Portland State University

PDXScholar

Dissertations and Theses

Dissertations and Theses

Summer 7-16-2019

Versatile Binary-level Concolic Testing

Bo Chen

Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds

Part of the [Computer Sciences Commons](#)

Let us know how access to this document benefits you.

Recommended Citation

Chen, Bo, "Versatile Binary-level Concolic Testing" (2019). *Dissertations and Theses*. Paper 5115.

<https://doi.org/10.15760/etd.6994>

This Dissertation is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.

Versatile Binary-level Concolic Testing

by

Bo Chen

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Computer Science

Dissertation Committee:

Fei Xie, Chair

Kai Cong

Feng Liu

Suresh Singh

Xiaoyu Song

Portland State University
2019

© 2019 Bo Chen

Abstract

Computing systems are experiencing an explosive growth, both in complexities and diversities, ushered in by the proliferation of cloud computing, mobile computing, and Internet of Things. This growth has also exposed the consequences of unsafe, insecure, and unreliable computing systems. These all point to the great needs of sophisticated system validation techniques. Recent advances in research on symbolic execution has shown great promises for automated software analysis, e.g., generating test cases, finding bugs, and detecting security vulnerabilities. However, symbolic execution is mostly adopted to analyze user applications, while modern computing systems in practice consist of many components shipped by various vendors, besides user applications, e.g., operating systems, firmware and hardware devices. In this dissertation, we propose versatile binary-level concolic testing, which defines a standard execution-trace format, and features an open and highly extensible architecture. It allows easy integration of multiple concrete execution frontends and symbolic execution backends, which significantly improves the applicability and flexibility of symbolic execution, especially to modern computing systems with various components.

First, we present the design and implementation of CRETE, the infrastructure of versatile binary-level concolic testing. CRETE provides an open and highly extensible architecture allowing easy integration of multiple concrete and symbolic execution environments, which communicate with each other only by exchanging standardized

traces and test cases. We also present several optimizations for scaling CRETE to practical user applications. Our experiments show CRETE outperformed state-of-the-art open-source systems for automated program analysis at source-level and binary-level. It also found numerous bugs that were previously unreported from mature open-source projects.

Second, we present COD, a framework based on versatile binary-level concolic testing for automated bug detection and replay of commercial off-the-shelf (COTS) Linux kernel modules (LKMs). 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. Our experiments show that COD can effectively detect various kernel bugs, and reports 5 new kernel vulnerabilities including an unknown flaw that allows non-privileged users to trigger a kernel panic. With the replay capability of our framework, we patched all the reported 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.

Last, we present how we leverage versatile binary-level concolic testing for system-level validation of Systems-on-Chips (SoC). We capture run-time traces of Hardware/-Software (HW/SW) components across the entire SoC stack which are emulated by multiple virtual platforms. Based on segmented traces captured from various SoC components, we assemble system-level traces and provide interfaces for users to inject system-level assertions to validate. The experimental results demonstrate that our approach can generate effective system-level test cases crosscutting the entire HW/SW stack of SoC and triggering an IP firmware bug from user inputs of an IP utility program, and can catch various bugs with system-level assertions.

Dedication

To Mom, Dad, Brother, and Weiling

Acknowledgements

First of all, I would like to thank my advisor Dr. Fei Xie. Without his continuous support and encouragement, I would not have a steady environment to focus on my research and this dissertation would not have been accomplished. Fei kept challenging me to be an independent researcher, a committed collaborator, and a good mentor. He also opened many opportunities for me to learn from others. His passion about solving real world problems and building practical systems has great influence on my Ph.D. research and future career.

I would like to thank Dr. Kai Cong, Dr. Feng Liu, Dr. Suresh Singh, and Dr. Xiaoyu Song for serving on my dissertation committee, for their valuable feedback on the dissertation draft and the insightful questions they asked during my defense.

I would like to thank my shepherds, Dr. Kai Cong, Dr. Zhenkun Yang, and Dr. Li Lei. Kai was a great mentor and helped me build initial traction for starting my Ph.D. research. Zhenkun hosted my internship at Intel Labs and taught me how to conduct collaborative research in industrial settings. Li always provided constructive advice to inspire me to think out-of-the-box. They are also great friends to me. Their help and encouragement made my Ph.D. study a joyful journey.

I would like to thank my co-authors: Christopher Havlicek, Dr. Raghudeep Kannavara, Dr. Qin Wang, Dr. Jialu Wang, Dr. Mark Tuttle, and Dr. Sandip Ray. I acknowledge my colleagues at PSU: Bin Lin, Simon Nicklaus, Hao Wu, Shu-ping

Chu, Yu Li, Yubin Fan, Zhe Li, Haifeng Gu, Yanzhao Wang, and Qiqi Hou. I am especially grateful to Rebecca Sexton-Lee, Beth Phelps, Kristine-Anne Sarreal, Evan Punongbayan and all other staff from the Department of Computer Science at PSU, for taking care of all administrative issues arising throughout my Ph.D. study.

Finally, I would like to thank my parents and brother for their consistent support and unconditional love. I was so fortunate to be spoiled by them, allowing me to explore, fail, and succeed. Special thanks to Weiling Sun for her endless love and patience. She witnessed all the major up and down of my Ph.D. study, and was the source of my perseverance when things get tough. I was so lucky to meet and know her, which I would never dare to ask for again. Together, we made it.

Table of Contents

Abstract	i
Dedication	iii
Acknowledgments	iv
List of Tables	x
List of Figures	xi
1 Introduction	1
1.1 Overview	1
1.2 Infrastructure of Versatile Binary-level Concolic Testing	2
1.2.1 Problem Statement	2
1.2.2 Solution Overview	3
1.3 Automated Bug Detection and Replay for COTS Linux Kernel Modules	6
1.3.1 Problem Statement	6
1.3.2 Solution Overview	8
1.4 Hardware/Software Co-validation of Systems-on-Chips	9
1.4.1 Problem Statement	9
1.4.2 Solution Overview	10
1.5 Dissertation Outline	11
2 Background	12
2.1 Symbolic Execution and the KLEE Engine	12
2.2 Concolic Testing	13
2.3 The QEMU Platform	14

3	The Infrastructure of Versatile Binary-level Concolic Testing	16
3.1	Overview	17
3.2	Design	19
3.2.1	CRETE Architecture	19
3.2.2	Standardized Runtime Trace	21
3.2.3	Selective Binary-level Tracing	24
3.2.4	Concolic Test Case Generation	25
3.2.5	Bug and Runtime Vulnerability Detection	26
3.3	Implementation	27
3.3.1	CRETE Runner	28
3.3.2	CRETE Tracer for QEMU	29
3.3.3	CRETE Manager	29
3.3.4	CRETE Replayer for KLEE	30
3.3.5	CRETE Tracer for 8051 Emulator	30
3.4	Experimental Results	31
3.4.1	GNU COREUTILS	31
3.4.2	TianoCore Utilities	35
3.5	Related Work	39
3.5.1	DART, CUTE, and SAGE	39
3.5.2	KLEE and S2E	39
3.5.3	BitBlaze, Mayhem, and MergePoint	40
3.5.4	Angr	41
3.5.5	Fuzzing Testing	41
3.6	Discussions	42
3.6.1	Parallelization for Scalability	42
3.6.2	Extensibility and Applicability	42
3.7	Summary	43
4	Automated Bug Detection and Replay for COTS Linux Kernel Modules	44
4.1	Background	45
4.1.1	Interfaces of LKMs	46
4.1.2	Kernel Dynamic Instrumentation	47
4.2	Overview	47

4.2.1	Methodology	47
4.2.2	Test Cases for LKMs	49
4.2.3	Handling Kernel Non-determinism	49
4.3	Test Case Generation	50
4.3.1	Architecture and Workflow	50
4.3.2	COD Agent	51
4.3.3	Kernel Shim and Hypercall Interface	52
4.3.4	COD Tracer	53
4.3.5	COD Trace Replayer	54
4.4	Test Case Replay	55
4.4.1	Architecture and Workflow	55
4.4.2	COD TC Replayer and kAPI Checker	56
4.4.3	Custom Kernel Modules	57
4.4.4	Measurement of Test Case Replay	57
4.5	Implementation	58
4.6	Experimental Results	59
4.6.1	Bug Detection	60
4.6.2	Test Case Replay	63
4.6.3	Bug Patching Example	64
4.6.4	Comparison with kAFL	67
4.7	Related Work	69
4.7.1	Kernel Vulnerability Detection	69
4.7.2	Symbolic and Concolic Execution	70
4.7.3	Kernel Bug Patching and Mining	70
4.8	Discussion	71
4.9	Summary	72
5	Hardware/Software Co-validation of Systems-on-Chips	74
5.1	Background	75
5.1.1	SoC Stack over Virtual Platforms	75
5.1.2	Virtual Devices in QEMU	76
5.2	Preliminary Definitions	78
5.3	Design	80

5.3.1	End-to-End Run-time Tracing	81
5.3.2	System-level Trace Assembling	81
5.3.3	Instrumentation Interface for System-level Analysis	83
5.4	Implementation	84
5.4.1	Full-stack Emulation of SoC	84
5.4.2	Tracers for QEMU, QEMU VD, and Emu8051	86
5.4.3	System-level Trace Assembler	87
5.4.4	Interface for System-level Analysis	88
5.4.5	Customized Concolic/symbolic Engine	89
5.5	Experimental Results	89
5.6	Related Work	95
5.7	Discussion	96
5.7.1	Limitations	96
5.7.2	Future Work	97
5.8	Summary	97
6	Conclusion	99
6.1	Summary of Contributions	99
6.2	Future Directions	100
	Bibliography	102

List of Tables

3.1	Comparison of Average and Median Coverage by KLEE, ANGR, and CRETE on COREUTILS	32
3.2	Distribution Comparison of Coverage Achieved by KLEE, ANGR, and CRETE on COREUTILS	33
3.3	Classified Crashes Found by CRETTE on Tianocore Utilities: 84 unique crashes from 8 programs.....	37
4.1	List of LKMs Evaluated by COD	60
4.2	List of Test Harnesses	61
4.3	New Linux Kernel Vulnerabilities Detected by COD	62
4.4	Average Replayable Rate of Test Case Replay	63
5.1	Target SoC SW Stack	90
5.2	The SoC Stimulus	90
5.3	Sizes of SoC Traces Captured with Each Stimulus	91
5.4	Runtime of End-to-end Tracing and Assembling	91
5.5	System-level Properties Validated by User-defined Assertions.....	92
5.6	Number of Generated Test-cases and Triggered Assertions from Concolic- symbolic Hybrid Execution	94

List of Figures

1.1	The overview of dissertation.....	2
2.1	A simple function <code>bad_abs</code> in C with its symbolic execution tree: (a) Function <code>bad_abs</code> in C. (b) Symbolic execution tree of <code>bad_abs</code> with symbolic value α assigned to input variable <code>x</code>	13
2.2	Concolic testing workflow.	13
3.1	CRETE architecture.....	19
3.2	CRETE workflow.	20
3.3	Example of standardized runtime trace.....	23
3.4	Execution tree of the example trace from Fig. 3.3: (a) for concrete execution, (b) for symbolic execution, and (c) for concolic execution....	27
3.5	Sample configuration file of CRETE Runner.....	28
3.6	Line coverage difference on COREUTILS by CRETE over KLEE and ANGR: positive values mean CRETE is better, and negative values mean CRETE is worse.....	34
3.7	Coverage improvement over seed test case by CRETE on GNU COREUTILS.	35
3.8	Coverage improvement over seed test case by CRETE on TianoCore utilities.	37
4.1	The interface of LKMs: (a) interactions between user application, base kernel and LKMs, and (b) a concrete example based on <code>ifconfig</code> and <code>e1000</code> . The interactions between hardware and LKMs are omitted.	46
4.2	The architecture of COD for automated test case generation.....	50
4.3	The architecture of COD for automated test case replay.	55
4.4	Excerpt of LKM <code>E1000</code> related to Bug 1 from Table 4.3.	65
4.5	Excerpt of JSON parser kernel module used for comparing with kAFL.	68
5.1	SoC stack on virtual platforms.	76
5.2	Excerpt of <code>E1000</code> virtual device from QEMU.	77
5.3	Architecture and workflow of end-to-end concolic testing for hardware/software co-validation: (1) execute SoC software stack over different VPs with partitioned VDs; (2) capture segmented traces from UOD, VD and firmware respectively; (3) assemble a system-level trace and inject system-level assertions; (4) inject symbolic values at HW/SW interfaces and perform concolic-symbolic hybrid execution, generating test cases to cover new usage of the SoC or trigger assertions.	80

5.4	Implementation of end-to-end concolic testing for hardware/software co-validation based on QEMU, Emu8051, and CRETE.	85
5.5	User-defined assertion examples in pseudo C code.....	93

1 Introduction

1.1 Overview

Computing systems are experiencing an explosive growth, both in complexities and diversities, ushered in by the proliferation of cloud computing, mobile computing, and Internet of Things. This growth has also exposed the consequences of unsafe, insecure, and unreliable computing systems, exemplified by recent high-profile security breaches and software system failures at major corporations such as British Airways [105] and Facebook [107]. These all point to the great needs of sophisticated system validation techniques. Recent advances in research on symbolic execution [58] has shown great promises for automated software analysis, e.g., generating test cases, finding bugs, and detecting security vulnerabilities [15–17, 25, 38, 39, 66, 81, 97, 100]. However, symbolic execution is mostly adopted to analyze user applications, while modern computing systems in practice consist of many components shipped by various vendors, besides user applications, e.g., operating systems, firmware and hardware devices. How to enable symbolic execution on modern computing systems remains a major challenge.

In this dissertation, we propose versatile binary-level concolic testing, which defines a standard execution-trace format, and features an open and highly extensible architecture. It allows easy integration of multiple concrete execution frontends and symbolic execution backends, which significantly improves the applicability and flex-

Figure 1.1: The overview of dissertation.

ibility of symbolic execution, especially to modern computing systems with various components. As shown in Fig. 1.1, this dissertation has three major pieces. First, we present CRETE, the infrastructure of versatile binary-level concolic testing, with the detail of its design, implementation, optimization, and evaluation, from which we demonstrate the proposed approach is scalable and effective to real-world applications. Second, we present COD, a framework based on versatile binary-level concolic testing for automated bug detection and replay of commercial off-the-shelf (COTS) Linux kernel modules. Third, we introduce how we leverage versatile binary-level concolic testing for hardware/software co-validation of Systems-on-Chips.

1.2 Infrastructure of Versatile Binary-level Concolic Testing

1.2.1 Problem Statement

There have been many recent approaches to symbolic execution [5, 6, 14, 55, 86–88, 91, 99, 102, 120]. Generally speaking, these approaches can be classified into two categories: online symbolic execution (e.g., BitBlaze [100], KLEE [15], and s²E [25]), and concolic execution (a.k.a., offline symbolic execution, e.g., CUTE [97], DART [38], and SAGE [39]). Online symbolic execution closely couples Symbolic Execution Engines (SEE) with the System Under Test (SUT) and explore all possible execution paths of

SUT online at once. On the other hand, concolic execution decouples SEE from the SUT through traces, which concretely runs a single execution path of a SUT and then symbolically executes it. Both online and offline symbolic execution are facing two major challenges for analyzing modern software systems: (1) the SUT involves many types of software for different hardware platforms and (2) the SUT involves many components distributed on different machines and as a whole the SUT cannot fit in any SEE.

What’s more, modern computing systems consist of many software components from various vendors, and access to all corresponding source code is rarely feasible. Even when source code is available, building the code exactly as in the shipped software product is difficult [12]. Moreover, even if the source code is available, compilers can optimize it in many unpredictable ways, such as undefined behaviors in C [23]. Thus, analyses of the software stack of computing systems ought to be at binary-level, in order to be practical and useful. Analysis at binary-level loses high-level semantics information from the source code that is critical for efficient symbolic analysis. It adds extra complications on top of the two open questions of symbolic execution, namely state explosion and expensive constraint solving. As a result, optimizations are required to deliver practical techniques that are using symbolic execution.

1.2.2 Solution Overview

Our approach focuses on how to extend concolic execution to satisfy the needs for analyzing modern software systems. There are two major observations behind our efforts on extending concolic execution:

- The decoupled architecture of concolic execution provides the flexibility in integrating new trace-captured frontends for modern platforms.

- The trace-based nature of concolic testing offers opportunities for synthesizing system-level traces from different components distributed on different machines.

We present CRETE, a versatile binary-level concolic testing framework, which features an open and highly extensible architecture allowing easy integration of concrete execution frontends and symbolic execution backends. CRETE’s extensibility is rooted in its modular design where concrete and symbolic execution is loosely coupled only through standardized execution traces and test cases. The standardized execution traces are LLVM-based, self-contained, and composable, providing succinct and sufficient information for SEE to reproduce the concrete executions. The CRETE framework is composed of:

- **A CRETE tracing plugin**, which is embedded in the concrete execution environment, captures binary-level execution traces of the SUT, and stores the traces in a standardized trace format.
- **A CRETE manager**, which archives the captured execution traces and test cases, schedules concrete and symbolic execution, and implements policies for selecting the traces and test cases to be analyzed and explored next.
- **A CRETE replayer**, which is embedded in the symbolic execution environment, performs concolic execution on standardized traces and collects test cases generated.

