The TC Replayer is also embedded with kAPI Checker which contains a set of assertions to check common bugs related to kernel API usages, e.g., detecting resource leak with paired function [34].

C. Custom Kernel Modules

The custom kernel module Kernel Shim is reused from the COD's design of test case generation, and is extended with KS-new which contains additional set of Kprobes on kernel API functions. With the Kprobes, function invocations of target LKMs are intercepted. TC Element Supplier provides a set of interface functions with the same signature as its counterpart in test case generation Kernel Hypercall Interface which is used by Kprobe handlers. But it is used to replay test case instead of communicating with VM to inject concolic values. In TC Element Supplier, the interface function cod_make_concolic() still takes as inputs a TC Identifier, the address and size of a piece of kernel memory. With the input TC Identifier, this function checks whether the current LKM function invocation matches the one from the test case under replay. If matched, the current invocation is modified with the corresponding values of function inputs or outputs from the test case, replaying the matched invocation from the test case. Otherwise, a mismatch of the test case replay is detected, indicating non-determinism occurs, which stops the replay of the test case for the current running command and resumes on the next command. The custom kernel module kAPI Tracer is used by the Kprobes defined in KS-new. It captures runtime information of the probed kernel APIs, including kernel API name, input values, return values, target LKM name and call site information that is the offset from the .text section of target LKM.

D. Measurement of Test Case Replay

In general, a test harness triggers a sequence of LKM function invocations that are intercepted by Kernel Shim. This in turn triggers a sequence of calls to function cod_make_concolic() in TC Element Supplier, and generates a sequence of TC Identifier representing the sequence of LKM function invocations from the current execution of the test harness. We measure the replayable rate of a test case by measuring the similarity score of the new TC Identifier sequence and TC Identifier sequence from the test case under replay. We define the similarity score of two sequences as follows.

Definition 1: **Sequence Similarity Score:** Let p and q be two sequences, and let $\text{LCP}(p, q)$ be the longest common prefix of p and q. The similarity score of p and q is defined as $\xi(p, q) = \frac{|\text{LPC}(p,q)|}{\max(|p|,|q|)}$.

For example, if $p =$ abcd, and $q =$ abcex, the longest common prefix of p and q is abc, so the similarity score of $\xi(p, q)$ is $60\%(3/5)$.

Since the new TC Identifier sequence, denoted as π_1, is triggered by x commands in the test harness, we can further partition π_1 into sub-sequences based on which command triggered which sub-sequence, denoted as

$$\pi_1 \triangleq ((\overbrace{k_0^0, \cdots, k_m^0}^{c_0^1}), (\overbrace{k_0^1, \cdots, k_n^1}^{c_1^1}), \cdots, (\overbrace{k_0^x, \cdots, k_p^x}^{c_x^1})),$$

or $\pi_1 \triangleq (c_0^1, c_1^1, c_x^1)$ in short. Similarly, the TC Identifier sequence from the test case under replay is denoted as $\pi_2 \triangleq (c_0^2, c_1^2, c_x^2)$. Since π_1 and π_2 are sequences of sub-sequences, we measure the similarity of π_1 and π_2 as $\xi(\pi_1, \pi_2) = \frac{\sum_{i=0}^{x} |\text{LCP}(c_i^1, c_i^2)|}{\sum_{i=0}^{x} \max(|c_i^1|, |c_i^2|)}$.

VI. Implementation

We built a prototype of COD based on CRETE [22]. We extended its front-end, the VM QEMU [29], with 1.1k LOC for supporting concolic interface of LKMs, tracing multiple processes, and monitoring interrupts. We also extended CRETE back-end, the symbolic engine KLEE [28], with 0.7k LOC code for supporting Constraint Editor and Trace Checker. We wrote a set of custom kernel modules based on Linux kernel v3.13 (default kernel for Ubuntu 14.04) and v4.4 (default kernel for Ubuntu 16.04). There are roughly 2.2k LOC, which defines 154 Kprobes [23] for test case generation, and another over 78 Kprobes for tracing kernel API usage during test case replay. We also wrote a set of checkers for 113 pairs of kernel API detecting resource leak bugs, and a checker for detecting redundant usage of netif_napi_del, which is a common problem in network drivers we learned from existing patches. The TC Replayer has 1.5k LOC code. It supports replaying a batch of test cases, collects bug reports (e.g., kernel dump image and system log), and resumes replay from kernel panics automatically by leveraging Kdump [33]. We also defined interfaces for users to easily add new Kprobes and checkers in the format of C macros.

