Saarland University
Department of Computer Science

Low-Level Software Memory Safety Analysis: Attack and Defense

Dissertation
zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Fakultät für Mathematik und Informatik
der Universität des Saarlandes

von
Jianqiang Wang

Saarbrücken, 2025

Tag des Kolloquiums: 04. Aug 2025

Dekan: Prof. Dr. Roland Speicher

Prüfungsausschuss:

Vorsitzender:	Prof. Dr. Reinhard Wilhelm
Berichterstattende:	Prof. Dr. Thorsten Holz
	Prof. Dr. Andreas Zeller
	Prof. Dr. Mathias Payer
Akademischer Mitarbeiter:	Dr. Hanwei Zhang

Zusammenfassung

Low-Level-Software ist, wie der Name schon sagt, eine Art von Software, die enger mit der Hardware zusammenarbeitet als normale Anwendungen. Sie übernimmt in der Regel Aufgaben wie die Initialisierung der Hardware, die Einrichtung der Ausführungsumgebung und die direkte Interaktion mit den Hardware-Komponenten. Aufgrund der entscheidenden Rolle, die Low-Level-Software spielt, benötigt und erlangt sie naturgemäß höhere Hardware-Privilegien als andere Software, was sie zu einem vielversprechenden Ziel für Systemangreifer macht. Um die Hardware bequem zu steuern, verwenden Low-Level-Softwareentwickler häufig speicherunsichere Programmiersprachen wie C und C++. Einerseits erleichtert die direkte Speicherverwaltung den Entwicklungsprozess, andererseits macht sie die Low-Level-Software jedoch anfällig für Speicherverletzungs-Schwachstellen.

Ausgehend von den Ursachen und Folgen von Speicherverletzungs-Schwachstellen demonstriert diese Arbeit, dass Speicherunsicherheit eine Hauptbedrohung für die Sicherheit von Low-Level-Software darstellt. Nachfolgend werden Speicherunsicherheitsprobleme von Low-Level-Software aus offensiver und defensiver Perspektive analysiert und diskutiert. Im offensiven Szenario geht die Arbeit auf die Probleme ein, die die Erkennung von Speicherverletzungs-Schwachstellen noch behindern. Insbesondere wird eine Technik namens Fuzz-Testing angewandt, bei der zufällige Eingaben in die Software eingespeist werden. Dazu wurden maßgeschneiderte Fuzzer entworfen und implementiert, um Speicherverletzungs-Schwachstellen in Low-Level-Software, Firmware eingebetteter Systeme und Bootloadern zu finden. Die Fuzzer zielen darauf ab, möglichst viele Teile der Low-Level-Software zu erkunden und möglichst viele Abstürze auszulösen. In den Experimenten wurden 46 zuvor unbekannte Schwachstellen gefunden, denen 11 CVEs (Schwachstellen-Identifikationsnummern) zugewiesen wurden. Aus einer defensiven Perspektive ist die Bereitstellung eines einfach zu verwendenden Frameworks mit umfangreichen Funktionen für High-Level-Anwendungen bei gleichzeitiger Gewährleistung der Speicher-Sicherheitsgarantien ein wesentlicher Aspekt. Um dieses Problem zu lösen, wurde im Rahmen dieser Arbeit ein Trusted Execution Environment-Framework für die RISC-V-Architektur entworfen und implementiert, das die RISC-V-Hypervisor-Erweiterung und eine bestehende Hardware-Speicherisolationstechnik nutzt. Das Framework bietet vollständige Rückwärtskompatibilität und sicheres I/O, was bedeutet, dass eine unveränderte virtuelle Maschine direkt auf dem Framework ausgeführt werden kann und von einer transparenten sicheren I/O-Übertragung profitiert. Abschließend wird gezeigt, dass das Framework eine ähnliche Leistung wie die AMD-SEV-Erweiterung erreichte und nur minimale Zusatzkosten im Vergleich zur nativen Ausführung verursachte.