We have implemented the CRETE framework on top of QEMU [11] and KLEE, particularly the tracing plugin for QEMU, the replayer for KLEE, and the manager that coordinates QEMU and KLEE to exchange runtime traces and test cases and manages the policies for prioritizing runtime traces and test cases. To validate CRETE extensibility, we have also implemented a tracing plugin for the 8051 emulator [56].

Intel 8051 series are representatives of microprocessors widely used as independent micro-controllers for IoT devices or as integrated components in large-scale systems-on-chips. Binaries are typically executed on these controllers bare-metal without runtime systems. Application of concolic testing to such binaries requires different tracing frontends than those with extensive runtime system supports. The trace-based architecture of CRETE has enabled us to integrate such tracing frontends seamlessly.

We also introduced a set of optimizations that we have designed and implemented to the prototype of CRETE to make it scale to analyze real-world user applications. We employ dynamic taint analysis to perform selective binary-level tracing, from which the size of captured trace is reduced dramatically and makes binary-level symbolic analysis feasible for practical applications. We also enforce concolic test generation from symbolic analysis over the captured traces to reduce the redundancy of generated test cases.

We evaluated our prototype of CRETE on GNU COREUTILS programs, and compared with KLEE and ANGR, which are two state-of-art open-source symbolic executors for automated program analysis at source-level and binary-level. The evaluation of COREUTILS programs shows that CRETE achieved comparable code coverage as KLEE directly analyzing the source code of COREUTILS and generally outperformed ANGR. CRETE also found 84 distinct and previously unreported crashes on widely-used and extensively-tested utility programs for UEFI BIOS development. Most of crashes have been confirmed by the project maintainer and several patches have been applied to fix them. We also make CRETE implementation publicly available to the community at github.com/SVL-PSU/crete-dev.

1.3 Automated Bug Detection and Replay for COTS Linux Kernel Modules

1.3.1 Problem Statement

Linux kernel is widely used, e.g., 90 percent of the public cloud workloads were running on Linux in 2017 [27]; in the first quarter of 2019, 75 percent of smartphones are equipped with Android which uses Linux as its core [101]; all of the top 500 supercomputers use Linux at the end of 2018 [79]. To support these diverse computing environments, the size of the Linux kernel has been steadily growing, reaching over 24.7 million LOC [27], and is continually changing to improve security, performance or maintainability, as well as to support new devices, file systems, and hardware architectures [70].

Linux kernel is typically split into two parts, e.g., the base kernel and Loadable Kernel Modules (LKM) [93]. The base kernel provides essential services for user applications and LKMs, such as process management, memory management, and inner-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 [21]. Bugs and vulnerabilities in LKMs can easily lead to the 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. [3] 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 has been many recent approaches to verifying and testing the Linux kernel and LKMs [8, 9, 12, 26, 28, 40, 45, 51, 57, 65, 80, 92, 94, 119]. Static analysis is widely used [8, 12, 80, 119], yet 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 [57, 65, 92]. However, they either need to instrument and recompile the kernel [92], or does not produce actionable test cases [65] which are essential for reproducing and debugging detected kernel bugs. Recently, fuzzing has been trending in detecting security vulnerabilities in OS kernels [9, 26, 28, 40, 45, 51, 94]. Most of the work [40, 45, 94] focus on fuzzing through system call interfaces of Linux kernel which is often far away from the target LKMs and cannot effectively analyze target LKM behaviors. Many of other works are not applicable to COTS LKMs [9, 28]. In summary, existing approaches have two major limitations: (1) lack of 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.

1.3.2 Solution Overview

We propose 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 with 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 [20]. Together with kernel dynamic instrumentation Kprobe [61], 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.

1.4 Hardware/Software Co-validation of Systems-on-Chips

1.4.1 Problem Statement

Systems-on-Chips (SoC) are pervasive ranging from wearable devices to smart phones, autonomous vehicles, and cloud servers. They are ubiquitous in every aspect of our life. Such pervasiveness demands that SoC must be highly secure and robust; otherwise, the consequences can be dire, from loss of confidential information to endangerment of lives [89]. Meanwhile, SoC development cycles have been greatly compressed. A new generation of SoC is often released with new hardware yearly if not more frequently, while its software is updated regularly after the initial release. Effective and efficient validations are highly desired for assuring SoC quality, given such high time-to-market pressure.

In today’s practices, different components of SoC are validated separately. Hardware validation typically relies on simulation and formal methods [22, 47, 52]. The software shipped with a SoC, including firmware (FW), device driver and user applications, is often validated through manual code review, static analysis [80, 98], and

testing [29, 73–75]. System-level validation [90], i.e., validating the entire SoC system by validating all components on the SoC chip and their interactions, are often missing. Furthermore, system-level SoC validation is challenging. First, different components of SoC are validated with different tools and methods due to their semantic differences. It is challenging to have a unified framework that carries out validation across the entire SoC stack. Second, it is challenging to develop system-level test cases which can simultaneously exercise multiple components on the SoC stack effectively. Third, there lacks of interfaces to easily insert user-defined assertions [112] to validate properties that cover the logic and requirements across multiple components of the SoC.

1.4.2 Solution Overview

In this work, we propose an approach to end-to-end concolic testing for HW/SW co-validation of SoC. Based on the simulation of SoC with multiple virtual platforms (VPs), our framework captures run-time execution traces hierarchically across the entire SoC stack. We assemble the captured traces from different components of the SoC stack into a single holistic system-level trace, representing a concrete usage of the entire SoC. We also provide an instrumentation interface for custom validation and analysis over the system-level trace, where users can insert assertions checking properties crosscutting the SoC stack and introduce symbolic values at various HW/SW interfaces. The instrumented trace is replayed in a concolic/symbolic engine to generate new system-level test cases that either explore new paths of the SoC stack or trigger assertions. Assertion violations often indicate logic bugs. We implemented our framework with QEMU [11] as the host VP and Emu8051 [59] as IP Core VP. We evaluated our prototype by validating a key SoC hardware component, Intel E1000 [49] Ethernet adapter, with its host software stack and IP firmware. The experiments

demonstrate that our approach generates effective system-level test cases, and catch various bugs with user-defined assertions, including two bugs from the E1000 virtual device (VD) in QEMU.

1.5 Dissertation Outline

The remainder of this dissertation is organized as follows. Chapter 2 provides background on symbolic execution, concolic testing, the KLEE engine and the QEMU platform. Chapter 3 presents the design, optimizations, and implementation of CRETE, the infrastructure of versatile binary-level concolic testing. Chapter 4 elaborates how we apply versatile binary-level concolic testing to automatically detect and replay kernel bugs. Chapter 5 illustrates how we leverage versatile binary-level concolic testing for hardware/software co-validation of SoC. Chapter 6 concludes and discusses future work.

2 Background

2.1 Symbolic Execution and the KLEE Engine

Symbolic execution [10] 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 [18] to effectively explore the execution tree of the target program. An execution state from the symbolic execution of a program includes a statement counter, values of variables and a path condition. Since the inputs are symbolic, the values of variables are expressions over symbolic inputs, and the path condition is a Boolean expression over symbolic inputs. Figure 2.1 illustrates an example of symbolic execution. At the entry of function `bad_abs`, input `x` is assigned with symbolic value α , which allows all valid values of integer type. For each conditional branch related to symbolic inputs, if both paths are feasible, a new execution state will be forked from the current execution state. By updating path condition based on the branch condition, both paths of the conditional branch can be covered and explored. For this example, symbolic execution forks states twice for two conditional branches, covering three paths in the function.

KLEE [15] is a state-of-the-art symbolic execution engine which is open-sourced on Github and actively maintained [106]. It performs source-level symbolic executions for C programs, generates high-coverage test cases, and detects various kinds of bugs. Essentially, KLEE is a multi-path program interpreter over LLVM [68] bit-code with a

Figure 2.1: A simple function `bad_abs` in C with its symbolic execution tree: (a) Function `bad_abs` in C. (b) Symbolic execution tree of `bad_abs` with symbolic value α assigned to input variable `x`.

set of checkers for detecting common bugs (buffer overflow, div by 0, etc.). It requires users to compile their target C program into LLVM bit-code, and specify symbolic values through KLEE intrinsic. In our work, KLEE is leveraged to build prototypes of the approaches we proposed.

2.2 Concolic Testing

Figure 2.2: Concolic testing workflow.

Concolic Execution [54, 97] combines concrete and symbolic execution. It leverages a concrete execution path to guide symbolic execution to achieve better scal-

ability [16]. It has advantages over concrete execution since it only explores each execution path once based on path constraints, while it is more scalable than symbolic execution because it leverage information from concrete execution to augment symbolic execution. Figure 2.2 illustrates the basic workflow of concolic testing.

Given an initial test case, the software program under test is concretely executed. During the concrete execution, a trace of the concrete execution is captured, which mainly contains path constraints of the exercised path. By using an off-line constraint solver, each branch condition from the captured trace is negated to generate a new test case, aiming at covering new paths of the program under test. Newly generated test cases are fed back into the concrete execution. This process repeats until all paths of the program have been explored or a user specified condition is satisfied.

2.3 The QEMU Platform

QEMU [11] is a virtual machine that provides full-system emulation of a computing system. It supports to emulate different processor architectures, such as x86, SPARC, and ARM, and allows to run unmodified software stack on its emulated systems, denoted as the guest system. QEMU provides two modes, namely KVM mode and dynamic-binary-translation (DBT) mode. In KVM mode, QEMU leverages hardware virtualization features, e.g., *Intel VT-x* [50], to emulate guest execution with minimal overhead. In this mode, the guest OS architecture is required to be the same as the system architecture where QEMU is running on, denoted as the host system. In the DBT mode, the guest system architecture can be different from the host system. To support different guest systems, QEMU provides different DBT frontends which disassemble instructions of various guest systems to a unified intermediate representation, namely QEMU-ir. It also provides different DBT backends to translate QEMU-ir to instructions runnable on the host system. The QEMU-ir interpreter is a

special DBT backend, which aims to make QEMU host agnostic. In our work, we mainly leverage the DBT mode of QEMU with IR interpreter backend.

3 The Infrastructure of Versatile Binary-level Concolic Testing

In this chapter, we present CRETE, the infrastructure of versatile binary-level concolic testing. It features an open and highly extensible architecture allowing easy integration of concrete execution frontends and symbolic execution engine backends. CRETE’s extensibility is rooted in its modular design where concrete and symbolic execution is loosely coupled only through standardized execution traces and test cases. The standardized execution traces are LLVM-based, self-contained, and composable, providing succinct and sufficient information for symbolic execution engines to reproduce the concrete executions. We have implemented CRETE with KLEE as the symbolic execution engine and multiple concrete execution frontends such as QEMU and 8051 Emulator. We have evaluated the effectiveness of CRETE on GNU COREUTILS programs and TianoCore utility programs for UEFI BIOS. The evaluation of COREUTILS programs shows that CRETE achieved comparable code coverage as KLEE directly analyzing the source code of COREUTILS and generally outperformed ANGR. The evaluation of TianoCore utility programs found numerous exploitable bugs that were previously unreported.

In summary, the CRETE framework makes several key contributions:

- **Versatile concolic testing.** CRETE provides an open and highly extensible architecture allowing easy integration of different concrete and symbolic exe-

cution environments, which communicate with each other only by exchanging standardized traces and test cases. This significantly improves applicability and flexibility of concolic execution to modern platforms and is amenable to leveraging new advancements in symbolic execution.

- **Standardizing runtime traces.** CRETE defines a standard binary-level trace format, which is LLVM based, self-contained and composable. Such a trace is captured during concrete execution, representing an execution path of a SUT. It contains succinct and sufficient information for reproducing the execution path in other program analysis environment, such as for symbolic execution. Having standardized traces minimizes the need of converting traces for different analysis environment and provides a basis for common trace-related optimizations.
- **Implemented a CRETE prototype.** We have implemented CRETE with KLEE as the SEE backend and multiple concrete execution frontends such as QEMU and 8051 Emulator. CRETE achieved comparable code coverage on COREUTILS binaries as KLEE directly analyzing at source-level and generally outperformed ANGR. CRETE also found 84 distinct and previously unreported crashes on widely-used and extensively-tested utility programs for UEFI BIOS development. We also make CRETE implementation publicly available to the community at github.com/SVL-PSU/crete-dev.

3.1 Overview

During the design of the CRETE framework for binary-level concolic testing, we have identified the following design goals:

- **Binary-level In-vivo Analysis.** It should require only the binary of the SUT and perform analysis in its real execution environment.

- **Extensibility.** It should allow easy integration of concrete execution frontends and SEE backends.
- **High Coverage.** It should achieve coverage that is not significantly lower than the coverage attainable by source-level analysis.
- **Minimal Changes to Existing Testing Processes.** It should simply provide additional test cases that can be plugged into existing testing processes without major changes to the testing processes.

To achieve the goals above, we adopt an online/offline approach to concolic testing in the design of the CRETE framework:

- **Online Tracing.** As the SUT is concretely executed in a virtual or physical machine, an online tracing plugin captures the binary-level execution trace into a trace file.
- **Offline Test Generation.** An offline SEE takes the trace as input, injects symbolic values and generates test cases. The new test cases are in turn applied to the SUT in the concrete execution.

This online tracing and offline test generation process is iterative: it repeats until all generated test cases are issued or time bounds are reached. We extend this process to satisfy our design goals as follows.

- Execution traces of a SUT are captured in its unmodified execution environment on binary-level. The tracing plugin can be an extension into a VM (Sec. 3.2.1), a hardware tracing facility, or a dynamic binary instrumentation tool, such as PIN [78], and DynamoRIO [13].

Figure 3.1: CRETE architecture.

- The concrete and symbolic execution environments are decoupled by standardized traces (Sec. 3.2.2). As long as they can generate and consume standardized traces, they can work together as a cohesive concolic process.
- Optimization can be explored on both tracing and test case generation, for example, selective binary-level tracing to improve scalability (Sec. 3.2.3), and concolic test generation to reduce test case redundancy (Sec. 3.2.4). This makes high-coverage test generation on binary-level possible.
- The tracing plugin is transparent to existing testing processes, as it only collects information. Therefore, no change is made to the testing processes.

3.2 Design

In this section, we present the design of CRETE with a VM as the concrete execution environment. The reason for selecting a VM is that it allows complete access to the whole system for tracing runtime execution states and is generally accessible as mature open-source projects.

3.2.1 CRETE Architecture

As shown in Fig. 3.1, CRETE has four key components: **CRETE Runner**, a tiny helper program executing in the guest OS of the VM, which parses the configuration

file and launches the target binary program (TBP) with the configuration and test cases; **CRETE Tracer**, a comprehensive tracing plug-in in the VM, which captures binary-level traces from the concrete execution of the TBP in the VM; **CRETE Replayer**, an extension of the SEE, which enables the SEE to perform concolic execution on the captured traces and to generate test cases; **CRETE Manager**, a coordinator that integrates the VM and SEE, which manages runtime traces captured and test cases generated, coordinates the concrete and symbolic execution in the VM and the SEE, and iteratively explores the TBP.

CRETE takes a TBP and a configuration file as inputs, and outputs generated test cases along with a report of detected bugs. The manual effort and learning curve to utilize CRETE are minimal. It makes virtually no difference for users to setup the testing environment for the TBP in a CRETE instrumented VM than a vanilla VM. The configuration file is an interface for users to configure parameters on testing a TBP, especially specifying the number and size of symbolic command-line inputs and symbolic files for test case generation.

Figure 3.2: CRETE workflow.

The workflow of CRETE is shown in Figure 3.2. CRETE works in iterations and each iteration includes the following phases:

- **Binary Execution Phase:** CRETE Runner first loads the input binary and a test case into the guest OS. Then CRETE Runner executes the binary with the data defined in the test case as inputs. In this way, the binary is executed within VM in its native, unmodified guest OS environment.
- **Trace Capture Phase:** Along with the execution of the target program, CRETE Tracer captures the runtime information needed to constitute a runtime trace for symbolic analysis.
- **Trace Selection Phase:** CRETE Manager takes the captured trace as input and maintains a pool of traces. CRETE Manager then selects a trace from this pool and passes it to CRETE Replayer.
- **Offline Replaying Phase:** CRETE Replayer, in turn, invokes the SEE to execute the selected trace symbolically. The SEE performs concolic test case generation.
- **Test Selection Phase:** CRETE Manager receives newly generated test cases from the SEE and maintains a test case pool. CRETE Manager then selects one test case from the pool and sends it back to CRETE Runner to start the next iteration of CRETE. This workflow iterates until no more test cases can be generated or user-specified time bounds are reached.