VII. Evaluation

In this section, we present the evaluation result of COD. First, we present the evaluation results of bug detection of COD in Section VII-A. It includes all new vulnerabilities that were found by COD, and evaluation of COD's ability to

TABLE I
List of LKMs evaluated by COD

Subsystem	Main LKM	Dependent LKMs
Network	e100	mii
	e1000	-
	pcnet32	mii
	ne2k-pci	8390
	8139too(cp)	mii
	tg3	ptp, pps_core
Sound	snd_intel8x0	snd-ac97-codec, ac97_bus, snd-pcm, snd-timer, snd, soundcore
	snd_hda_intel	snd_hda_codec_generic,snd_hda_codec, snd_hda_core, snd_hwdep,snd_pcm, snd_timer, snd, soundcore
	snd_ens1370	snd_rawmidi, snd_seq_devicesnd_pcm, snd_timer, snd, soundcore

TABLE II
List of Test Harnesses

Idx.	Test Harness	Target

1	`insmod` *xxx*
	`ifconfig ens4 up`
	`dhclient ens4`
	`ip route add` *xxx* `dev ens4`
	`ethtool` *xxx* `ens4` *xxx*
	`ping -I ens4 -c 1 -W 1` *xxx*
	`curl --interface ens4` *xxx*
	`rmmod` *xxx*
		LKMs from network subsys.

2	`insmod` *xxx* `speaker-test -l 1` `rmmod` *xxx*	LKMs from sound subsys.

find known vulnerabilities. Second, we measure the replayable rate of test cases generated by COD on both virtual and physical platforms in Section VII-B. Third, based on the patches we submitted to Linux kernel upstream, we elaborate on how to leverage COD's capability of test case replay to locate and fix Linux kernel bugs efficiently in Section VII-C. Finally, we present the comparison of bug detection capability with kAFL, the state-of-the-art fuzzing engine capable of testing unmodified Linux kernels. If not stated otherwise, the evaluations were performed on a desktop system with an Intel i7-4770 processor @ 3.40GHz and 16GB DDR3 RAM @ 1600MHz running 64-bit Ubuntu 14.04.6 operating system.

A. Bug Detection

To highlight the effectiveness of our engine, we applied COD to LKMs that are widely used and validated both in industry and academia. Table I shows the list of LKMs we evaluated with COD, and Table II shows the test harnesses we used in our experiments. All the main LKMs have been released at least 14 years [35]. They are also being actively maintained by the Linux kernel community and large vendors, such as RedHat, SUSE, Broadcom, and Intel. This is because those LKMs are providing important functionality to modern computer systems, such as Ethernet device drivers, network middleware, HDA codec, and core sound module, etc. For the same reason, many of the LKMs, e.g., `E1000`, `PCNet32`, `8139too` and `snd_ens1370`, have been studied and used as benchmarks for evaluation by numerous previous research prototypes [13], [15], [16].

TABLE III
New Linux Kernel Vulnerabilities Detected by COD

Index	LKM	Bug Description	Patch hash
1	E1000	Resource Leak	`ee400a3`
2	E1000	Null-pointer dereference	`cf1acec`
3	Pcnet32	Resource leak	`d7db318`
4	8139too(cp)	Kernel API misuse	`a456757`
5	hda_intel	Null-pointer dereference	`a3aa60d`

We applied COD for test generation with a time-out of 24 hours on each main LKM along with their dependent LKMs as listed in Table I. By replaying all generated test cases with COD on both virtual and physical machines, COD reported a total of 5 new distinct vulnerabilities from 4 different kernel module. As shown in Table III, COD detected various kinds of vulnerabilities, including null-pointer dereference, resource leak, and kernel API misuse. All the bugs were reported to the Linux kernel community, and were patched immediately.

We now take `Bug 1` as an example to explain why COD is able to generate test cases from COTS LKMs to trigger and report the new flaws in Table III. `Bug 1` is detected by `TC Replayer` during the replay of COD generated test cases, where `kAPI checker` reported a piece of memory allocated by function `__kmalloc` is not paired with any memory deallocation function. By examining test cases triggering this bug, we found COD only flipped a single kernel API return from the initial test case. COD can explicitly flip these single API returns because there are conditional branches in the target LKM depending on the flipped API returns. By leveraging concolic execution, COD was able to negate these branch conditions precisely, generate a compact set of test cases to explore new code in the LKM and finally catch the bug with `TC Replayer` and `kAPI checker`. For the similar reason, COD flipped more kernel APIs, generated LKM test cases with the right kernel API combination to reach error paths, and finally reported these vulnerabilities with `TC Replayer`.