Abstract

Low-level software, as the name suggests, is the kind of software that runs more closely with the hardware than normal applications. They usually undertake the responsibilities of initializing the hardware, setting up the execution environment, and directly interacting with the hardware functionalities. Due to the intrinsic features of the key role played by low-level software, it naturally requires and gains higher hardware privilege than other software to run, making it a promising target for system attackers. To conveniently manipulate the hardware, low-level software developers commonly adopt memory-unsafe programming languages such as C and C++. On the one hand, the direct memory access programming language makes the development process easier, on the other hand, however, it makes the low-level software prone to be compromised by memory corruption vulnerabilities.

In this thesis, we start with the causes and consequences of memory corruption vulnerability and then showcase that memory safety issue is a main threat to low-level software security. We discuss and analyze low-level software memory safety issues in terms of both their attack and defense sides. From the perspective of the attack scenario, we tackle the problems that still hinder the detection of memory corruption vulnerability detection. In particular, feeding random inputs to the software—a technique called fuzz testing—is used. Specifically, We designed and implemented tailored fuzzers to find the memory corruption vulnerabilities in the low-level software, embedded system firmware, and bootloader. For each, the fuzzers aim to explore as many parts of the low-level software as well as trigger more crashes as possible. In our experiments, 46 previously unknown vulnerabilities were found, and 11 CVEs were assigned to the findings. From the perspective of defense, providing an easy-to-use framework with rich features for high-level applications while maintaining the memory safety guarantees is the essential part. To solve this problem, we designed and implemented a trusted execution environment framework for RISC-V architecture by utilizing RISC-V hypervisor extension and an existing hardware memory isolation technique. The framework provides full backward compatibility and secure IO, which means an unmodified virtual machine can run directly on top of the framework and benefit from transparent, secure IO transmission. Our experiment showed that the framework achieved similar performance with AMD SEV extension and trivial overhead compared with native running.

Background of This Thesis

This thesis is based on the papers mentioned in the following. The author contributed to all papers as the main author.

- A Comprehensive Memory Safety Analysis of Bootloaders (NDSS'25)
- AidFuzzer: Adaptive Interrupt-Driven Firmware Fuzzing via Run-Time State Recognition (Usenix security'24)
- VirTEE: A Full Backward-Compatible TEE with Native Live Migration and Secure I/O (DAC'22)

Further Contributions of the Author

The author also fully or partially contributed to the following papers.

- SEnFuzzer: Detecting SGX Memory Corruption via Information Feedback and Tailored Interface Analysis (RAID'23)
- RiscyROP: Automated Return-Oriented Programming Attacks on RISC-V and ARM64 (RAID'22)
- NLP-EYE: Detecting Memory Corruptions via Semantic-Aware Memory Operation Function Identification (RAID'19)

Acknowledgments

I want to thank everyone who helped me during my pursuit of the PhD. My kind and friendly lab mates, paper co-authors, and friends' companions motivated me to work and live in a cozy environment. I want to give special thanks to my supervisor, Thorsten Holz, whose humble, gentle, and kind personality impressed me when I started working in his group. I enjoyed the atmosphere he built for this group and I really appreciate being part of it.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Low-Level Software and Its Risks

Modern computer systems provide a wide range of methods to protect end-users from malicious attackers. Hardware mechanisms or extended features directly or indirectly enforce those protections. As the name suggests, *low-level software* is used by developers to drive the mechanisms and features, which thus runs closely to the hardware. A typical example is the permission control system for users and groups supported by all Linux kernels. Although fully implemented by software, the permission control system implementation in an x86 architecture platform is based on a processor privilege isolation mechanism where the kernel space and user space code run on different privilege levels. The kernel, which functions as low-level software, implements permission control and governs the execution of user-space applications. It is unavoidable for an attacker who resides in user space and aims to bypass the permission control to follow the processor-enforced access control, which states that the user space code cannot directly read or write kernel space memory. Software vulnerability, especially memory corruption in low-level software, plays an important role in the system attack scenario to help attackers achieve their goals.

Memory corruption vulnerabilities can lead to severe effects when exploited properly. For example, the Heartbleed out-of-bound read vulnerability (CVE-2014-0160) in OpenSSL reported in 2014 can leak secret data from a server. A write-to-read-only-memory vulnerability (DirtyCOW) in the Linux kernel (CVE-2016-5195) makes use of a race condition to escalate from user to root privileges. A report by Microsoft demonstrated that around 70 percent of vulnerabilities in their products are memory safety issues [150].