3.2.2 Standardized Runtime Trace

To enable the modular and plug-and-play design of CRETE, a standardized binary-level runtime trace format is needed. A trace in this format must capture

sufficient information from the concrete execution, so the trace can be faithfully replayed within the SEE. In order to integrate a concrete execution environment to the CRETE framework, only a plug-in for the environment needs to be developed, so that the concrete execution trace can be stored in the standard file format. Similarly, in order to integrate a SEE into CRETE, the engine only needs to be adapted to consume trace files in that format.

We define the standardized runtime trace format based on the LLVM assembly language [69]. The reasons for selecting the LLVM instruction sets are: (1) it has become a de-facto standard for compiler design and program analysis [67, 68]; (2) there have been many program analysis tools based on LLVM assembly language [15, 32, 35, 42]. A standardized binary-level runtime trace is packed as a self-contained LLVM module that is directly consumable by a LLVM interpreter. It is composed of (1) a set of assembly-level basic blocks in the format of LLVM functions (2) a set of hardware states in the format of LLVM global variables (3) a set of CRETE-defined helper functions in LLVM assembly (4) a main function in LLVM assembly. The set of assembly-level basic blocks is captured from a concrete execution of a program. It is normally translated from another format (such as QEMU-IR) into LLVM assembly, and each basic block is packed as a LLVM function. The set of hardware states are runtime states along the execution of the TBP. It consists of CPU states, memory states and maybe states of other hardware components, which are packed as LLVM global variables. The set of helper functions are provided by CRETE to correlate captured hardware states with captured basic blocks, and open interface to SEE. The main function represents a concrete execution path of a program. It contains a sequence of calls to captured basic blocks (LLVM functions), and calls to CRETE-defined helper functions with appropriate hardware states (LLVM global variables).

An example of a standardized runtime trace of CRETE is listed in Fig. 3.3. The

Concrete Execution Path			Initial HW State		HW Side Effects		Standardized Trace as a LLVM Module
			CPU	Memory	CPU	Memory	
			r0,r1,...,rn [0x1234]				@init_state = {<r0,r1,...,rn>, <[0x1234]>} @side_effects = {<r1>,<[0x5678]>}
1	mem_ld r1, [0x1234]	BB_1					define asm_BB_1() { %1 = load * 0x1234 %2 = getelementptr @init_state, r0_offset %3 = load * %2 %4 = add %1, %3 %5 = getelementptr @init_state, r1_offset store %4, * %5 store %4, * 0x1234 br %4, %.path_true, %.path_false %.path_true: %.path_false: }; asm_BB_3() is omitted
2	add r1, r0						
3	mem_st [0x1234], r1						
4	Br r1, inst_5, xxx						
5	mem_ld r1, [0x5678]	BB_2			r1		external aync_state() ;crete helper function
6	add r1, r0				r1	[0x5678]	define main() { call sync_state(@init_state) call asm_BB_1() call sync_state(@side_effects) call asm_BB_3() }
7	mem_st [0x5678], r1						
8	Jump inst_9						
9	mem_ld r0, [0x1234]	BB_3		[0x1234]			
10	add r1, r0						
11	mem_st [0x5678], r1						
12	Br r0, xxx, inst_13						
13	nop	BB_4					
14	nop						

Figure 3.3: Example of standardized runtime trace.

first column of this figure is a complete execution path of a program with given concrete inputs. It is in the format of assembly-level pseudo-code. Assuming the basic blocks BB_1 and BB_3 are of interest and are captured by CRETE Tracer, while other basic blocks are not (see Sec. 3.2.3 for details). As shown in the second and third column of the figure, hardware states are captured in two categories, initial state and side-effects from basic blocks not being captured. As shown in the fourth column of the figure, captured basic blocks are packed as LLVM functions, and captured hardware states are packed as LLVM global variables in the standardized trace. A main function is also added making the trace a self-contained LLVM module. The main function first invokes CRETE helper functions to initialize hardware states, then it calls into the first basic block LLVM function. Before it calls into the second basic block LLVM function, the main function invokes CRETE helper functions to update hardware states. For example, before calling `asm_BB_3`, it calls function `sync_state` to update register `r1` and memory location `0x5678`, which are the side effects brought by BB_2.

3.2.3 Selective Binary-level Tracing

A major part of a standardized trace is assembly-level basic blocks which are essentially binary-level instruction sequences representing a concrete execution of a TBP. It is challenging and unnecessary to capture the complete execution of a TBP. First, software binaries can be very complex. If we capture the complete execution, the trace file can be prohibitively large and difficult for the SEE to consume and analyze. Second, as the TBP is executing, it is very common to invoke many runtime libraries (such as `libc`) of no interest to the testers. Therefore, an automated way of selecting the code of interest is needed.

CRETE utilizes Dynamic Taint Analysis (DTA) [95] to achieve selective tracing. The DTA algorithm is a part of CRETE Tracer. It tracks the propagation of tainted inputs, normally specified by users, during the execution of a program. It works on binary-level and in byte-wise granularity. By utilizing the DTA algorithm, CRETE Tracer only captures basic blocks that operate on tainted values, while only captures side-effects from other basic blocks. Assuming the tainted source is from memory location `0x1234`, storing the value of user's input to the program, in the example trace of Fig. 3.3. DTA captures basic block `BB_1` and `BB_3`, because both of them operates on tainted values, while other two basic blocks don't touch tainted values and are not captured by DTA.

CRETE Tracer captures the initial state of CPU by capturing a copy of the CPU state before the first interested basic block is executed. The initial CPU state is normally a set of register values. As shown in Fig. 3.3, the initial CPU state is captured before instruction (1). Naïvely, the initial memory state can be captured in the same way; however, the typical size of memory makes it impractical to dump entirely. To minimize the trace size, CRETE Tracer only captures the parts of memory

that are accessed by the captured read instructions, like instruction (1) and (9). The memory being touched by the captured write instructions, like instruction (3) and (11), can be ignored because the state of this part of the memory has been included in the write instructions and has been captured. As a result, CRETE Tracer monitors every memory read instruction that is of interest, capturing memory as needed on-the-fly. In the example above, there are two memory read instructions. CRETE Tracer monitors both of them, but only keeps the memory state taken from instruction (1) as a part of the initial state of memory, because instruction (1) and (9) access the same address.

The side effects of hardware states are captured by monitoring uncaptured write instructions of hardware states. In the example in Fig. 3.3, instructions (5) and (6) write CPU registers which cause side effects to the CPU state. CRETE Tracer monitors those instructions and keeps the updated register values as part of the runtime trace. As register `r1` is updated twice by two instructions, only the last update is kept in the runtime trace. Similarly, CRETE Tracer captures the side effect of memory at address `0x5678` by monitoring instruction (7).

3.2.4 Concolic Test Case Generation

While a standardized trace is a self-contained LLVM module and can be directly executed by a LLVM interpreter, it opens interfaces to SEE to inject symbolic values for test case generation. Normally SEE injects symbolic values by making a variable in source code symbolic. From source code level to machine code level, references of variables by names have become memory accesses by addresses. For instance, a reference of a concrete input variable of a program becomes an access of a piece of memory that stores the state of that input variable. CRETE injects self-defined helper function, `crete_make_concolic`, to the captured basic blocks while capturing

trace. This helper function provides the address and size of the piece of memory for injecting symbolic values, along with a name to offer better readability for test case generation. By catching this helper function, SEE can introduce symbolic values at the right time and right place.

A standardized trace in CRETE represents only a single path of a TBP as shown in Fig. 3.4 (a). Test case generation on this trace with naïve symbolic execution by SEE won't be effective, as it ignores the single path nature of the trace. As illustrated in Fig. 3.4 (b), native symbolic replay of CRETE trace produces execution states and test cases that are exponential to the number of branches within the trace. To get most effective results, SEE should adopt concolic test generation, by only negating encountered branch conditions from CRETE trace instead of forking states. As a result, the SEE in CRETE normally maintains only one program state during the offline test case generation with symbolic values. For a branch instruction from a captured basic block, if both of the paths are feasible given the collected constraints so far on the symbolic values, the SEE only keeps the program state of the path that was taken by the original concrete execution in the VM by adding the corresponding constraints of this branch instruction, while the state of the other path is killed after generating a test case for that state. The test cases generated from the killed states lead the TBP to a different execution path, as the last branch condition was negated. As shown in Fig. 3.4 (c), the number of test cases generated from concolic replay of CRETE trace is linear to the number of branches in that trace.

3.2.5 Bug and Runtime Vulnerability Detection

CRETE detects bugs and runtime vulnerabilities in two ways. First, all the native checks embedded in SEE are checked during the symbolic replay over the trace captured from concrete execution. If there is a violation to a check, a bug report

Figure 3.4: Execution tree of the example trace from Fig. 3.3: (a) for concrete execution, (b) for symbolic execution, and (c) for concolic execution.

is generated and associated with the test case that is used in the VM to generate this trace. Second, since CRETE does not change the native testing process and simply provides additional test cases that can be applied in the native process, all the bugs and vulnerability checks that are used in the native process are effective in detecting bugs and vulnerabilities that can be triggered by the CRETE generated test cases. For instance, Valgrind [83] can be utilized to detect memory related bugs and vulnerabilities along the paths explored by CRETE test cases.

3.3 Implementation

To demonstrate the practicality of CRETE, we have implemented its complete workflow with QEMU [11] as the frontend and KLEE [15] as the backend respectively. And to demonstrate the extensibility of CRETE, we have also developed the tracing plug-in for the 8051 emulator which readily replaces QEMU.

```

1 <crete>
2   <exec>./prog</exec>
3   <args>
4     <arg index="1" size="8" concolic="true"/>
5     <arg index="2" value="./data" concolic="false"/>
6   </args>
7   <files>
8     <file path="./data" size="10" concolic="true"/>
9   </files>
10  <stdin size="10" value="" concolic="true"/>
11 </crete>

```

Figure 3.5: Sample configuration file of CRETE Runner.

3.3.1 CRETE Runner

We implemented CRETE Runner as a simple executable in the guest OS. It reads the configuration file, communicates the configuration to the CRETE Tracer, and then launches the program under test with the parameter setup specified in the configuration file.

A sample configuration file is shown in Fig. 3.5. The configuration file is in XML format [63]. The user specifies the target executable’s file path, and what input will be treated as concolic which is also the taint source for DTA. In this example, lines (4) and (5) specify that there will be two command line arguments to the program. The first will be treated as concolic and the second will be treated as concrete, to refer to a file on disk. Line (8) specifies that the file itself referred by the second argument will be treated as concolic. Line (10) specifies that standard input will be treated as concolic and its size will be ten. For concolic argument or `stdin`, when its size is provided, its initial value is optional, and its default value is binary zero.

3.3.2 CRETE Tracer for qemu

QEMU uses dynamic-binary-translation to execute instructions of a guest platform on the host machine. QEMU divides its dynamic binary translation into two parts. Its frontend translates different guest instructions into a unified format, QEMU-IR, while the backend translates QEMU-IR to different host instructions. To give CRETE the best potential of supporting various guest platforms supported by QEMU, CRETE Tracer captures the basic blocks in the format of QEMU-IR. To convert captured basic blocks into standardized trace format, we implemented a QEMU-IR to LLVM translator based on the x86-LLVM translator of s²E [24]. We offload this translation from the runtime tracing as a separate offline process to reduce the runtime overhead of CRETE Tracer. QEMU maintains its own virtual states to emulate physical hardware state of a guest platform. For example, it utilizes virtual memory state and virtual CPU state to emulate states of physical memory and CPU. Those virtual states of QEMU are essentially source-level structs. CRETE Tracer captures hardware states by monitoring the runtime values of those structs maintained by QEMU. QEMU emulates the hardware operations by manipulating those virtual states through corresponding helper functions defined in QEMU. CRETE Tracer captures the side effects on those virtual hardware states by monitoring the invocation of those helper functions. As a result, the initial hardware states being captured are the runtime values of these QEMU structs, and the side effects being captured are the side effects on those structs from the uncaptured instructions.

3.3.3 CRETE Manager

CRETE Manager CRETE manager coordinates online tracing within QEMU and offline test generation within KLEE. It interacts with QEMU by accepting the

captured traces from concrete execution and dispatches the newly selected test cases. It interacts with KLEE by dispatching the selected traces and accepting the newly generated test cases. As shown in Figure 3.2, CRETE manager maintains a pool of captured traces and a pool of generated test cases. It makes trace and test case selections from these two pools respectively.

3.3.4 CRETE Replayer for klee

KLEE takes as input the LLVM modules compiled from C source code. As the CRETE trace is a self-contained LLVM module, CRETE Replayer mainly injects symbolic values and achieves concolic test generation. To inject symbolic values, CRETE Replayer provides a special function handler for CRETE interface function `crete_make_concolic`. KLEE is an online symbolic executor natively, which forks execution states on each feasible branches and explores all execution paths by maintaining multiple execution states simultaneously. To achieve concolic test generation, CRETE Replayer extends KLEE to generate test cases only for feasible branches while not forking states.

3.3.5 CRETE Tracer for 8051 Emulator

The 8051 emulator executes an 8051 binary directly by interpreting its instructions sequentially. For each type of instruction, the emulator provides a helper function. Interpreting an instruction entails calling this function to compute and change the relevant registers and memory states. The tracing plug-in for the 8051 emulator extends the interpreter. When the interpreter executes an instruction, a LLVM call to its corresponding helper function is put in the runtime trace. The 8051 instruction-processing helper functions are compiled into LLVM and incorporated into the runtime trace serving as the helper functions that map the captured instructions to the cap-

tured runtime states. The initial runtime state is captured from the 8051 emulator before the first instruction is executed. The resulting trace is of the same format as that from QEMU and is readily consumable by KLEE.

3.4 Experimental Results

In this section, we present the evaluation results of CRETE from its application to GNU COREUTILS [36] and Tianocore utility programs for UEFI BIOS [110]. Those evaluations demonstrate that CRETE generates effective test cases that are as effective in achieving high code coverage as the state-of-the-art tools for automated test case generation, and can detect serious deeply embedded bugs.

3.4.1 GNU Coreutils

Experiment Setup. GNU COREUTILS is a package of utilities widely used in Unix-like systems. The 87 programs from COREUTILS (version 6.10) contain 20,559 lines of code, 988 functions, 14,450 branches according to `lcov` [84]. The program size ranges from 18 to 1,475 in lines, from 2 to 120 in functions, and from 6 to 1,272 in branches. It is an often used benchmark for evaluating automated program analysis systems, including KLEE, MergePoint and others [6, 15, 114]. This is why we chose it as the benchmark to compare with KLEE and ANGR.

CRETE and ANGR generates test cases from program binaries without debug information, while KLEE requires program source code. To measure and compare the effectiveness of test cases generated from different systems, we rerun those tests on the binaries compiled with coverage flag and calculate the code coverage with `lcov`. Note that we only calculate the coverage of the code in GNU COREUTILS itself, and do not compute code coverage of the library code.

We adopted the configuration parameters for those programs from KLEE’s experi-

ment instructions¹. As specified in the instructions, we ran KLEE on each program for one hour with a memory limit of 1GB. We increased the memory limit to 8GB for the experiment on ANGR, while using the same timeout of one hour. CRETE utilizes a different timeout strategy, which is defined by *no new instructions being covered in a given time-bound*. We set the timeout for CRETE as 15 minutes in this experiment. This timeout strategy was also used by DASE [114] for its evaluation on COREUTILS. We conduct our experiments on an Intel Core i7-3770 3.40GHz CPU desktop with 16GB memory running 64-bit Ubuntu 14.04.5. We built KLEE from its release v1.3.0 with LLVM 3.4, which was released on November 30, 2016. We built ANGR from its mainstream on Github at revision `e7df250`, which was committed on October 11, 2017. CRETE uses Ubuntu 12.04.5 as the guest OS for its VM frontend in our experiments.