We also evaluated COD's ability to find previous known vulnerabilities. We chose an older version of the Linux kernel `v3.13`, which was released on Jan 2014 and was the default kernel for Long-term-support release of Ubuntu 14.04. We selected two LKMs that COD did not report new issues as target main LKMs, namely `E100` and `NE2K-PCI`. By running COD to generate test cases for 24 hours on each LKM set, 3 bugs were detected, including 1 null-pointer dereference, 1 resource leak from `E100`, and 1 resource leak from `ne2k-pci`. To the best of our knowledge, by manually browsing the patches of the target LKMs since version `v3.13`, we believe COD covered all known vulnerabilities of target LKMs related to null-pointer deference and resource leak.

B. Test Case Replay

In this section, we measure the replayable rates (Section V-D) of test cases generated by COD. We selected two examples from Table I, namely `e1000` and `snd-hda-intel` along with their dependent LKMs, and used the test harness as shown in Table II. We also evaluated each example with test

TABLE IV
AVERAGE REPLAYABLE RATE OF TEST CASE REPLAY

Test Harness	Complete Harness		'insmod/rmmod' Only	
Platforms	VM	PM	VM	PM
snd-intel-hda	100%	100%	100%	100%
E1000	95.31%	96.76%	100%	100%

harnesses that only contains 'insmod/rmmod', considering LKM complexity stems mostly from initialization and cleanup code [36]. We performed evaluations on both virtual platform, using QEMU v2.3 in KVM mode, and physical platform, using a desktop system with Intel Pentium processor @ 3.2GHz with 1GB RAM. For each set of LKMs, we first run its test harness once to record a test case, and then replay the test case with the same test harness to measure the replayable rate.

Table IV shows the average replayable rate for running each set of LKMs repeatedly for 1000 times with different test harness. The results show that the replayable rate is 100% for snd-intel-hda with all test harness and e1000 with test harness of 'insmod/rmmod' only, on both virtual and physical platforms. Also, inconsistencies are observed from the experiments on e1000 with complete test harness. This is mainly caused by non-determinism from communications with remote machines when executing commands ping and curl. An interesting observation is that the replayable rate from physical machine is higher than virtual machine for e1000. We believe the reason is that the network vitalization of virtual machine depends on the host machine's network which has a more dynamic environment and brings extra non-determinism. We also want to point out that, with the test harness of 'insmod/rmmod' only, the replayable rate is always 100% for both sets of LKMs evaluated on both virtual and physical platforms, which shows the potential of COD in testing various LKMs' initialization and cleanup code.

C. Bug Patching Example

By leveraging COD's capability of automated test case replay, we were able to reproduce, debug, analyze and finally fix all the detected bugs listed in Table III. We are all new to network and sound subsystem of Linux kernel, especially have no previous knowledge and experience of the specific LKMs that COD reported new bugs. Despite of that, we fixed all the bugs efficiently. The total time to fix each bug was ranging from 1 hour to 3 hours. We submitted all our patches to the Linux kernel upstream, and all patches were accepted immediately by the subsystem maintainers. Especially, three of our patches were selected and merged to the stable tree of Linux kernel. These patches are back-ported to various long-term-support release of Linux OS, e.g., Ubuntu 16.04/18.04 and Debian 8/9, and are now running on numerous machines.

We now elaborate on a few examples of how COD assisted us to patch the reported vulnerabilities. For Bug 1 in table III, we took three steps to fix it. Figure 5 shows the code excerpt related to this bug.

```
 1  // Allocation site
 2  int e1000_alloc_queues
 3  (struct e1000_adapter *adapter, ...) {
 4    ...; adapter->tx_ring = kcalloc(...);
 5  }
 6  // Branch Negated by COD
 7  int e1000_setup_rx_resources
 8  (struct e1000_adapter *adapter, ...) {
 9    ...;  rxdr->buffer_info = vzalloc(size);
10    if (!rxdr->buffer_info) return -ENOMEM;
11    ...;  return 0;
12  }
13  // Deallocation site
14  int e1000_set_ringparam(...) {
15    ...; tx_old = adapter->tx_ring; ...;
16    if (netif_running(adapter->netdev))
17    {...; kfree(tx_old); ...;}; ...; }
```

Fig. 5. Excerpt of LKM E1000 related to Bug 1 from Table III.