Table 3.1: Comparison of Average and Median Coverage by KLEE, ANGR, and CRETE on COREUTILS

Cov.	Line (%)			Function (%)			Branch (%)		
	KLEE	ANGR	CRETE	KLEE	ANGR	CRETE	KLEE	ANGR	CRETE
Average.	70.48	66.79	74.32	78.54	79.05	83.00	58.23	54.26	63.18
Median.	88.09	81.62	86.60	100	100	100	79.31	70.59	77.57

Comparison with klee and Angr. As shown in Table 3.1, our experiments demonstrate that CRETE achieves comparable test coverage to KLEE and generally outperforms ANGR. The major advantage of KLEE over CRETE is that it works on source code with all semantics information available. When the program size is small, symbolic execution is capable of exploring all feasible paths with given resources, such as time and memory. This is why KLEE can achieve great code coverage, such as line

¹<http://klee.github.io/docs/coreutils-experiments/>

Table 3.2: Distribution Comparison of Coverage Achieved by KLEE, ANGR, and CRETE on COREUTILS

Cov.	Line			Function			Branch		
	KLEE	ANGR	CRETE	KLEE	ANGR	CRETE	KLEE	ANGR	CRETE
90-100%	40	24	33	65	60	65	15	16	19
80-90%	15	22	25	12	8	10	27	12	17
70-80%	13	14	10	3	7	5	14	16	25
60-70%	9	12	10	2	4	3	9	15	6
50-60%	5	7	4	1	4	1	8	11	9
40-50%	1	2	3	1	1	2	8	7	6
0-40%	4	6	2	3	3	1	6	10	5

coverage over 90%, on more programs than CRETE, as shown in Table 3.2. KLEE requires to maintain execution states for all paths being explored at once. This limitation becomes bigger when size of program gets bigger. What’s more, KLEE analyzes programs within its own virtual environment with simplified model of real execution environment. Those models sometimes offer advantages to KLEE by reducing the complexity of the TBP, while sometimes they lead to disadvantages by introducing inaccurate environment. This is why CRETE gradually caught up in general as shown in Table 3.2. Specifically, CRETE gets higher line coverage on 33 programs, lower on 31 programs, and the same on other 23 programs. Figure 3.6 (a) shows the coverage differences of CRETE over KLEE on all 87 COREUTILS programs. Note that our coverage results for KLEE are different from KLEE’s paper. As discussed and reported in previous works [6, 114], the coverage differences are mainly due to the major code changes of KLEE, an architecture change from 32-bit to 64-bit, and whether manual system call failures are introduced.

Figure 3.6: Line coverage difference on COREUTILS by CRETE over KLEE and ANGR: positive values mean CRETE is better, and negative values mean CRETE is worse.

ANGR shares the same limitation as KLEE requiring to maintain multiple states and provide models for execution environment, while it shares the disadvantage of CRETE in having no access to semantics information. Moreover, ANGR provides models of environment at machine level supporting various platforms, which is more challenging compared with KLEE’s model. What’s more, we found and reported several crashes of ANGR from this evaluation, which also affects the result of ANGR. This is why ANGR performs worse than both KLEE and CRETE in this experiment. Figure 3.6 (b) shows the coverage differences of CRETE over ANGR on all 87 COREUTILS programs. While CRETE outperformed ANGR on majority of the programs, there is one program `printf` that ANGR achieved over 40% better line coverage than CRETE, as shown in the left most column in Fig. 3.6 (b). We found the reason is `printf` uses many string routines from `libc` to parse inputs and ANGR provides effective models for those string routines. Similarly, KLEE works much better on `printf` than CRETE.

Coverage Improvement over Seed Test Case. Since CRETE is a concolic testing

Figure 3.7: Coverage improvement over seed test case by CRETE on GNU COREUTILS.

framework, it needs an initial seed test case to start the test of a TBP. The goal of this experiment is to show that CRETE can significantly increase the coverage achieved by the seed test case that the user provides. To demonstrate the effectiveness of CRETE, we set the non-file argument, the content of the input file and the `stdin` to zeros as the seed test case. Of course, well-crafted test cases from the users would be more meaningful and effective to serve as the initial test cases. Figure 3.7 shows the coverage improvement of each program. On average, the initial seed test case covers 17.61% of lines, 29.55% of functions, and 11.11% of branches. CRETE improves the line coverage by 56.71%, function coverage by 53.44%, and branch coverage by 52.14% respectively. The overall coverage improvement on all 87 COREUTILS programs is significant.

Bug Detection. In our experiment on COREUTILS, CRETE was able to detect all three bugs on `mkdir`, `mkfifo`, and `mknod` that were detected by KLEE. This demonstrates that CRETE does not sacrifice bug detection capacity while working directly on binaries without debug and high-level semantic information.

3.4.2 TianoCore Utilities

Experiment Setup. TianoCore utility programs are part of the open-source project EDK2 [109], a cross-platform firmware development environment from Intel. It includes 16 command-line programs used to build BIOS images. The TianoCore utility

programs we evaluated are from its mainstream on Github at revision `75ce7ef` committed on April 19, 2017. According to `1cov`, the 16 TianoCore utility programs contain 8,086 lines of code, 209 functions, and 4,404 branches. Note that we only calculate the coverage of the code for TianoCore utility programs themselves, and do not compute the coverage of libraries.

The configuration parameters we used on those utility programs are based on our rough high-level understanding of these programs from their user manuals. We assigned each program a long argument of 16 Bytes, and four short arguments of 2 Bytes, along with a file of 10 Kilobytes. We conduct our experiments on the same platform with the same host and guest OS as we did for the COREUTILS evaluation, and set the timeout also as 15 minutes for each program.

High Coverage Test Generation From Scratch. For all the arguments and file contents in the parameter configuration, we set their initial value as binary zeros to serve as the seed test case of CRETE. Figure 3.8 shows that CRETE delivered high code coverage, above 80% line coverage, on 9 out of 16 programs. On average, the initial seed test case covers 14.56% of lines, 28.71% of functions, and 12.38% of branches. CRETE improves the line coverage by 43.61%, function coverage by 41.63%, and branch coverage by 44.63% respectively. Some programs got lower coverage because of: (1) inadequate configuration parameters; (2) error handling code triggered only by failed system calls; (3) symbolic indices for arrays and files not well handled by CRETE.

Bug Detection. To further demonstrate CRETE’s capability in detecting deeply embedded bugs, we performed a set of evaluations focusing on concolic file with CRETE on TianoCore utility programs. From the build process of a tutorial image, OvmfPkg, from EDK2, we extracted 509 invocations to TianoCore utility programs and the corresponding intermediate files generated, among which 37 unique invoca-

Figure 3.8: Coverage improvement over seed test case by CRETE on TianoCore utilities.

tions cover 6 different programs. By taking parameter configurations from those 37 invocations and using their files as seed files, we ran CRETE with a timeout of 2 hours on each setup, in which only files are made symbolic.

Table 3.3: Classified Crashes Found by CRETE on Tianocore Utilities: 84 unique crashes from 8 programs

Crash Type	Count	Severity	Crashed Programs
Stack Corruption	1	High (Exploitable)	VfrCompile
Heap Error	6	High (Exploitable)	GenFw
Write Access Violation	23	High (Exploitable)	EfiLdrImage, GenFw, EfiRom, GenFfs
Abort Signal	2	Medium (Signs of exploitable)	GenFw
Read Access Violation	45	Low (May not exploitable)	GenSec, GenFw, Split, GenCrc32, VfrCompile
Other Access Violation	7	Mixed	GenFw

Combining experiments on concolic arguments and concolic files, CRETE found 84 distinct crashes (by stack hash) from eight TianoCore utility programs. We used a GDB extension [34] to classify the crashes, which is a popular way of classifying

crashes for AFL users [2]. Table 3.3 shows that CRETE found various kinds of crashes including many exploitable ones, such as stack corruption, heap error, and write access violation. There are 8 crashes that are found with concolic arguments while the other 76 crashes are found with concolic files. We reported all those crashes to the TianoCore development team. So far, most of the crashes have been confirmed as real bugs, and ten of them have been fixed.

We now elaborate on a few sample crashes to demonstrate that the bugs found by CRETE are significant. **VfrCompile** crashed with a segmentation fault due to stack corruption when the input file name is malformed, e.g., `'\\.*a'` as generated by CRETE. This bug is essentially a format string exploit. **VfrCompile** uses function `vsprintf()` to compose a new string from a format string and store it in a local array with a fixed size. When the format string is malicious, like `'%*a'`, function `vsprintf()` will keep reading from the stack and the local buffer will be overflowed, hence causing a stack corruption. Note that CRETE generated a well-formed prefix for the input, `'\\.'`, which is required to pass the preprocessing check from **VfrCompile**, so that the malicious format string can attack the vulnerable code.

CRETE also exposed several heap errors on **GenFw** by generating malformed input files. **GenFw** is used to generate a firmware image from an input file. The input file needs to follow a very precise file format, because **GenFw** checks the signature bytes to decide the input file type, uses complex nested structs to parse different sections of the file, and conducts many checks to ensure the input file is well-formed. Starting from a seed file of 223 Kilobyte extracted from EDK2's build process, CRETE automatically mutated 29 bytes in the file header. The mutated bytes introduced a particular combination of file signature and sizes and offsets of different sections of the file. This combination passed all checks on file format, and directed **GenFw** to a vulnerable function which mistakenly replaces the buffer already allocated for storing the input

file with a much smaller buffer. Follow-up accesses of this malformed buffer caused overflow and heap corruption.

3.5 Related Work

3.5.1 DART, CUTE, and SAGE

DART [38] and CUTE [97] are both early representative work on concolic testing. They operate on the source code level. We further extend concolic testing and targets close-source binary programs. We also modularize concolic testing by loosely coupling concrete execution and symbolic execution only by standardized trace files based on the LLVM bitcode and test cases. SAGE [39] is a Microsoft internal concolic testing tool that particularly targets at X86 binaries on Windows. We are trying to provide a platform agnostic approach: as long as a trace from concrete execution can be converted into the LLVM-based trace format, it can be analyzed to generate test cases.

3.5.2 KLEE and S2E

KLEE [15] is a source-code-level symbolic execution tool that is built on the LLVM infrastructure [68] and is capable of generating high-coverage test cases for C programs. KLEE analyzes the LLVM bitcode compiled from the C SUT, symbolically explores the execution paths of the program, and generates a test case for each path explored. CRETE adopts KLEE as its SEE, and extends it to perform concolic execution on standardized binary-level traces. In CRETE, the binary level run-time execution trace is converted into the LLVM bitcode format and sent to KLEE for analysis. The test cases that KLEE generates are converted into the test case file format for the concrete execution environment.

s²E [25] provides a framework for developing tools for analyzing close-source software programs. It augments a Virtual Machine (VM) with a SEE and path analyzers. It features a tight coupling of concrete and symbolic execution. The execution of a SUT can cross back and forth between concrete and symbolic execution. CRETE, however, has taken a loosely coupled approach to the interaction of concrete and symbolic execution. CRETE captures complete execution traces of the SUT online and conducts whole trace symbolic analysis off-line. CRETE reduces runtime overhead to the concrete execution, conducts global optimization over traces in symbolic execution, and achieves modularity so that different virtual or physical machines and symbolic engines can be plugged in the CRETE framework. CRETE brings better scalability, applicability, and modularity.

3.5.3 BitBlaze, Mayhem, and MergePoint

BitBlaze [100] is an early representative work on binary analysis for computer security. It provides TEMU, a QEMU-based run-time analysis frontend, and VINE, a symbolic execution backend. TEMU and VINE were closely integrated into Rudder, a tool for symbolic execution of software binaries. BitBlaze, particularly Rudder, focuses on effective detection of security vulnerabilities by leveraging the close coupling of TEMU and VINE. Mayhem [17] and MergePoint [6] build on BitBlaze and further optimize the close coupling of their concrete execution frontend and symbolic analysis backend to improve their effectiveness in detecting exploitable software bugs. CRETE has a different focus on providing an open architecture for binary-level concolic testing that enables flexible integration of various concrete and symbolic execution environments and extends concolic testing to complex software systems that may involve binaries executing on various hardware platforms.

3.5.4 Angr

ANGR is an extensible Python framework for binary analysis using VEX [83] as an intermediate representation (IR). It implemented a number of existing analysis techniques and enabled the comparison of different techniques in a single platform. ANGR provides CLE to load the binary under test in its own virtual environment, and provides lifters to disassemble binary code into VEX IR, from where it conducts symbolic execution over VEX IRs. As ANGR performs in-vitro binary analysis, it requires to model the real execution environment for the binary under test, like system calls and common library functions. This is one of the biggest limitations of angr, because the environment model can never be complete nor accurate. CRETE, however, performs in-vivo binary analysis, by analyzing binary-level trace captured from unmodified execution environment of target binary. On the other hand, ANGR needs to maintain execution states for all paths being explored at once, while CRETE reduces memory usage dramatically by analyzing a program path by path and separates symbolic execution from trace capturing. What's more, CRETE produces LLVM-based, self-contained trace, which favors many program analysis tools.

3.5.5 Fuzzing Testing

Our work is also related to fuzz testing [37], a black or white box testing technique which tests software programs by inputting massive amounts of random data. A popular representative tool for fuzzing is AFL [1]. Fuzzing is fast and quite effective for bug detection; however, it can easily get stuck when a specific input, like magic number, is required to pass a check and explore new paths of a program. Concolic testing guides the generation of test cases by solving constraints from the source code or binary execution traces and is quite effective in generating complicated

inputs. Therefore, fuzzing and concolic testing are complementary software testing techniques.

3.6 Discussions

3.6.1 Parallelization for Scalability

The current workflow of CRETE does not exploit the potential of multiprocessing and parallelism, and exchanges data through slow socket communication across different components, which provides major opportunities for future optimizations. The modularity of CRETE enables parallelization. Since the concrete execution and symbolic execution of CRETE are separated through trace files and test cases, they can be executed in parallel, even on different machines. In addition, there can be multiple instances of both concrete and symbolic execution. This enables CRETE to leverage computing resources and scale to complex programs.

CRETE Manager manages those instances of concrete and symbolic execution. It selects test cases from the test case pool and launches concrete execution in QEMU, and selects traces from the trace pool and launches symbolic execution in KLEE. As new traces and test cases are generated, CRETE Manager also merges them into the corresponding pools. Particularly, for the trace pool, the trace has to be merge into the execution tree incrementally. Some traces may overlap with existing traces, therefore, have to be removed.

3.6.2 Extensibility and Applicability

The design of CRETE frontend as a plug-in for the VM makes CRETE inherently extensible to support programs running on different OS such as Linux and Windows and hardware architectures such as x86 and ARM. Meanwhile, CRETE is a highly

modular framework, which has great potential of being extended to support different concrete and symbolic execution environments or engines.

In term of applicability, CRETE can work with programs from different system levels, including user-level executables and libraries, kernel-level modules and drivers, and programs running on bare metal hardware such as BIOS and firmware. Since CRETE directly operates on binaries, it is applicable to programs written in different programming languages, such as assembly, C and C++, and potentially Java.

3.7 Summary

In this chapter, we have presented CRETE, a versatile binary-level concolic testing framework, which is designed to have an open and highly extensible architecture allowing easy integration of concrete execution frontends and symbolic execution backends. At the core of this architecture is a standardized format for binary-level execution traces, which is LLVM-based, self-contained, and composable. Standardized execution traces are captured by concrete execution frontends, providing succinct and sufficient information for symbolic execution backends to reproduce the concrete executions. We have implemented CRETE with KLEE as the symbolic execution engine and multiple concrete execution frontends such as QEMU and 8051 Emulator. The evaluation of COREUTILS programs shows that CRETE achieved comparable code coverage as KLEE directly analyzing the source code of COREUTILS and generally outperformed ANGR. The evaluation of TianoCore utility programs found numerous exploitable bugs that were previously unreported.

4 Automated Bug Detection and Replay for COTS Linux Kernel Modules

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 may not have access to their source code, making the validation even harder.

In this chapter, 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 kernel panic. By leveraging the replay capability of our framework, we patched all the reported 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. 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.

In summary, our approach 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.
- 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.

4.1 Background

In this section, we first introduce the interface of LKMs, and then we introduce kernel dynamic instrumentation, an important techniques COD is based on.

Figure 4.1: 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.

4.1.1 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 4.1 (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 4.1 (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 work.

4.1.2 Kernel Dynamic Instrumentation

We leverage Kprobe [61], 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.

4.2 Overview

In this section, we first present the high-level methodology of our approach. Then we introduce the definition of test cases for LKMs. At last, we discuss how our approach handles non-determinism and concurrency of the Linux kernel.

4.2.1 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 4.3.3). 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 4.3.4).
- 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 constraints on generated test cases (Section 4.3.5).
- 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 4.4). It is embedded with **kAPI checkers** (Section 4.4.2) to detect common kernel bugs and produce informative reports to boost bug analysis.

4.2.2 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 $f : \vec{\alpha} \rightarrow \tau$, 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 $\pi \triangleq (k_0, k_1, \dots, k_n)$ 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 is composed of function name, invocation site, LKM name, and index of user commands in the test harness.

4.2.3 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 concurrency nature of the kernel itself.

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

Figure 4.2: The architecture of COD for automated test case generation.

process (details in Section 4.3.4). 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 4.4).

4.3 Test Case Generation

4.3.1 Architecture and Workflow

As shown in figure 4.2, 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.

4.3.2 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.

4.3.3 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 underline 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()`.

4.3.4 COD Tracer

The **Tracer** produces runtime traces of COTS LKMs. Like traces of CRETE, the captured trace is a self-contained LLVM [68] 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 non-determinism of the kernel for tracing and test case generation.

4.3.5 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 `_kmallocc`'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**

Figure 4.3: The architecture of COD for automated test case replay.

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()`.

4.4 Test Case Replay

4.4.1 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 4.3, 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.

4.4.2 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 leverages `Kdump` [41] 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 [77].

4.4.3 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.

4.4.4 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 $\xi(p, q) = \frac{|\text{LCP}(p, q)|}{\max(|p|, |q|)}$.

For example, if $p = \text{abcd}$, and $q = \text{abcex}$, 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

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

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 $\xi(\pi_1, \pi_2) = \frac{\sum_{i=0}^x |\text{LCP}(c_i^1, c_i^2)|}{\sum_{i=0}^x \max(|c_i^1|, |c_i^2|)}$.

4.5 Implementation

We built a prototype of COD based on CRETE [20]. We extended its frontend, the VM QEMU [11], with 1.1k LOC for supporting concolic interface of LKMs, tracing multiple processes, and monitoring interrupts. We also extended CRETE backend, the symbolic engine KLEE [15], 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 [61] 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 [41]. We also defined interfaces for users to easily add new Kprobes and checkers in the format of C macros.

4.6 Experimental Results

In this section, we present the evaluation result of COD. First, we present the evaluation results of bug detection of COD in Section 4.6.1. It includes all new vulnerabilities that were found by COD, and evaluation of COD’s ability to find known vulnerabilities. Second, we measure the replayable rate of test cases generated by COD on both virtual and physical platforms in Section 4.6.2. 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 4.6.3. 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.

Table 4.1: 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

4.6.1 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 4.1 shows the list of LKMs we evaluated with COD, and Table 4.2 shows the test harnesses we used in our experiments. All the main LKMs have been released at least 14 years [111]. 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

Table 4.2: List of Test Harnesses

Idx.	Test Harness	Target
1	<pre>insmod <i>xxx</i> ifconfig ens4 up dhclient ens4 ip route add <i>xxx</i> dev ens4 ethtool <i>xxx</i> ens4 <i>xxx</i> ping -I ens4 -c 1 -W 1 <i>xxx</i> curl --interface ens4 <i>xxx</i> rmmod <i>xxx</i></pre>	LKMs from network subsys.
2	<pre>insmod <i>xxx</i> speaker-test -l 1 rmmod <i>xxx</i></pre>	LKMs from sound subsys.

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 [9, 26, 92].

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 4.1. 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 4.3, 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. The links of the submitted bugs are omitted for double-blind review purpose.

We now take **Bug 1** as an example to explain why COD is able to generate test

Table 4.3: 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

cases from COTS LKMs to trigger and report the new flaws in Table 4.3. 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 de-allocation function. By examining test cases triggering this bug, we found COD only flipped a single kernel API return from the initial test case. COD was able to 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

Table 4.4: 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%

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.

4.6.2 Test Case Replay

In this section, we measure the replayable rates (Section 4.4.4) of test cases generated by COD. We selected two examples from Table 4.1, namely **e1000** and **snd-hda-intel** along with their dependent LKMs, and used the test harness as shown in Table 4.2. We also evaluated each example with test harnesses that only contains ‘insmod/rmmod’, considering LKM complexity stems mostly from initialization and cleanup code [53]. 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 4.4 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.

4.6.3 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 4.3. 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 4.3, we took three steps to fix it. Figure 4.4 shows

```

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); ...}; ...; }

```

Figure 4.4: Excerpt of LKM E1000 related to Bug 1 from Table 4.3.

the code excerpt related to this bug.

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 mem-

ory de-allocation function. The `linux-image-dbg` package [4] 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 4.4.

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 4.4 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 4.4.

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 4.4. 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 4.3, are among the most common and critical bugs, while is notoriously difficult to debug and fix. As elaborated in Charm [104], 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.

4.6.4 Comparison with kAFL

To have an apple-to-apple comparison with kAFL [94], 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 [96] from multiple JSON tokens of the parsed input string and matches the hash value with the hash value of "LKM"(see Figure 4.5).

As kAFL requires special CPU features, e.g., *Intel VT-x* and *Intel-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

```

1 // Robert Sedgewick Hash function
2 unsigned int RSHash(const char* str,
3                     unsigned int len);
4
5 jsmn_parser p;
6 jsmntok_t tokens[5];
7 jsmn_init(&p);
8
9 int res = jsmn_parse(&p, input, len, tokens, 5);
10 if(res != 3 &&
11     tokens[0].type == JSMN_PRIMITIVE &&
12     tokens[1].type == JSMN_PRIMITIVE &&
13     tokens[2].type != JSMN_PRIMITIVE ) return;
14
15 char arr[3] = {input[tokens[0].start],
16               input[tokens[1].start],
17               input[tokens[2].start]};
18 if(RSHash(arr, 3) == RSHash("LKM", 3))
19     panic(KERN_INFO "LKM...\n");

```

Figure 4.5: Excerpt of JSON parser kernel module used for comparing with kAFL.

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 was able to trigger the crash within 16 hours after exercising 52K test cases on average, 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 [94]. Our work on COD is attacking this research problem by extending concolic execution to kernel modules. The experiment results demonstrate 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.

4.7 Related Work

4.7.1 Kernel Vulnerability Detection

Static analysis [12] 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 [119], WHOOP [30] Dr. Checker [80], DSAC [8], DEADLINE [116], DCNS [7]. 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 [44].

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 [1]) to kernel level,

e.g., TriforceAFL [45], kAFL [94], syzkaller [40], DIFUZE [28], and Razzer [51]. Robustness testing of Linux kernel with fault injection is also an effective technique. Two examples are KFI [26] and DHTest [9]. Many of these tools either still heavily rely on the source code availability [9, 28] that is not applicable to COTS kernel and LKMs, or perform blindly fuzzing and explore the same execution path repeatedly [26, 40, 45, 94] 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.

4.7.2 Symbolic and Concolic Execution

Symbolic execution [10] 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 [18] to effectively explore the execution tree of the target program. Concolic execution [97] leverages a concrete execution path to guide symbolic execution to achieve better scalability [16]. Both of them have been largely adopted for automated test case generation and bug detection of software on both source and binary level [6, 15, 17, 25, 39, 102, 103, 113, 117]. Some representative work of applying symbolic or concolic execution to kernel code are DDT [65], SymDrive [92], and CAB-Fuzz [57]. They heavily rely on source-level instrumentation to perform effective dynamic analysis [92], or do not produce actionable test cases [57, 65] that are crucial for efficient replay and debugging on detected problems [31].

4.7.3 Kernel Bug Patching and Mining

Our work is also related to automated kernel patching [70]. The Coccinelle project [76] allows software developers to write code manipulation rules with a generalization of the patch syntax [85], 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 [60, 71, 108], e.g., detecting resource leak with paired function [77].

4.8 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 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 [11]), COD cannot effectively analyze LKMs requiring unsupported hardware in the VM. Symbolic device [92] 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. The major effort will be re-designing a kernel hypercall interface to consume fuzzed buffer from kAFL, and convert it to a sequence of LKM invocations. All other parts of COD should be readily reused.

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 [62], 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 [71] 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.

4.9 Summary

In this chapter, 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.

5 Hardware/Software Co-validation of Systems-on-Chips

Many recent approaches have been proposed to improve the quality of Systems-on-Chips (SoC), mainly focusing on a specific part of the SoC, e.g., device driver, hardware, firmware, etc. System-level validation of the entire SoC stack remains a major challenge, and so far research on end-to-end validation of SoC that covers both hardware and software (HW/SW) components is comparatively sparse. In this chapter, we present an approach to end-to-end concolic testing for HW/SW co-validation of SoC [19]. Based on the simulation of SoC with multiple virtual platforms, we capture a set of run-time traces from different components of the entire SoC, and assemble them into holistic system-level traces. We also provide instrumentation interfaces over the SoC trace for custom validation and analysis, allowing insertions of user-defined assertions and symbolic values at various HW/SW interfaces. The instrumented trace is replayed in a concolic/symbolic engine to generate new system-level test cases that either explore new paths of the SoC stack or trigger assertions. We emulated a complete SoC stack based on several open-source projects, from which we demonstrated that our approach can generate effective system-level test cases which crosscut the entire HW/SW stack of SoC and pinpoint an IP firmware buggy path from the user inputs to the host SW, and can catch various bugs with user-defined assertions including two bugs of QEMU’s E1000 Virtual Device.

In summary, our approach makes the following contributions:

- **System-level analysis framework.** We propose an approach to capturing holistic system-level traces of a SoC emulated by multiple VPs, through end-to-end tracing and assembling. This enables system-level analysis over the entire SoC stack.
- **Instrumentation interface for custom validation and analysis.** We define an instrumentation interface over the SoC trace, allowing insertions of user-defined assertions and symbolic values at various HW/SW interfaces. This enables symbolic/concolic analysis of the captured trace to generate system-level test cases for exploring new SoC paths and validating system-level properties.
- **Prototype Implementation and Evaluation.** We emulated a complete SoC stack based on QEMU and Emu8051, implemented a prototype based on CRETE. Our evaluation shows that our prototype generates effective system-level tests, and catches various bugs using user-defined assertions.

5.1 Background

In this section, we first present how an entire SoC stack is emulated with multiple VPs, and then we introduce Virtual Devices (VDs) in QEMU [11], which are important hardware models used in our prototype.

5.1.1 SoC Stack over Virtual Platforms

A SoC stack consists of hardware and software components. The hardware components usually contain a main processor, RAM and other hardware IPs, such as GPU, network device, etc. Each hardware IP can have its own core and RAM. The SoC software components normally consist of host software stack and IP firmware. Host software stack includes applications, OS and drivers, which are running on the main

Figure 5.1: SoC stack on virtual platforms.

processor of the SoC. The driver provides a software interface to the SoC hardware components, enabling the OS to access SoC hardware functions. A set of applications can be shipped with a SoC as interfaces for users to manage and configure the SoC hardware. The IP firmware runs on its IP core, which provides fine control over the hardware IP and defines low-level functionalities. System-level validation over a SoC is to validate its entire HW/SW stack, which is the focus of this work.

VPs are software systems that emulate physical computing systems. By providing functionalities of the emulated system, a VP allows execution of software and OS written for a different architecture. As shown in Fig. 5.1, the hardware stack of a SoC can be emulated with multiple VPs. A host VP emulates the main processor with the main RAM, allowing native host software execution. An IP VD emulates the interface and hardware logic of a hardware IP. The IP core and RAM are emulated by a different VP, which supports running native IP firmware.

5.1.2 Virtual Devices in QEMU

We adopt QEMU as the host VP for our implementation, because it is a well-maintained open-source project, and supports various IP VDs. VDs are fully functional software models of hardware IP, and usually implemented as a part of the VP.

```

1 // Virtual device state
2 typedef struct E1000State_st {
3     uint32_t mac_reg[0x8000]; // Interface Registers
4     uint32_t rxbuf_size;      // Internal variables
5     ...;
6 } E1000State;
7 // Interface function: write interface register
8 void e1000_mmio_write(E1000State *s, uint64_t index,
9                       uint64_t val) {
10     s->mac_reg[index] = val;
11     switch(index) {
12         case RCTL:
13             s->rxbuf_size = rxbufsize(val); ...; break;
14         ...;
15     }
16 }
17 // Internal function: calculate receive buffer size
18 static int rxbufsize(uint32_t v) {
19     v &= E1000_RCTL_BSEX | E1000_RCTL_SZ_16384 | ...;
20     switch (v) {
21         case E1000_RCTL_SZ_512: return 512;
22         ...;
23     }
24 } ...

```

Figure 5.2: Excerpt of E1000 virtual device from QEMU.

For example, Fig. 5.2 is the E1000 VD implemented in QEMU. It defines a struct of `E1000State` to model E1000 hardware state, such as interface registers and receive buffer size. It also provides a set of interface functions, such as `e1000_mmio_write`, for QEMU to trigger VD internal functions. The VD internal functions, such as `rxbufsize`, realize internal functionalities of an IP hardware.

5.2 Preliminary Definitions

In this section, we introduce a set of preliminary definitions formalized in our approach.

Definition 2 (Trace) *A trace $\tau \triangleq \langle r, s, \pi \rangle$ of a system S is a triple, where r is a stimulus (or request) to system S , s is the state of system S before processing stimulus r , and $\pi \triangleq (i_0, i_1, \dots, i_n)$ is a sequence of machine-level instructions which represents an execution path of system S for processing stimulus r .*

Informally, a trace τ models that given a stimulus r , a system transits from state s following the execution path π . An instruction $i \in \pi$ is either a normal instruction that only interacts with the state of current system s , or a special instruction that triggers interactions with other systems. The trace τ is an abstract definition; thus applies to host software, virtual device, and firmware.

Definition 3 (Host Software Trace) *A Host Software Trace is a triple $\tau^h \triangleq \langle r^h, s^h, \pi^h \rangle$, where r^h is the host software stimulus which is application inputs from users. s^h is the state of SoC host VP, and is a pair $\langle s_{cpu}^h, s_{mem}^h \rangle$, where s_{cpu}^h is a set of CPU registers with their values, and s_{mem}^h is a set of memory cell values. π^h is the execution path of the host software.*

Informally, $i^h \in \pi^h$ is either an instruction that only interacts with state s^h , or a special operation that triggers interactions with VD such as MMIO and Port I/O

operations.

Definition 4 (Virtual Device Trace) *A Virtual Device Trace is a triple $\tau^v \triangleq \langle r^v, s^v, \pi^v \rangle$, where r^v is the VD stimulus which is the request from the device driver, the VD state s^v is a set of device registers with their values, and the VD execution path π^v is the execution path of the VD.*

Informally, $i^v \in \pi^v$ is either an instruction that only interacts with VD state s^v , or a special operation that triggers interactions with firmware, such as calls to interface functions of the IP Core VP.

Definition 5 (Firmware Trace) *A Firmware Trace is a triple $\tau^f \triangleq \langle r^f, s^f, \pi^f \rangle$, where r^f is the request from VD, $s^f \triangleq \langle s_{cpu}^f, s_{mem}^f \rangle$ is the state of IP Core VP which consists of CPU and memory states, π^f is the execution path of firmware.*

Definition 6 (SoC System-level Trace) *An SoC System-level Trace is a 5-tuple $\tau^S \triangleq \langle r^h, s^h, S^v, S^f, \pi^S \rangle$, where r^h and s^h are the stimulus and state of the host software respectively, $S^v \triangleq (s_0^v, s_1^v, \dots, s_m^v)$ is a sequence of VD states, $S^f \triangleq (s_0^f, s_1^f, \dots, s_n^f)$ is a sequence of FW states, and π^S is an execution path of the SoC.*

Informally, for the simplicity of presentation, we assume a SoC system-level trace τ^S contains only one stimulus from user input, and interacts with only one IP hardware. Hence, τ^S is composed of only one host SW trace with a sequence of VD traces and FW traces. The SoC execution path π^S is composed of instructions from host SW path π^h , VD path π^v , FW path π^f , and special NOP instructions (defined in Sec. 5.3.2). A SoC trace with multiple stimulus is simply a sequence of τ^S we defined. Also, a SoC trace with multiple IP hardware can be modeled by extending current definition with a set of VD states $\{S^v\}$ and FW states $\{S^f\}$.

Figure 5.3: Architecture and workflow of end-to-end concolic testing for hardware/-software co-validation: (1) execute SoC software stack over different VPs with partitioned VDs; (2) capture segmented traces from UOD, VD and firmware respectively; (3) assemble a system-level trace and inject system-level assertions; (4) inject symbolic values at HW/SW interfaces and perform concolic-symbolic hybrid execution, generating test cases to cover new usage of the SoC or trigger assertions.

5.3 Design

As shown in Fig. 5.3, our approach mainly has two phases, on-line tracing and off-line analysis. At run-time (on-line), we perform end-to-end tracing over the entire SoC stack emulated by multiple VPs, from which a sequence of traces is captured, including Host SW Traces, VD Traces, and Firmware Traces. Statically (off-line), we assemble segmented traces into a holistic system-level trace, provide instrumentation interfaces for user-defined assertions and symbolic values over the assembled trace, and utilize concolic/symbolic engines to generate test cases that either explore new usages of the SoC or trigger user-defined assertions.

5.3.1 End-to-End Run-time Tracing

As shown in Fig. 5.3, a set of tracers are provided to each SoC hardware component for run-time tracing. The tracer for host SW is an extension to the SoC host VP. When a target application is invoked, the tracer captures user inputs as r^h , and takes a snapshot of the VP’s CPU and memory as s^h . It also monitors the complete execution of host SW to capture a sequence of machine-level instructions as the execution path of host SW π^h . The VD tracer is a wrapper to the IP VD, which intercepts all interactions between the IP VD and the SoC host VP. For each host SW/VD interaction, the tracer captures the VD requests (e.g. `e1000_mmio_write` in Fig. 5.2) as r^v , and takes the snapshot of the VD state (e.g. `E1000State` in Fig. 5.2) as s^v . The π^v is a concrete execution path of the VD, and can be derived from the VD source code with the captured r^v . The tracer for firmware is an extension to the IP Core VP. For each request from VD, it captures the request input as r^f , takes a snapshot of the VP’s CPU and memory before the execution of the firmware as s^f , and monitors the complete execution of the firmware to capture a sequence of machine-level instructions as the execution path of firmware π^f . A unified instruction format is needed to make traces captured by different tracers compatible. We choose to use LLVM IR [69] as the unified instruction format. As a result, instruction translators may be needed as a part of the run-time tracer to produce instruction sequences in LLVM IR.