Step 1: locate the 'allocation site'. We started with the bug report produced by kAPI checker in TC Replayer. The report looks like the following:

```
[Resource leak]
address: 0xf1717ac0,
alloc site: __kmalloc @ 0x47aa (e1000),
command in harness: 'ifconfig ens4 up',
```

which means that the kernel memory with virtual address 0xf1717ac0 was allocated by function __kmalloc from offset 0x47aa of LKM E1000 during the execution of command 'ifconfig ens4 up', and it was never paired with a corresponding memory de-allocation function. The linux-image-dbgsym package [37] on Ubuntu allows mapping an offset from stripped LKMs to their source code. In this way, we located the allocation site that was in function e1000_alloc_queues as shown in line 4 of Figure 5.

Step 2: find the 'de-allocation site'. The major challenge to fix a resource leak bug is to find the right place in the code to de-allocate the leaked memory. Fixing this bug is practically more challenging considering no prior knowledge and the size of the module E1000 (over 11k lines of code). COD also contributed to tackle this problem by providing not only the test case π' that can be used to reproduce the current bug, but also providing a reference test case π from which the test case π' is generated from. Root-causing the issue would be easier by cross-referencing the execution of π and π'. The test case π' is generated by concolic execution engine from π by flipping the branch condition b along exercising the test case π. In this example, COD flipped the return value of the kernel API in line 9 of Figure 5 and generated test case π'. By replaying test case π with COD and checking its kAPI trace captured by kAPI Tracer, we located the same 'allocation site' and its pair de-allocation function. We now have a reference 'de-allocation site' that is from function e1000_set_ringparam as show in line 17 of Figure 5.

Step 3: analyze the reason of resource leak and write a patch. To understand why test case π' triggered the resource leak while its reference test case π did not, we checked the code near the negated branch in function e1000_setup_rx_resources. This function returns on success with test case π, and returns on error with test case π'. It is invoked during the execution of command `'ifconfig ens4 up'` as part of the initialization of `E1000` network interface. Returning on error of this function leads to the failure of the initialization and notifies the base kernel that the current network interface is not up. By checking the 'free site', we noticed the de-allocation function is guarded by a condition that is true only when the network interface is running, as shown in line 16 of Figure 5. Finally, we fixed the bug by moving the de-allocation function out of the condition check. The whole process of debugging and fixing the bug took us less than one hour by using COD.

Kernel vulnerabilities involving pointer operations, e.g., the null-pointer dereference in `Bug 5` in table III, are among the most common and critical bugs, while is notoriously difficult to debug and fix. As elaborated in Charm [38], the debugging process usually starts from the crash site to backtrack the usage of vulnerable pointer by using GDB (breakpoint, watch-point, single-step, etc.). With the help of COD, we not only quickly pinpointed the crash site, but also easily located the source of the null-pointer dereference that is the location of negated branch. Starting from the source of the bug and tracing down, we were able to fix the `Bug 5` within three hours.

D. Comparison with kAFL

To have an apple-to-apple comparison with kAFL [19], we adopted the example in their evaluation for comparing with other state-of-the-art kernel fuzzers. It is a custom kernel module that is a JSON parser, decodes user inputs, and contains a known vulnerability. They demonstrated that kAFL was able to learn correct JSON syntax, and finally trigger the known vulnerability in around 8 minutes, while other fuzzers failed. The vulnerability was triggered by matching string `"kAFL"` byte by byte. To further challenge both kAFL and COD, we modified the crashing condition which now computes a hash value [39] from multiple JSON tokens of the parsed input string and matches the hash value with the hash value of `"LKM"`(see Figure 6).

As kAFL requires special CPU features, e.g., *Intel VT-x* and *Inte-PT*, all experiments are conducted on a system with an Intel 3.76GHz Xeon E-2176G processor and 32GB RAM. We run kAFL in single process mode, as the multi-process mode of kAFL does not show much efficiency benefit and sometimes is even less efficient as showed in their evaluations. We performed 3 repeated experiments for both kAFL and COD with a timeout of 24 hours. We measured the time and the number of test cases being exercised to find the known crash.