5.3.2 System-level Trace Assembling

As shown in Algorithm 1, our system-level trace assembler takes $\tau^h, \mathcal{T}^v, \mathcal{T}^f$ as inputs, where τ^h is the captured host SW trace, $\mathcal{T}^v$ is a sequence of captured VD traces, and $\mathcal{T}^f$ is a sequence of captured firmware traces. The *project* operator STATE

Algorithm 1: ASSEMBLE-SYS-TRACE ($\tau^h, \mathcal{T}^v, \mathcal{T}^f$)

```

1   $\langle r^h, s^h, \pi^h \rangle \leftarrow \tau^h$ 
2   $S^v \leftarrow \mathbf{map}(\text{STATE}, \mathcal{T}^v)$ ,  $S^f \leftarrow \mathbf{map}(\text{STATE}, \mathcal{T}^f)$ 
3   $\pi \leftarrow []$   $\triangleright$  initialize  $\pi$  to be an empty sequence
4  foreach  $i^h \in \pi^h$  do
5      if  $i^h$  is normal instruction then APPEND( $\pi, i^h$ )
6      else  $\triangleright i^h$  interacts with virtual device
7          APPEND( $\pi, \text{NopEnterVd}$ )
8           $\langle r^v, s^v, \pi^v \rangle \leftarrow \text{NEXT}(\mathcal{T}^v)$ 
9          foreach  $i^v \in \pi^v$  do
10             if  $i^v$  is normal instruction then APPEND( $\pi, i^v$ )
11             else  $\triangleright i^v$  interacts with firmware
12                 APPEND( $\pi, \text{NopEnterFw}$ )
13                  $\langle r^f, s^f, \pi^f \rangle \leftarrow \text{NEXT}(\mathcal{T}^f)$ 
14                 foreach  $i^f \in \pi^f$  do
15                     APPEND( $\pi, i^f$ )
16                 APPEND( $\pi, \text{NopLeaveFw}$ )
17             APPEND( $\pi, \text{NopLeaveVd}$ )
18 return  $\langle r^h, s^h, S^v, S^f, \pi \rangle$ 

```

takes a trace $\langle r, s, \pi \rangle$ and returns the state element s . The check on whether an instruction $i \in \pi$ is a special instruction or not is implementation-dependent, which is related to the instruction set of the target VP and how it emulates its interaction with VDs. Function APPEND(x, y) appends element y to the end of sequence x . Function NEXT(x) returns the first element in the sequence x , and removes the element from

x. Special NOPs in the execution path of the system-level trace include **NopEnterVd**, **NopLeaveVd**, **NopEnterFw**, and **NopLeaveFw** to assist the execution transition between host software and VD, or between VD and firmware. For example, **NopEnterVd** correlates a special host SW instruction with a VD stimulus r^v , synchronizes the VD state with s^v , and transfers execution to VD, while **NopLeaveVd** propagates VD's return value back to s^h and transfers execution back to host SW. **NopEnterFw** and **NopLeaveFw** provide similar functionalities at the VD/FW interface.

5.3.3 Instrumentation Interface for System-level Analysis

Algorithm 2: INTERPRET-SYS-TRACE ($\tau^S, callbacks$)

```

1  $\langle r^h, s^h, S^v, S^f, \pi^S \rangle \leftarrow \tau^S$ 
2 foreach  $i \in \pi^S$  do
3   switch  $i$  do
4     case NopEnterVd:  $s^v \leftarrow \text{NEXT}(S^v)$ 
5     case NopEnterFw:  $s^f \leftarrow \text{NEXT}(S^f)$ 
6     otherwise EXECUTE( $\langle s^h, s^v, s^f \rangle, i$ )
7   if  $i$  is Nops then
8     PROCESS-CALLBACKS( $\langle r^h, s^h, s^v, s^f \rangle, callbacks$ )

```

Algorithm 2 shows the algorithm to interpret the system-level trace τ^S . The interpreter iterate through all the instructions in the execution path π^S , and execute them in sequence. When **NopEnterVd** and **NopEnterFw** are encountered, we synchronize the VD and firmware states from the captured states accordingly. The *callbacks* defined as the second argument in algorithm 2 serves as an instrumentation interface to allow the user to easily inject custom functionalities and checks. The user-defined callback

functions are written in high-level programming languages, such as *C*. Those callback functions will be called at corresponding special **NOPs**. User-defined functions have access to various run-time states in the SoC trace τ^S , namely r^h , s^h , s^v , and s^f , and the information from HW/SW interactions, e.g. value writing from the IP driver to VD interface registers. In this way, users can check properties related the state of host SW, VD and FW, as well as properties related to HW/SW interactions. For example, users can put an assertion to check whether user inputs to IP applications from host SW can cut-across the entire SoC HW/SW stack, and directly control the execution state of IP firmware. Also, users can introduce symbolic values at different levels of SoC stack, mainly at the HW/SW interfaces. This enables more thorough exercise of the captured SoC trace, making concolic/symbolic hybrid execution possible. This also allows users to make trade-offs between soundness and completeness over analysis of the SoC trace, which is similar to various consistency models described in S2E [25].

5.4 Implementation

This section presents our full-stack emulation of SoC with two open-source VPs, and illustrates some implementation details of our prototype.

5.4.1 Full-stack Emulation of SoC

To build the full-stack emulation framework, we adopt QEMU as the SoC host VP, and use Emu8051 as the IP core VP, as shown in Fig. 5.4. QEMU emulates an X86 platform, a popular architecture of SoC processor, while Emu8051 emulates an 8051 platform, a representative architecture of IP cores. QEMU VDs are used to emulate IP hardware devices. We made several modifications to the QEMU VDs since native QEMU VDs are complete emulations of hardware IPs. They not only provide IP hardware interfaces to the SoC host VP, but also abstract all other functionalities

Figure 5.4: Implementation of end-to-end concolic testing for hardware/software co-validation based on QEMU, Emu8051, and CRETE.

with software models, including hardware logic, IP core and firmware. To provide a cohesive simulation of the SoC hardware stack, we extracted the logic parts which are typically implemented inside firmware from the VD and executed them on Emu8051. We modified VDs to communicate with Emu8051 for firmware transactions. The Emu8051 is modified with interface functions that take inputs from QEMU VDs, execute a firmware transaction, and return results to the IP VD. For the software stack, we run a full-stack of native Linux OS on the x86 platform as host software, including IP applications, OS kernel, and IP drivers. Given the limited availability of firmware and its emulators in public, we wrote our own IP firmware based on the IP specification for our prototype. The IP firmware is written in C, compiled to 8051 executable,

and executed on the Emu8051.

5.4.2 Tracers for qemu, qemu VD, and Emu8051

We leverage the tracing facility of CRETE [20] for QEMU to capture traces of SoC host software. CRETE supports full-stack system tracing, and outputs a self-contained trace in the format of LLVM IR [69], which contains user inputs, states of CPU and memory, and instruction sequences. CRETE treats all captured instructions as regular instructions that operate on CPU and memory. We augment CRETE with the capability to distinguish special instructions that requires VD operations. During runtime tracing, we intercept every memory transactions issued by QEMU’s virtual CPU, and monitor its address translation from virtual address to physical address. When the physical address is translated to a device address of the target IP, we capture the mapping from the CPU address to the device address. The device address is the index of device interface registers, and is the address that VD can understand and work with. In a similar way, we capture mappings between CPU port address to device address by intercepting QEMU’s processing of port IOs. The captured mappings are used to distinguish special operations, e.g. memory-mapped I/O (MMIO) or port I/O from normal operations when assembling system-level traces.

To capture IP VD traces, we combine both static and dynamic approaches. We dynamically capture (1) invocations to the VD interface function along with the VD states; (2) mappings between DMA address and CPU address by monitoring the DMA operations as a part of the IP VD. Statically, we transform the target IP VD source code into a self-contained LLVM module by writing stubs for its external functions. Together with the captured run-time information, we can reproduce the execution path for each VD transaction. Note that the calls to Emu8051 interface functions in the VD source code are the special instructions that require firmware transactions,

and can be easily distinguished from other normal operations.

To capture FW traces, we designed and implemented an Emu8051 tracer based on CRETE. Similar to tracing from QEMU, CRETE captures partial traces that represent single execution paths of the emulated firmware from Emu8051. Our observation shows that both the emulator itself and its software stack are much smaller and simpler comparing with QEMU and its software stack. We chose to capture complete states of Emu8051 at run-time. Especially, we have the whole firmware binary captured with the Emu8051 RAM as a part of the Emu8051 state. Similar to tracing VD states, we capture the complete Emu8051 state only once along with state differences between two firmware transactions. We also implemented a stand-alone interpreter of 8051 instructions based on Emu8051. By taking a complete 8051 state and a request input, the interpreter can reproduce a firmware transaction and returns its result.

5.4.3 System-level Trace Assembler

Our trace assembler provides a set of helper functions, and performs static instrumentation and recompilation over the segmented traces. The NOPs instructions discussed before are implemented as calls to our helper functions. To have a more fine-grained control over the host SW/VD/FW interactions, we defined more NOPs instructions than what is discussed in section 5.3.2. To distinguish MMIO operations, the assembler replaces all the memory operations from the host SW execution path π^h with calls to the helper function `cpu_mem_op`. This function mainly checks whether the current memory operation requires a device operation based on the captured mappings, and it invokes either `vd_mmio_op` for device operations or `normal_mem_op` for normal operations. The helper function `vd_mmio_op` translates CPU address to device address, and invokes VD interface functions from the VD trace, such as `e1000_mmio_write`. Port IO operations are easily distinguished, and

are replaced with a helper function `vd_port_io`. To bridge DMA operations, the assembler replaces the DMA functions invoked in QEMU VDs, such as `pci_dma_read`, with the helper function `vd_dma_op`. This helper function translates the DMA address to CPU address by looking up the captured mapping, and performs corresponding data copy. Calls to Emu8051 interface functions are the special instructions in the VD execution path π^v , and they are wrapped with calls to `helper_fw_enter` and `helper_fw_ret` by the assembler.

5.4.4 Interface for System-level Analysis

The interface is implemented as call-back functions of the helper functions as discussed in section 5.4.3. Each helper function mainly has two call back functions which are executed right before and after itself. For example, function `vd_mmio_op` has two call-backs, `pre_vd_mmio_op` and `post_vd_mmio_op`. By overwriting those call-back functions, users define custom checkers as functions written in C, which are compiled and linked to the system-level trace for analysis. As discussed in section 5.3.3, the checkers have access to execution states from various layers of the SoC stack. Users can define arbitrary assertions over those run-time states expressible in C language. Also, by calling helper function `make_symbolic` in the call backs, symbolic values can be introduced at SW/HW interfaces. Note that symbolic values introduced in this way are not restricted with any constraints, and may not be the feasible values at the HW/SW interfaces of the original SoC stack. This can improve the exploration efficiency but can introduce false alarms, which is a trade-off between completeness and soundness [25]. Section 5.5 presents examples of using this interface to inject system-level assertions and symbolic value.

5.4.5 Customized Concolic/symbolic Engine

We implemented an engine for concolic/symbolic hybrid execution based on CRETE. The concolic engine in CRETE was based on KLEE that is a symbolic engine. We augmented the CRETE engine with the ability to switch back and forth concolic and symbolic execution. To maximize the exploration efficiency, we perform concolic executions on the portion of SoC trace from host SW to respect its partial trace nature, and perform symbolic execution on the portion of SoC trace from VD and FW to utilize the full trace nature.

5.5 Experimental Results

In this section, we present the evaluations of our prototype, which demonstrates the usefulness and effectiveness of its two major capabilities: (1) system-level tracing over the entire SoC stack; (2) instrumentation interface for custom validation and analysis.

Experiment Setup. We evaluated our approach on validating an Ethernet IP, Intel E1000 Gigabyte Network Adapter [49]. The hardware interface and logic of the target Ethernet IP is emulated by the E1000 VD in QEMU. Table 5.1 shows the software stack of the emulated SoC. The off-the-shelf host software stack runs on the host x86 processor, while the firmware runs on the emulated IP 8051 core. The firmware mainly contains the IP control logic related to the Receive Control Register RCTL, and is implemented by us based on the Intel E1000 IP specification [49]. We conduct our experiments on an Intel Core i7-3770 3.40 GHz CPU desktop with 16 GB memory running 64-bit Ubuntu 14.04.5.

System-level Tracing. Table 5.2 shows the stimulus used in our experiments, which is a set of network application commands. Each stimulus contains some user

Table 5.1: Target SoC SW Stack

App.	<code>ifconfig</code> ,
	<code>ethtool</code>
OS	Ubuntu 16.04
IP Driver	e1000.ko
IP FW	Crafted

Table 5.2: The SoC Stimulus

#	Stimulus
1	<code>ifconfig eth mtu xxx</code>
2	<code>ifconfig eth ether xxx</code>
3	<code>ethtool -s eth speed xxx</code>
4	<code>ethtool -s eth advisory xxx</code>

inputs (denoted as **xxx** in the table). Different stimulus drives the target SoC following different usages, and produce different traces. Table 5.3 shows the sizes of segmented traces and system-level traces. Host SW traces are captured in binary-level, and hence their sizes are measured by the number of basic blocks. For VD and firmware traces, their sizes are measured by the number of corresponding transactions. System-level traces are self-contained LLVM modules, and their sizes are presented the same as normal software in terms of Kilobytes. We run our prototype on each stimulus three times. Table 5.4 shows the average runtime information of end-to-end tracing and assembling, including the time usage and maximum memory usage. In summary, the time usage of end-to-end tracing is from 15 to 33 seconds, and memory consumption is from 364 MB to 541 MB, while the trace assembler took 0.11 to 0.28 seconds, and consumed 40 MB to 140 MB of memory. It demonstrates that our prototype can capture system-level traces of the SoC with modest amount of time and memory.

Instrumentation Interface for Custom Validation and Analysis. In this experiment, we defined eight assertions to enforce eight system-level properties that the target SoC should hold. These properties are retrieved from the specification of Intel E1000 network adapter [49]. As shown in Table 5.5, these properties are related to the Receive Control Register (RCTL) and Control Register (CTRL) of E1000. Figure 5.5 shows three examples of our assertions, which illustrates how we leverage

Table 5.3: Sizes of SoC Traces Captured with Each Stimulus

Stimulus	Host SW	VD	Firmware	Sys-Level Trace
Index	(BasicBlock)	(Trans.)	(Trans.)	in LLVM (KB)
1	7368	1488	6	1858
2	93	67	0	957
3	692	1179	5	1115
4	660	1116	5	1081

Table 5.4: Runtime of End-to-end Tracing and Assembling

Stimulus	Tracing	Tracing	Assembling	Assembling
Index	Time (s)	Memory (MB)	Time (s)	Memory (MB)
1	32.33	540.97	0.28	140.27
2	32.67	364.38	0.11	39.98
3	15.33	437.76	0.15	76.54
4	16.33	395.07	0.15	72.58

the instrumentation interface of our framework to express and inject the assertions into the system-level traces. The assertions in the examples are mainly related to the interface register `RCTL`. For example, assertions $P1$ and $P2$ check properties related to MMIO at the interface of SW driver and VD. Assertions can also check on the entire SoC states, such as assertion $P3$ is based on both the VD state and user inputs to stimulus (network applications). We also injected symbolic values at host SW/VD and VD/FW interfaces, as well as the user inputs to stimulus. Figure 5.5 shows an example of injecting symbolic values to `value` which are the input of VD interface function passed from the IP driver. Then we applied our symbolic/concolic engine to the instrumented traces for system-level analysis and validation. Table 5.6 shows the

Table 5.5: System-level Properties Validated by User-defined Assertions

#	Description
1	The software should only write 0x00/0x01 to <code>RCTL.LBM</code> .
2	When <code>RCTL.BSEX</code> is 0x01, the software should not program value 0x00 to <code>RCTL.BSIZE</code> .
3	The values 0x10 and 0x11 are reserved to <code>RCTL.DTYP</code> .
4	When <code>RCTL.FLXBUF</code> is not 0x00, the receive buffer size is represented by the value of <code>RCTL.FLXBUF</code> in KB.
5	When <code>RCTL.DTYP</code> is 0x01, the buffer sizes for the descriptor are controlled by fields in the <code>PSRCTL</code> register.
6	The first byte of <code>RCTL</code> is reserved.
7	When a write to <code>CTRL</code> is finished, <code>CTRL.RST</code> should always be cleared by the device every time.
8	The value 0x11 is reserved to <code>CTRL.SPEED</code> .

experiment results over the trace captured from stimulus #1. It contains the number of test cases being generated, the number of assertions being validated, the number of assertions being violated, and the number of actual bugs being detected.

Results Analysis. Symbolic values of user inputs to stimulus generated the least number of test cases, and triggered the least number of assertions since they crosscut the entire SoC stack and accumulate complete constraints of the SoC execution. Following the strictest constraints also makes all generated test cases valid to the entire SoC, and hence does not introduce false alarms. Behind the only assertion failure triggered by the test cases of application inputs, we discovered a bug in the FW. Although this bug is hand-crafted, it demonstrates that our approach can precisely explore the impact of user inputs to the top level of the host SW stack across the

```

1 void pre_vd_mmio_op(uint32_t *reg_idx,
2                     uint32_t *value, int is_write) {
3     // Introduce symbolic values to VD input 'value'
4     make_symbolic(value, *value, sizeof(*value));
5
6     if(reg_idx == RCTL && is_write) {
7         //P1: Only 00/01'b is valid to LBM
8         uint8_t lbm = (*value >> LBM) & 0x03;
9         assert( lbm == 0x00 || lbm == 0x01);
10        //P2: 00'b is invalid to DTYP when BSEX is 01'b
11        uint8_t bsex = (*value >> BSEX) & 0x01;
12        uint8_t bsize = (*value >> BSIZE) & 0x03;
13        if(bsex == 0x01)
14            assert(bsize != 0x00);
15    }
16 }
17 void post_vd_mmio_op(uint32_t reg_idx,
18                     uint32_t value, int is_write) {
19     if(reg_idx == RCTL && is_write) {
20         //P3: when FLXBUF is not 00'b, use FLXBUF
21         //      in KB as the rx buffer size
22         uint8_t flxbuf = (value >> FLXBUF) & 0x0f;
23         if(flxbuf != 0x00)
24             assert(vd_state.rxbufsize == (flxbuf * 1024));
25     }
26 }

```

Figure 5.5: User-defined assertion examples in pseudo C code.

Table 5.6: Number of Generated Test-cases and Triggered Assertions from Concolic-symbolic Hybrid Execution

	User-inputs to Stimulus	Driver/VD Interface	VD/Firmware Interface
Generated Test Cases	20	1001	49
Validated Assertions	5	1	4
Fired Assertions	1	7	2
Detected Bugs	1	3	1

entire SoC HW/SW stack. It generated an exact test case that pinpoints the FW buggy path from the user inputs to the host SW.

Symbolic values of Driver/VD interfaces generated the largest number of test cases and triggered the largest number of assertions, while it has a much higher false rate on assertions triggered. By following partial constraints of the SoC stack, it is easier to explore partial stack more thoroughly, but it also produces test cases that might be invalid to the entire SoC stack. Manual efforts are needed to review all the triggered assertions. In our experiment, there are 7 triggered assertions in total where 3 of them are real alarms and report real bugs. Besides the FW bug we introduced, two bugs from the E1000 VD in QEMU are detected by our approach. One is reported by assertion *P3* as shown in Fig. 5.5, both of them are the functionalities that are required according to the Intel E1000 Manual while not being implemented in QEMU’s E1000 VD. Moreover, as the FW is written by us and has basic logic, the test cases and triggered assertions from VD/FW interface are much less comparing to those from Driver/VD interface.

5.6 Related Work

There has been a lot work on concolic and symbolic testing. Early work focused on bug detection and test generation for source-level applications, such as DART [38] and KLEE [15], while recent work is extended to binary analysis [115], such as S2E [25], Driller [102], and CRETE [20]. The closest related research is applying concolic testing to a certain part of the SoC stack with specialized optimizations. For example, S2E and SymDrive [92] are built for testing system software, especially device drivers, and Avatar [118] and Inception [29] focus on firmware validation. None of those systems is able to deal with a complete SoC stack, let alone perform system-level validation or analysis. Simply combining those systems together for the whole SoC validation is not feasible either, due to their dramatic differences of the architectures, designs, and implementations. To the best of our knowledge, our work is the first attempt to apply concolic testing to the entire SoC platforms end-to-end.

Much research has been done in the area of HW/SW co-validation which are close to our work. HW/SW co-verification is a common technique which mainly uses model checking to verify HW/SW interface protocols against the driver and various device models [64,82]. Recently research work leverages VDs for HW/SW co-validation and SoC validation [43,72]. Symbolic execution with VD co-verification is proposed to verify hardware and firmware interactions [46]. These work either focuses on device/driver interfaces or device/firmware interfaces. None of these approaches provide the same benefits as our approach, including interfaces for users to customize analysis at various SoC layers, and holistic system-level view of the entire SoC stack.

5.7 Discussion

We have shown that our approach enables end-to-end validation over the SoC, generates effective system-level test cases, and detects various bugs with user-defined assertions. However, there are limitations of this approach and directions for improvement, which we will discuss in this section.

5.7.1 Limitations

First, as a testing methodology, our approach provides SoC developers a framework to catch bugs from the entire SoC stack through system-level assertions. It only reports existence of bugs, and does not guarantee free of bugs. Complete verification [80] is not our goal.

Second, our approach may have false alarms, which can either stem from symbolic values with incomplete constraints as discussed in Section 5.5 or from imprecise hardware modeling, e.g., VD bugs being reported do not necessarily indicate bugs in the real IP hardware. Therefore, manual efforts are needed in confirming the raised alarms. We believe such manual efforts are justified since the reported bugs may not be real bugs in current SoC composition, but could well be real bugs in a new SoC using the involved IPs.

Third, as it currently stands, our approach is not well suited to detect bugs caused by hardware concurrency [48]. In our prototype, we only handle SoCs with one IP, and assume only one stimulus running at a time. In complex SoCs, there can be multiple IPs and stimulus running simultaneously, where concurrency can be a good source of errors and needs to be validated. We will address this in the future work.

5.7.2 Future Work

We are planning to apply our approach on the SoC with multiple IPs. Our prototype currently supports end-to-end tracing over several IPs, besides E1000, including EEPro100 (Intel PRO/100 Ethernet adapter), NE2000 (Novell’s Ethernet adapter), PCNet (AMD Ethernet adapter), and RTL8139 (Realtek Ethernet adapter). Enabling new IPs in our framework is relatively easy, taking us 1 to 3 developer days. We are extending our prototype to capture traces from multiple IPs and stimulus simultaneously. The other major challenge is to put together a meaningful SoC with multiple IPs, given the limited availability of real IP firmware and their emulates in public. We are actively looking for more interesting IP models, firmware, and VPs.

Second, we will explore techniques to scale our approach to large and complex SoC systems. One important direction is to enable the iterative process of our approach, applying newly generated test cases back to the end-to-end tracing process and repeating the whole validation process. We have a primitive iterative framework running, but it faces challenges of state explosion [25]. We are exploring additional optimizations, including minimizing sizes of segmented traces, biased search and iteration towards test cases triggering user-defined assertions, and adaptive selection strategies of SoC components to balance efficiency and effectiveness. What’s more, we will explore how to introduce symbolic values related to hardware concurrency, such as symbolic interrupt [92], so that our prototype can be applied to validate concurrency errors.

5.8 Summary

In this chapter, we presented an approach to HW/SW co-validation on the entire SoC stack through end-to-end concolic testing. Our approach captures run-time

traces of the entire SoC stack, assembles system-level traces, and provides instrumentation interfaces for custom validation and analysis, allowing insertions of user-defined assertions and symbolic values at HW/SW interfaces. New system-level test cases can be generated from the concolic/symbolic analysis over the SoC trace. Our prototype based on multiple open-source projects demonstrated the feasibility and effectiveness of our approach.

6 Conclusion

In this dissertation, we introduced versatile binary-level concolic testing, which significantly improves the applicability and flexibility of symbolic execution, especially to modern computing systems with various components. We also presented the designs, implementations and evaluations of several systems based on the proposed approach to make major components of modern computing systems more reliable and secure, including user applications, Linux kernel modules, firmware, and hardware devices. To conclude, this chapter summarizes the main contributions and highlights some directions for future research.

6.1 Summary of Contributions

Broadly, this dissertation advanced the state-of-the-art techniques for system validation, enriched the research communities of both academia and industry (specially for software engineering and security areas) by developing and maintaining open-source systems, and contributed directly to build more reliable and secure computing systems in the real world by detecting and fixing various unknown bugs and vulnerabilities in many important software systems. In summary, this dissertation makes the following specific contributions:

- Design and develop CRETE, the infrastructure of versatile binary-level concolic testing, to enable symbolic execution on modern computing systems, and scale it

with a set of optimizations, which delivered competitive results comparing with state-of-the-art tools for automated software analysis and detected numerous unknown bugs from various real-world applications.

- Design and develop COD, a system for automated bug detection and replay of COTS Linux kernel modules, which makes the Linux kernel more reliable and secure by detecting and fixing various unknown vulnerabilities.
- Design and develop an approach to HW/SW co-validation with end-to-end concolic testing, which helps tackle the challenge of system-level validation over the entire SoC stack.
- Release and maintain the open-source project for CRETE, which supported several research projects from both academia and industry, including Intel’s EXCITE project [33] for BIOS security validation.

6.2 Future Directions

In this section, we highlight some most interesting future directions, while more detailed discussions about limitations and future work can be found in previous sections (Sec. 3.6, Sec. 4.8, and Sec. 5.7).

Refined symbolic interfaces of user applications. All current symbolic and concolic engines, including CRETE, supports only standard symbolic interfaces of user applications, e.g., symbolic command-line, symbolic file, and symbolic `stdin`. One observation is that user applications are built upon commonly-used libraries shipped as a part of the operating system, such as `libc` on Linux systems. Those system libraries provide services to user applications through well-defined APIs, such as memory management, and socket communication. One future work is to add new symbolic interfaces from system library APIs to user applications. For example,

introducing symbolic values to the return value of `malloc` can help validate the error handling code of user applications and is promising to detect memory related issues. This idea is similar to what we have done in COD, in which we introduced symbolic values to the return values of kernel API to analyze Linux kernel modules.

Hybrid fuzzing for Linux kernel modules. 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 plan to combine the concolic engine in COD’s prototype with state-of-the-art fuzzers, e.g., kAFL, to perform hybrid fuzzing on LKM interfaces. By leveraging the precision of concolic execution and speed of fuzzing, we believe more Linux kernel vulnerabilities can be detected and fixed.

Improving the CRETE prototype. The current workflow of CRETE does not exploit the potential of multiprocessing and parallelism, and exchanges data through slow socket communication across different components, which provides major opportunities for future optimizations. Also, our current implementation of CRETE mainly focuses on the x86 architecture and QEMU/Emu8051 frontends. Therefore, another future work is to extend CRETE frontend with more architectures and more virtual machines to broaden the applicability of the CRETE framework.

Bibliography

- [1] AFL: American fuzzy lop. <http://lcamtuf.coredump.cx/afl/>
- [2] AFL-Utills: Utilities for automated crash sample processing/analysis. <https://github.com/rc0r/afl-utils>
- [3] Arnold, J., Abbott, T., Daher, W., Price, G., Elhage, N., Thomas, G., Kaseorg, A.: Security impact ratings considered harmful. In: Proceedings of HotOS'09: 12th Workshop on Hot Topics in Operating Systems, Monte Verità, Switzerland (2009)
- [4] Arnold, S.: Debug symbol packages. <https://wiki.ubuntu.com/Debug%20Symbol%20Packages> (2018)
- [5] Avgerinos, T., Cha, S.K., Rebert, A., Schwartz, E.J., Woo, M., Brumley, D.: Automatic exploit generation. *Commun. ACM* **57**(2), 74–84 (2014)
- [6] Avgerinos, T., Rebert, A., Cha, S.K., Brumley, D.: Enhancing symbolic execution with veritesting. In: 36th International Conference on Software Engineering, ICSE '14, Hyderabad, India. pp. 1083–1094 (2014)
- [7] Bai, J., Lawall, J., Tan, W., Hu, S.: 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. pp. 287–299 (2019)
- [8] Bai, J.J., Wang, Y.P., Lawall, J., Hu, S.M.: DSAC: Effective static analysis of sleep-in-atomic-context bugs in kernel modules. In: 2018 USENIX Annual Technical Conference (USENIX ATC 18). USENIX Association, Boston, MA (2018)
- [9] Bai, J.J., Wang, Y.P., Yin, J., Hu, S.M.: Testing error handling code in device drivers using characteristic fault injection. In: Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference. pp. 635–647. USENIX ATC '16, USENIX Association, Berkeley, CA, USA (2016)
- [10] Baldoni, R., Coppa, E., D’Elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. *ACM Comput. Surv.* **51**(3), 50:1–50:39 (2018)
- [11] Bellard, F.: Qemu, a fast and portable dynamic translator. In: Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference, Anaheim, CA, USA (2005)
- [12] Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., Engler, D.: A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM* **53**(2), 66–75 (2010)
- [13] Bruening, D., Zhao, Q., Amarasinghe, S.: Transparent dynamic instrumentation. In: Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments. pp. 133–144. VEE ’12, ACM, New York, NY, USA (2012)

- [14] Bucur, S., Kinder, J., Candea, G.: Prototyping symbolic execution engines for interpreted languages. In: Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 239–254. ASPLOS '14, ACM, New York, NY, USA (2014)
- [15] Cadar, C., Dunbar, D., Engler, D.: 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. pp. 209–224. OSDI'08 (2008)
- [16] Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. *Commun. ACM* **56**(2), 82–90 (2013)
- [17] Cha, S.K., Avgerinos, T., Rebert, A., Brumley, D.: Unleashing mayhem on binary code. In: IEEE Symposium on Security and Privacy, SP 2012, San Francisco, California, USA. pp. 380–394 (2012)
- [18] Cha, S., Hong, S., Lee, J., Oh, H.: Automatically generating search heuristics for concolic testing. In: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden. pp. 1244–1254 (2018)
- [19] Chen, B., Cong, K., Yang, Z., Wang, Q., Wang, J., Lei, L., Xie, F.: End-to-End Concolic Testing for Hardware/Software Co-Validation. In: Proceedings of the 15th IEEE International Conference on Embedded Software and Systems, ICESS 2019, Las Vegas, Nevada, USA (2019)
- [20] Chen, B., Havlicek, C., Yang, Z., Cong, K., Kannavara, R., Xie, F.: CRETE: A versatile binary-level concolic testing framework. In: Fundamental Approaches to Software Engineering. pp. 281–298. Springer International Publishing, Cham (2018)

- [21] Chen, H., Mao, Y., Wang, X., Zhou, D., Zeldovich, N., Kaashoek, M.F.: Linux kernel vulnerabilities: state-of-the-art defenses and open problems. In: APSys '11 Asia Pacific Workshop on Systems, Shanghai, China. p. 5 (2011)
- [22] Chen, W., Ray, S., Bhadra, J., Abadir, M.S., Wang, L.: Challenges and trends in modern soc design verification. IEEE Design & Test (2017)
- [23] Chipounov, V.: S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems. Ph.D. thesis, ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE (2014)
- [24] Chipounov, V., Candea, G.: Dynamically translating x86 to llvm using qemu. Technical Report EPFL-TR-149975 (2010)
- [25] Chipounov, V., Kuznetsov, V., Candea, G.: The s2e platform: Design, implementation, and applications. ACM Trans. Comput. Syst. **30**(1), 2:1–2:49 (2012)
- [26] Cong, K., Lei, L., Yang, Z., Xie, F.: Automatic fault injection for driver robustness testing. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis. pp. 361–372. ISSSTA 2015, ACM, New York, NY, USA (2015)
- [27] Corbet, J., Kroah-Hartman, G.: 2017 linux kernel development report. A Publication of The Linux Foundation (2017)
- [28] Corina, J., Machiry, A., Salls, C., Shoshitaishvili, Y., Hao, S., Kruegel, C., Vigna, G.: Difuze: Interface aware fuzzing for kernel drivers. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 2123–2138. CCS '17, ACM, New York, NY, USA (2017)