In the experiments, on average, COD triggered the crash within 16 hours after exercising 52K test cases, while kAFL failed to detect the crash within 24 hours and exercised 5500 times more test cases (around 290M) than COD. Actually, kAFL stops finding new paths after about 100 minutes of running. This indicates that fuzzing, even with advanced coverage feedback algorithm and speed boost from newest hardware features, is not capable of finding vulnerabilities that requires complicated and precise conditions. It aligns with the discussion of kAFl's limitations in their paper, which concluded that it remains an open research problem how to deal with these situations on the kernel level [19]. Our work on COD is attacking this research problem by extending concolic execution to kernel modules. The experiment results show that concolic execution is still a very effective technique to detect kernel vulnerabilities on binary-level, and is a strong complimentary testing approach to fuzzing on kernel level.

```
// Robert Sedgewick Hash function
unsigned int RSHash(const char* str,
                    unsigned int len);

jsmn_parser p;
jsmntok_t tokens[5];
jsmn_init(&p);

int res = jsmn_parse(&p, input, len, tokens, 5);
if(res != 3 &&
   tokens[0].type == JSMN_PRIMITIVE &&
   tokens[1].type == JSMN_PRIMITIVE &&
   tokens[2].type != JSMN_PRIMITIVE )  return;

char arr[3] = {input[tokens[0].start],
               input[tokens[1].start],
               input[tokens[2].start]};
if(RSHash(arr, 3) == RSHash("LKM", 3))
  panic(KERN_INFO "LKM...\n");
```

Fig. 6. Excerpt of JSON parser kernel module used for comparing with kAFL.

VIII. Related Works

A. Kernel Vulnerability Detection

Static analysis [8] on Linux kernel source code is very popular, because it normally has no requirement for hardware devices, can be applied to a broad range of the kernel code, and is promising to deliver verification (at least on a specific property of the kernel). Its effectiveness has been demonstrated by many recent tools, e.g., LDV [9], WHOOP [40] Dr. Checker [10], DSAC [11], DEADLINE [41], DCNS [42]. However, static analysis is facing major challenges, including (1) prone to false positives because of the pointer-heavy nature of kernel code, (2) ill-suited for detecting run-time errors involving multiple modules, (3) requiring access to source code, and (4) not easily usable by general developers on new kernel modules [43].

In recent years, dynamic analysis over Linux kernel has received increasingly attention. Especially, feedback-driven fuzzing is proved to be a very effective technique to unveil various vulnerabilities in systems software. Many recent efforts were spent on extending fuzzing on user-level applications (especially AFL [27]) to kernel level, e.g., TriforceAFL [18], kAFL [19], syzkaller [20], DIFUZE [17], and Razzer [21]. Robustness testing of Linux kernel with fault injection is also an effective technique. Two examples are ADFI [15] and EH-Test [16]. Many of these tools either still heavily rely on the source code availability [16], [17] that is not applicable to COTS kernel and LKMs, or perform blindly fuzzing and explore the same execution path repeatedly [15], [18]–[20]

that is inefficient to detect bugs. Also, most of these tools are limited to fuzzing the system call interface of Linux kernel and do not support LKM interfaces.

B. Symbolic and Concolic Execution

Symbolic execution [44] is a program analysis technique that takes symbolic inputs, maintains different execution states and constraints of each path in a program, and utilizes scheduling heuristics [45] to effectively explore the execution tree of the target program. Concolic execution [24] leverages a concrete execution path to guide symbolic execution to achieve better scalability [46]. Both of them have been largely adopted for automated test case generation and bug detection of software on both source and binary level [28], [47]–[55]. Some representative work of applying symbolic or concolic execution to kernel code are DDT [12], SymDrive [13], and CAB-Fuzz [14]. They heavily rely on source-level instrumentation to perform effective dynamic analysis [13], or do not produce actionable test cases [12], [14] that are crucial for efficient replay and debugging on detected problems [56].

C. Kernel Bug Patching and Mining

Our work is also related to automated kernel patching [4]. The Coccinelle project [57] allows software developers to write code manipulation rules with a generalization of the patch syntax [58], and have automatically generated over $6,000$ commits to the Linux kernel. Instead of generating patches to fix kernel bugs automatically, our work tries to improve the process of kernel bug patching by generating actionable test cases for COTS LKMs and enabling automated replay of detected bugs. Additionally, The assertions we defined in COD's `kAPI Checker` (for proactively detecting common kernel bugs) were inspired by previous works on repository mining of Linux kernel [59]–[61], e.g., detecting resource leak with paired function [34].