- [29] Corteggiani, N., Camurati, G., Francillon, A.: Inception: System-wide security testing of real-world embedded systems software. In: *USENIX Security 18*. pp. 309–326. USENIX Association, Baltimore, MD (2018)
- [30] Deligiannis, P., Donaldson, A.F., Rakamaric, Z.: 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*. pp. 166–177 (2015)
- [31] Devecsery, D.: Enabling Program Analysis Through Deterministic Replay and Optimistic Hybrid Analysis. Ph.D. thesis, The University of Michigan (2018)
- [32] Dhurjati, D., Kowshik, S., Adve, V.: Safecode: Enforcing alias analysis for weakly typed languages. In: *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. pp. 144–157. PLDI '06, ACM, New York, NY, USA (2006)
- [33] Engblom, Jakob: Finding BIOS Vulnerabilities with Symbolic Execution and Virtual Platforms. <https://software.intel.com/en-us/blogs/2017/06/06/finding-bios-vulnerabilities-with-excite> (2017)
- [34] Foote, J.: The 'exploitable' gdb plugin. <https://github.com/jfoote/exploitable>
- [35] Geoffray, N., Thomas, G., Lawall, J., Muller, G., Folliot, B.: Vmkit: A substrate for managed runtime environments. In: *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. pp. 51–62. VEE '10, ACM, New York, NY, USA (2010)
- [36] GNU: Gnu coreutils - core utilities. <https://www.gnu.org/s/coreutils>

- [37] Godefroid, P.: Random testing for security: Blackbox vs. whitebox fuzzing. In: Proceedings of the 2Nd International Workshop on Random Testing: Co-located with the 22Nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007). pp. 1–1. RT '07, ACM, New York, NY, USA (2007)
- [38] Godefroid, P., Klarlund, N., Sen, K.: Dart: Directed automated random testing. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 213–223. PLDI '05, ACM, New York, NY, USA (2005)
- [39] Godefroid, P., Levin, M.Y., Molnar, D.A.: SAGE: whitebox fuzzing for security testing. *Commun. ACM* **55**(3), 40–44 (2012)
- [40] Google: syzkaller - kernel fuzzer. <https://github.com/google/syzkaller> (2019)
- [41] Goyal, V., Biederman, E.W., Nellitheertha, H.: Kdump, a kexec-based kernel crash dumping mechanism. In: Proc. of the Linux Symposium (2005)
- [42] Grosser, T., Größlinger, A., Lengauer, C.: Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters* **22**(4) (2012)
- [43] Gu, H., Chen, M., Wei, T., Lei, L., Xie, F.: Specification-driven automated conformance checking for virtual prototype and post-silicon designs. In: Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA (2018)
- [44] Guo, P.J., Engler, D.: Linux kernel developer responses to static analysis bug reports. In: Proceedings of the 2009 Conference on USENIX Annual Technical

- Conference. pp. 22–22. USENIX’09, USENIX Association, Berkeley, CA, USA (2009)
- [45] Hertz, J., Newsham, T.: The triforceafl project. <https://github.com/nccgroup/TriforceAFL.git> (2017)
- [46] Horn, A., Tautschnig, M., Val, C., Liang, L., Melham, T., Grundy, J., Kroening, D.: Formal Co-Validation of Low-Level Hardware/Software Interfaces. In: Proc. of FMCAD (2013)
- [47] Hu, M., Wei, T., Zhang, M., Mallet, F., Chen, M.: Sample-guided automated synthesis for CCSL specifications. In: Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA. pp. 98:1–98:6 (2019)
- [48] Huang, B., Ray, S., Gupta, A., Fung, J.M., Malik, S.: Formal security verification of concurrent firmware in socs using instruction-level abstraction for hardware. In: Proceedings of the 55th Annual Design Automation Conference, San Francisco, CA, USA (2018)
- [49] Intel: Pci/pci-x family of gigabit ethernet controllers software developer’s manual. <https://www.intel.com/content/dam/doc/manual/pci-pci-x-family-gbe-controllers-software-dev-manual.pdf> (2009)
- [50] Intel Corporation: Intel® Virtualization Technology (Intel® VT). <https://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html> (2019)

- [51] Jeong, D.R., Kim, K., Shivakumar, B.A., Lee, B., Shin, I.: Razzer: Finding kernel race bugs through fuzzing. In: IEEE Symposium on Security and Privacy, SP 2019, San Francisco, California, USA (2019)
- [52] Jiang, Y., Song, H., Yang, Y., Liu, H., Gu, M., Guan, Y., Sun, J., Sha, L.: Dependable model-driven development of CPS: from stateflow simulation to verified implementation. TCPS **3**(1) (2019)
- [53] Kadav, A., Swift, M.M.: Understanding modern device drivers. In: Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK. pp. 87–98 (2012)
- [54] Kannavara, R., Havlicek, C.J., Chen, B., Tuttle, M.R., Cong, K., Ray, S., Xie, F.: Challenges and opportunities with concolic testing. In: 2015 National Aerospace and Electronics Conference (NAECON). pp. 374–378 (2015)
- [55] Kasikci, B., Zamfir, C., Candea, G.: Automated classification of data races under both strong and weak memory models. ACM Trans. Program. Lang. Syst. **37**(3), 8:1–8:44 (2015)
- [56] Kasolik, M.: 8051 emulator. <http://emu51.sourceforge.net/>
- [57] Kim, S.Y., Lee, S., Yun, I., Xu, W., Lee, B., Yun, Y., Kim, T.: Cab-fuzz: Practical concolic testing techniques for COTS operating systems. In: 2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA. pp. 689–701 (2017)
- [58] King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)

- [59] Komppa, J.: 8051/8052 emulator. <https://github.com/jarikomppa/emu8051>
- [60] Koyuncu, A., Liu, K., Bissyandé, T.F., Kim, D., Klein, J., Monperrus, M., Traon, Y.L.: Fixminer: Mining relevant fix patterns for automated program repair. CoRR **abs/1810.01791** (2018)
- [61] Krishnakumar, R.: Kernel korner: Kprobes-a kernel debugger. Linux J. **2005**(133), 11– (2005)
- [62] Kroah-Hartman, G.: Stable api nonsense. <https://www.kernel.org/doc/Documentation/process/stable-api-nonsense.rst>
- [63] Kuchling, A.: Xml, the extensible markup language. Linux J. **1998**(55es) (1998)
- [64] Kurshan, R.P., Levin, V., Minea, M., Peled, D., Yenigün, H.: Combining software and hardware verification techniques. Formal Methods in System Design (FMSD) **21**(3), 251–280 (2002)
- [65] Kuznetsov, V., Chipounov, V., Candea, G.: Testing closed-source binary device drivers with ddt. In: Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference. pp. 12–12. USENIXATC’10, USENIX Association, Berkeley, CA, USA (2010)
- [66] Kuznetsov, V., Kinder, J., Bucur, S., Candea, G.: Efficient state merging in symbolic execution. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 193–204. PLDI ’12, ACM, New York, NY, USA (2012)
- [67] Lattner, C.: LLVM and Clang: Next generation compiler technology. In: The BSD Conference. pp. 1–2 (2008)

- [68] Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization. pp. 75–. CGO '04, IEEE Computer Society, Washington, DC, USA (2004)
- [69] Lattner, C., Adve, V.: LLVM language reference manual. <http://llvm.org/docs/LangRef.html> (2006)
- [70] Lawall, J., Muller, G.: Coccinelle: 10 years of automated evolution in the linux kernel. In: 2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA. pp. 601–614 (2018)
- [71] Lawall, J., Palinski, D., Gnirke, L., Muller, G.: 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. pp. 15–26 (2017)
- [72] Lei, L., Cong, K., Yang, Z., Chen, B., Xie, F.: Hardware/Software Co-monitoring. In: CoRR, arXiv:1905.03915 [cs.SE] (2019)
- [73] Lin, B., Cong, K., Yang, Z., Liao, Z., Zhan, T., Havlicek, C., Xie, F.: Concolic testing of systemc designs. In: 19th International Symposium on Quality Electronic Design, ISQED 2018, Santa Clara, CA, USA. pp. 1–7 (2018)
- [74] Lin, B., Xie, F.: Scbench: A benchmark design suite for systemc verification and validation. In: 23rd Asia and South Pacific Design Automation Conference, ASP-DAC 2018, Jeju, Korea (South). pp. 440–445 (2018)

- [75] Lin, B., Yang, Z., Cong, K., Xie, F.: Generating high coverage tests for system designs using symbolic execution. In: 21st Asia and South Pacific Design Automation Conference, ASP-DAC 2016, Macao, Macao. pp. 166–171 (2016)
- [76] LIP6: The coccinelle project. <http://coccinelle.lip6.fr/>
- [77] Liu, H., Wang, Y., Bai, J., Hu, S.: PF-Miner: A practical paired functions mining method for android kernel in error paths. *Journal of Systems and Software* **121**, 234–246 (2016)
- [78] Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 190–200. PLDI '05, ACM, New York, NY, USA (2005)
- [79] Lunduke, B.: Linux and supercomputers. <https://www.linuxjournal.com/content/linux-and-supercomputers> (2018)
- [80] Machiry, A., Spensky, C., Corina, J., Stephens, N., Kruegel, C., Vigna, G.: DR. CHECKER: A soundy analysis for linux kernel drivers. In: USENIX Security Symposium. pp. 1007–1024. USENIX Association (2017)
- [81] Marinescu, P.D., Cadar, C.: Make test-zesti: A symbolic execution solution for improving regression testing. In: Proceedings of the 34th International Conference on Software Engineering. pp. 716–726. ICSE '12, IEEE Press, Piscataway, NJ, USA (2012)
- [82] Mukherjee, R., Purandare, M., Polig, R., Kroening, D.: Formal techniques for effective co-verification of hardware/software co-designs. In: Proceedings of

the 54th Annual Design Automation Conference, DAC 2017, Austin, TX, USA (2017)

- [83] Nethercote, N., Seward, J.: Valgrind: A framework for heavyweight dynamic binary instrumentation. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 89–100. PLDI '07, ACM, New York, NY, USA (2007)
- [84] Oberparleiter, P.: A graphical front-end for gcc's coverage testing tool gcov. <http://ltp.sourceforge.net/coverage/lcov.php>
- [85] Padiou, Y., Lawall, J.L., Hansen, R.R., Muller, G.: Documenting and automating collateral evolutions in linux device drivers. In: Proceedings of the 2008 EuroSys Conference, Glasgow, Scotland, UK. pp. 247–260 (2008)
- [86] Palikareva, H., Cadar, C.: Multi-solver support in symbolic execution. In: Proceedings of the 25th International Conference on Computer Aided Verification. pp. 53–68. CAV'13, Springer-Verlag, Berlin, Heidelberg (2013)
- [87] Palikareva, H., Kuchta, T., Cadar, C.: Shadow of a doubt: Testing for divergences between software versions. In: Proceedings of the 38th International Conference on Software Engineering. pp. 1181–1192. ICSE '16, ACM, New York, NY, USA (2016)
- [88] Ramos, D.A., Engler, D.: Under-constrained symbolic execution: Correctness checking for real code. In: Proceedings of the 24th USENIX Conference on Security Symposium. pp. 49–64. SEC'15, USENIX Association, Berkeley, CA, USA (2015)

- [89] Ray, S., Peeters, E., Tehranipoor, M.M., Bhunia, S.: System-on-chip platform security assurance: Architecture and validation. *Proceedings of the IEEE* **106**(1) (2018)
- [90] Ray, S., Yang, J., Basak, A., Bhunia, S.: Correctness and security at odds: post-silicon validation of modern soc designs. In: *Proceedings of Annual Design Automation Conference*, San Francisco, CA, USA (2015)
- [91] Redini, N., Machiry, A., Das, D., Fratantonio, Y., Bianchi, A., Gustafson, E., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Bootstomp: On the security of bootloaders in mobile devices. In: *26th USENIX Security Symposium (USENIX Security 17)*. pp. 781–798. USENIX Association, Vancouver, BC (2017)
- [92] Renzelmann, M.J., Kadav, A., Swift, M.M.: Symdrive: Testing drivers without devices. In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. pp. 279–292. OSDI’12, USENIX Association, Berkeley, CA, USA (2012)
- [93] Salzman, P.J., Burian, M., Pomerantz, O.: The linux kernel module programming guide. <http://www.tldp.org/LDP/lkmpg/2.6/html/> (2007)
- [94] Schumilo, S., Aschermann, C., Gawlik, R., Schinzel, S., Holz, T.: kaff: Hardware-assisted feedback fuzzing for OS kernels. In: *26th USENIX Security Symposium, USENIX Security 2017*, Vancouver, BC, Canada. pp. 167–182 (2017)
- [95] Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: *Proceedings of the 2010 IEEE Symposium on Security and*

- Privacy. pp. 317–331. SP '10, IEEE Computer Society, Washington, DC, USA (2010)
- [96] Sedgewick, R.: Algorithms in C. Wiley (1990)
- [97] Sen, K., Marinov, D., Agha, G.: 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. pp. 263–272. ESEC/FSE-13, ACM, New York, NY, USA (2005)
- [98] Sherman, E., Dwyer, M.B.: Structurally defined conditional data-flow static analysis. In: Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2018 (2018)
- [99] Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Krügel, C., Vigna, G.: SOK: (state of) the art of war: Offensive techniques in binary analysis. In: IEEE Symposium on Security and Privacy. pp. 138–157. SP '16, IEEE Computer Society (2016)
- [100] Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: Bitblaze: A new approach to computer security via binary analysis. In: Proceedings of the 4th International Conference on Information Systems Security. pp. 1–25. ICISS '08, Springer-Verlag, Berlin, Heidelberg (2008)
- [101] StatCounter: Mobile operating system market share worldwide. <http://gs.statcounter.com/os-market-share/mobile/worldwide> (2019)
- [102] Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: Augmenting Fuzzing Through

- Selective Symbolic Execution. In: Proceedings of the Network and Distributed System Security Symposium. NDSS '16, The Internet Society (2016)
- [103] Sun, Y., Wu, M., Ruan, W., Huang, X., Kwiatkowska, M., Kroening, D.: Concolic testing for deep neural networks. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France. pp. 109–119 (2018)
- [104] Talebi, S.M.S., Tavakoli, H., Zhang, H., Zhang, Z., Sani, A.A., Qian, Z.: Charm: Facilitating dynamic analysis of device drivers of mobile systems. In: 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA. pp. 291–307 (2018)
- [105] The Guardian: IT meltdown has cost British Airways £80m so far, says Willie Walsh. <https://www.theguardian.com/business/2017/jun/15/it-meltdown-cost-british-airlines-80m-so-far-willie-walsh-iag> (2017)
- [106] The KLEE Community: The KLEE Project. <https://github.com/klee/klee> (2019)
- [107] The New York Times: Facebook security breach exposes accounts of 50 million users. <https://www.nytimes.com/2018/09/28/technology/facebook-hack-data-breach.html> (2018)
- [108] Tian, Y., Lawall, J.L., Lo, D.: Identifying linux bug fixing patches. In: 34th International Conference on Software Engineering, ICSE 2012, Zurich, Switzerland. pp. 386–396 (2012)
- [109] Tianocore: EDK II. <https://github.com/tianocore/edk2>

- [110] Tianocore: Tianocore. <http://www.tianocore.org/>
- [111] Torvalds, L.: Initial commit of linux kernel's git repository. <https://git.io/fjGug> (2005)
- [112] Wang, C., Jiang, Y., Zhao, X., Song, X., Gu, M., Sun, J.: Weak-assert: a weakness-oriented assertion recommendation toolkit for program analysis. In: Proceedings of the 40th International Conference on Software Engineering, Companion Proceedings, Gothenburg, Sweden (2018)
- [113] Wang, X., Sun, J., Chen, Z., Zhang, P., Wang, J., Lin, Y.: Towards optimal concolic testing. In: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden. pp. 291–302 (2018)
- [114] Wong, E., Zhang, L., Wang, S., Liu, T., Tan, L.: Dase: Document-assisted symbolic execution for improving automated software testing. In: Proceedings of the 37th International Conference on Software Engineering - Volume 1. pp. 620–631. ICSE '15, IEEE Press, Piscataway, NJ, USA (2015)
- [115] Xu, H., Zhou, Y., Kang, Y., Lyu, M.R.: 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. pp. 181–188 (2017)
- [116] Xu, M., Qian, C., Lu, K., Backes, M., Kim, T.: Precise and scalable detection of double-fetch bugs in OS kernels. In: 2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, San Francisco, California, USA. pp. 661–678 (2018)

- [117] Yun, I., Lee, S., Xu, M., Jang, Y., Kim, T.: QSYM : A practical concolic execution engine tailored for hybrid fuzzing. In: 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA. pp. 745–761 (2018)
- [118] Zaddach, J., Bruno, L., Francillon, A., Balzarotti, D.: AVATAR: A framework to support dynamic security analysis of embedded systems’ firmwares. In: 21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA (2014)
- [119] Zakharov, I.S., Mandrykin, M.U., Mutilin, V.S., Novikov, E.M., Petrenko, A.K., Khoroshilov, A.V.: Configurable toolset for static verification of operating systems kernel modules. *Program. Comput. Softw.* **41**(1), 49–64 (2015)
- [120] Zheng, H., Li, D., Liang, B., Zeng, X., Zheng, W., Deng, Y., Lam, W., Yang, W., Xie, T.: Automated test input generation for android: Towards getting there in an industrial case. In: Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track. pp. 253–262. ICSE-SEIP ’17, IEEE Press, Piscataway, NJ, USA (2017)