IX. Discussion

We have demonstrated COD can generate compact sets of test cases from COTS LKM interfaces, detect various kinds of kernel vulnerabilities, and enable automated test case replay to assist efficient debugging and patching of detected bugs. However, there are limitations of this approach and directions for future work, which we will discuss in this section.

Hardware inputs to LKMs. Our approach focuses on the software interactions within the Linux kernel, and does not analyze the effects from hardware inputs [62] to the LKMs. As a result, COD cannot detect bugs of LKMs related to hardware inputs. Also, without a symbolic model to emulate missing hardware modules in the VM (specifically QEMU [29]), COD cannot effectively analyze LKMs requiring unsupported hardware in the VM. Symbolic device [13] is a potential solution to support hardware inputs, and can be incorporated into COD.

Bottleneck of concolic execution. As a concolic testing approach, COD's performance for test case generation is bounded by theoretical limits such as state explosion and expensive constraint solving. We believe fuzzing provide an effective complimentary to concolic testing. We are planning to swap CRETE, the concolic engine in COD's prototype, with state-of-art fuzzers, e.g., kAFL, to perform fuzzing on LKM interfaces [63].

Manual efforts. While COD is mostly automated, developers' manual efforts are still needed in three situations. First, as Linux kernel chose not to adopt a stable interface for LKMs [64], users need to pay attention to the changes of kernel APIs when applying COD to a new version of kernel, and may need to adjust the Kprobes defined in COD. Second, users need to double check all reported bugs because COD can have false alarms. False alarms mainly stem from invalid kernel API models or kernel non-determinism, as kernel API keeps changing, and COD alleviates and tolerates non-determinism, but not removing it. Third, users' manual efforts are required to extend COD to detect new category of bugs, e.g., adding assertions on new kernel API usages, or defining properties about concurrency to detect race conditions, etc. We believe repository mining over Linux kernel [59] can be leveraged to automate the process of tracking kernel API change across different versions and extracting the valid constraint of kernel APIs.

Improvement on the prototype. COD now only supports x86 architecture, but we would like to explore its potential on analyzing COTS LKMs from embedded systems. Also, the current workflow of COD for test case generation does not exploit the potential of multiprocessing and parallelism, and exchanges data through slow socket communication across different components. We are planning to optimize the workflow of COD to improve efficiency.

X. Conclusion

In this paper, we presented COD, an automated testing framework for COTS LKMs in Linux kernel. COD generates compact sets of test cases from LKM interfaces using concolic execution, proactively checks for common kernel bugs with embedded checkers, and allows to reproduce reported bugs repeatedly with actionable test cases on both virtual and physical platforms. We evaluated our prototype of COD on more than 20 LKMs covering the network and sound subsystems of Linux kernel. The experiments showed that COD detected various kinds of kernel bugs, including 5 new vulnerabilities from LKMs that have been maintained and validated for over 14 years. Through patching all the detected flaws in the Linux kernel upstream, we demonstrated the potential of COD's automated test case replay in assisting efficient debugging and fixing of kernel bugs. With the comparison between COD and kAFL, the state-of-the-art kernel fuzzer, we showed that concolic execution remains as an effective complementary testing technique to fuzzing on the kernel level.

Acknowledgment. This research received financial support in part from National Science Foundation (Grant #1908571).

References

[1] J. Corbet and G. Kroah-Hartman, "2017 linux kernel development report," *A Publication of The Linux Foundation*, 2017.

[2] StatCounter, "Mobile operating system market share worldwide," http://gs.statcounter.com/os-market-share/mobile/worldwide, 2019.

[3] B. Lunduke, "Linux and supercomputers," https://www.linuxjournal.com/content/linux-and-supercomputers, 2018.

[4] J. Lawall and G. Muller, "Coccinelle: 10 years of automated evolution in the linux kernel," in *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018.*, 2018, pp. 601–614.

[5] P. J. Salzman, M. Burian, and O. Pomerantz, "The linux kernel module programming guide," http://www.tldp.org/LDP/lkmpg/2.6/html/, 2007.

[6] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek, "Linux kernel vulnerabilities: state-of-the-art defenses and open problems," in *APSys '11 Asia Pacific Workshop on Systems, Shanghai, China, July 11-12, 2011*, 2011, p. 5.

[7] J. Arnold, T. Abbott, W. Daher, G. Price, N. Elhage, G. Thomas, and A. Kaseorg, "Security impact ratings considered harmful," in *Proceedings of HotOS'09: 12th Workshop on Hot Topics in Operating Systems, May 18-20, 2009, Monte Verità, Switzerland*, 2009.

[8] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: Using static analysis to find bugs in the real world," *Commun. ACM*, vol. 53, no. 2, pp. 66–75, Feb. 2010.

[9] I. S. Zakharov, M. U. Mandrykin, V. S. Mutilin, E. M. Novikov, A. K. Petrenko, and A. V. Khoroshilov, "Configurable toolset for static verification of operating systems kernel modules," *Program. Comput. Softw.*, vol. 41, no. 1, pp. 49–64, Jan. 2015.

[10] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, "DR. CHECKER: A soundy analysis for linux kernel drivers," in *USENIX Security Symposium*. USENIX Association, 2017, pp. 1007–1024.

[11] J.-J. Bai, Y.-P. Wang, J. Lawall, and S.-M. Hu, "DSAC: Effective static analysis of sleep-in-atomic-context bugs in kernel modules," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, 2018.

[12] V. Kuznetsov, V. Chipounov, and G. Candea, "Testing closed-source binary device drivers with ddt," in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX-ATC'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 12–12.

[13] M. J. Renzelmann, A. Kadav, and M. M. Swift, "Symdrive: Testing drivers without devices," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 279–292.

[14] S. Y. Kim, S. Lee, I. Yun, W. Xu, B. Lee, Y. Yun, and T. Kim, "Cab-fuzz: Practical concolic testing techniques for COTS operating systems," in *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017.*, 2017, pp. 689–701.

[15] K. Cong, L. Lei, Z. Yang, and F. Xie, "Automatic fault injection for driver robustness testing," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 361–372.

[16] J.-J. Bai, Y.-P. Wang, J. Yin, and S.-M. Hu, "Testing error handling code in device drivers using characteristic fault injection," in *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '16. Berkeley, CA, USA: USENIX Association, 2016, pp. 635–647.

[17] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "Difuze: Interface aware fuzzing for kernel drivers," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: ACM, 2017, pp. 2123–2138.

[18] J. Hertz and T. Newsham, "The triforceafl project," https://github.com/nccgroup/TriforceAFL.git, 2017.

[19] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kafl: Hardware-assisted feedback fuzzing for OS kernels," in *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.*, 2017, pp. 167–182.

[20] Google, "syzkaller - kernel fuzzer," https://github.com/google/syzkaller, 2019.

[21] D. R. Jeong, K. Kim, B. A. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing," in *IEEE Symposium on Security and Privacy, SP 2019, San Francisco, California, USA*, 2019.

[22] B. Chen, C. Havlicek, Z. Yang, K. Cong, R. Kannavara, and F. Xie, "CRETE: A versatile binary-level concolic testing framework," in *Proceedings of the Fundamental Approaches to Software Engineering, 21st International Conference, FASE 2018, Thessaloniki, Greece*, 2018.

[23] R. Krishnakumar, "Kernel korner: Kprobes-a kernel debugger," *Linux J.*, vol. 2005, no. 133, pp. 11–, May 2005.

[24] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, 2005, pp. 263–272.

[25] R. Kannavara, C. J. Havlicek, B. Chen, M. R. Tuttle, K. Cong, S. Ray, and F. Xie, "Challenges and opportunities with concolic testing," in *2015 National Aerospace and Electronics Conference (NAECON)*, June 2015, pp. 374–378.

[26] H. Xu, Y. Zhou, Y. Kang, and M. R. Lyu, "Concolic execution on small-size binaries: Challenges and empirical study," in *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017, Denver, CO, USA, June 26-29, 2017*, 2017, pp. 181–188.

[27] AFL, "American fuzzy lop," http://lcamtuf.coredump.cx/afl/.

[28] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08, 2008, pp. 209–224.

[29] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference, Anaheim, CA, USA*, 2005.

[30] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, 2010, pp. 317–331.

[31] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen, "Execution replay of multiprocessor virtual machines," in *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '08. ACM, 2008, pp. 121–130.

[32] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, 2004, pp. 75–88.

[33] V. Goyal, E. W. Biederman, and H. Nellitheertha, "Kdump, a kexec-based kernel crash dumping mechanism," in *Proc. of the Linux Symposium*, 2005.

[34] H. Liu, Y. Wang, J. Bai, and S. Hu, "PF-Miner: A practical paired functions mining method for android kernel in error paths," *Journal of Systems and Software*, vol. 121, pp. 234–246, 2016.

[35] L. Torvalds, "Initial commit of linux kernel's git repository," https://git.io/fjGug, 2005.

[36] A. Kadav and M. M. Swift, "Understanding modern device drivers," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, 2012, pp. 87–98.

[37] S. Arnold, "Debug symbol packages," https://wiki.ubuntu.com/Debug%20Symbol%20Packages, 2018.

[38] 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 *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.*, 2018, pp. 291–307.

[39] R. Sedgewick, *Algorithms in C*. Wiley, 1990.

[40] P. Deligiannis, A. F. Donaldson, and Z. Rakamaric, "Fast and precise symbolic analysis of concurrency bugs in device drivers," in *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, 2015, pp. 166–177.

[41] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim, "Precise and scalable detection of double-fetch bugs in OS kernels," in *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, 2018, pp. 661–678.

[42] J. Bai, J. Lawall, W. Tan, and S. Hu, "DCNS: automated detection of conservative non-sleep defects in the linux kernel," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, 2019, pp. 287–299.

[43] P. J. Guo and D. Engler, "Linux kernel developer responses to static analysis bug reports," in *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, ser. USENIX'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 22–22.

[44] 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. 50:1–50:39, 2018.

[45] S. Cha, S. Hong, J. Lee, and H. Oh, "Automatically generating search heuristics for concolic testing," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, 2018, pp. 1244–1254.

[46] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[47] P. Godefroid, M. Y. Levin, and D. A. Molnar, "SAGE: whitebox fuzzing for security testing," *Commun. ACM*, vol. 55, no. 3, pp. 40–44, 2012.

[48] V. Chipounov, V. Kuznetsov, and G. Candea, "The S2E platform: Design, implementation, and applications," *ACM Trans. Comput. Syst.*, vol. 30, no. 1, pp. 2:1–2:49, 2012.

[49] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, 2012, pp. 380–394.

[50] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," in *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, 2014, pp. 1083–1094.

[51] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016.

[52] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM : A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.*, 2018, pp. 745–761.

[53] X. Wang, J. Sun, Z. Chen, P. Zhang, J. Wang, and Y. Lin, "Towards optimal concolic testing," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, 2018, pp. 291–302.

[54] Y. Sun, M. Wu, W. Ruan, X. Huang, M. Kwiatkowska, and D. Kroening, "Concolic testing for deep neural networks," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, 2018, pp. 109–119.

[55] B. Chen, K. Cong, Z. Yang, Q. Wang, J. Wang, L. Lei, and F. Xie, "End-to-end concolic testing for hardware/software co-validation," in *15th IEEE International Conference on Embedded Software and Systems, ICESS 2019, Las Vegas, NV, USA, June 2-3, 2019*, 2019, pp. 1–8.

[56] D. Devecsery, "Enabling program analysis through deterministic replay and optimistic hybrid analysis," Ph.D. dissertation, The University of Michigan, 2018.

[57] LIP6, "The coccinelle project," http://coccinelle.lip6.fr/.

[58] Y. Padioleau, J. L. Lawall, R. R. Hansen, and G. Muller, "Documenting and automating collateral evolutions in linux device drivers," in *Proceedings of the 2008 EuroSys Conference, Glasgow, Scotland, UK, April 1-4, 2008*, 2008, pp. 247–260.

[59] J. Lawall, D. Palinski, L. Gnirke, and G. Muller, "Fast and precise retrieval of forward and back porting information for linux device drivers," in *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017.*, 2017, pp. 15–26.

[60] Y. Tian, J. L. Lawall, and D. Lo, "Identifying linux bug fixing patches," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, 2012, pp. 386–396.

[61] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. L. Traon, "Fixminer: Mining relevant fix patterns for automated program repair," *CoRR*, vol. abs/1810.01791, 2018.

[62] L. Lei, K. Cong, Z. Yang, B. Chen, and F. Xie, "Hardware/Software Co-monitoring," in *CoRR, arXiv:1905.03915 [cs.SE]*, 2019.

[63] B. Chen, "Versatile binary-level concolic testing," Ph.D. dissertation, Portland State University, 2019.

[64] G. Kroah-Hartman, "Stable api nonsense," https://www.kernel.org/doc/Documentation/process/stable-api-nonsense.rst.