Unfortunately, to gain better performance and manipulate the hardware conveniently, memory-unsafe programming languages such as C and C++ are commonly adopted by low-level software developers. On the one hand, the choice makes the development process easier, however, on the other hand, C and C++ do not automatically prohibit memory corruption, leaving a large number of vulnerabilities in the software. Nowadays, memory-unsafe ones are still the mainstream programming languages [53] in low-level software such as operating system kernels, hypervisors, and bootloaders.

1.2 Memory Safety Issues in Low-Level Software

Low-level software is usually responsible for setting up the environment and communicating with the hardware. Naturally, it requires a higher hardware privilege or special access permission to execute, making it an attractive target for attackers. The reasons are straightforward: the earlier or the higher privilege malicious code can get to execute, the more severe consequences the attack can cause. For example, compromising a user-space application might cause a segmentation fault or data leakage within a single process. However, a kernel memory corruption vulnerability can lead to root privilege escalation and affect all system processes. Even worse, if a memory safety issue occurs in the BIOS firmware, the malicious code might remain after reinstalling the operating system.

Attack and defense against memory safety issues in low-level software has been a

cat-and-mouse game for a long time. The battlefields spread across various platforms, hardware mechanisms, processor features, and low-level software. We briefly introduce the current situation based on the rough loading time of the low-level software.

BIOS. Legacy BIOS firmware was replaced by modern UEFI standard firmware, which the researchers focus on nowadays. Finding memory corruption vulnerabilities in system management mode code is a popular topic. Prior works proposed symbolic execution [18], static analysis [237, 188, 51, 52], fuzzing [236, 235], and a mix of them [99] to detect memory safety issues in system management mode interrupt handlers. Memory corruption vulnerabilities and exploitations beyond the system management mode, such as [72, 142, 62, 230, 194] in UEFI firmware were found. Despite the severe effects, software protections such as CFI enforcement and obfuscation against memory corruption vulnerabilities in UEFI firmware are still missing. EDK II [210], the modern, feature-rich, cross-platform firmware development environment provides basic address sanitizer features for developers to uncover the issues in the early stage. However, it is far from enough to prohibit them from happening. Surve et al. systematically analyzed UEFI security [198]. Besides memory safety issues, they also pinpoint other attacks, such as credential theft and UEFI firmware image tampering. UEFI memory safety is far from well-investigated due to its complexity. Since static analysis cannot be applied to large-scale software because of its performance bottleneck, dynamic analysis is the preferred method. Modern dynamic analysis usually requires the execution environment to gain run-time data. However, the UEFI firmware image is tightly coupled with the processor and platform. The state-of-the-art simulator Qemu [20] supports hundreds of CPU models and platforms—which seems a lot, but still does not satisfy the requirements compared with tens of thousands of real-world CPU models and millions of peripherals. Without a simulated environment, the loss of run-time data, such as function pointers and port register value, hinders the deep analysis of the firmware image. It is still unclear how to simulate the full UEFI firmware run-time environment and perform a memory safety analysis on it.

Bootloader. With the legacy BIOS, the bootloader starts from the master boot record (MBR). However, it functions as a UEFI application in the modern epoch. The bootloader is supposed to be the bridge between the firmware and the operating system. The first edition of the popular Linux bootloader GRUB [87] was released in 1995. Originally, it only supported less than ten types of file systems. However, nowadays, bootloaders gradually provide richer features for end-users, such as customized GUI, various file systems, and user authentication. As the code base grows, it inevitably exposes more memory corruption vulnerabilities than before. There are several scattered works targeting bootloader vulnerability detection. Axtens [60] and Starke [157] have proposed fuzzing techniques for GRUB [87] and Das U-Boot [63]. Mobile device bootloaders are usually closed-source and hard to analyze. BootStomp [174] and Roee [93] have only analyzed bootloaders for Android systems, with a specific focus: BootStomp analyzed storage data controlled by the attacker that could compromise the bootloader, while Roee focused exclusively on command line inputs. Despite the efforts made by researchers on the bootloader memory safety analysis, a comprehensive analysis of bootloaders is still missing, which is what we need to address in this thesis.

Hypervisor. During the last five years, hypervisor security has been a hot topic.

After the first hypervisor fuzzing work VDF [94] that targets a single Qemu simulated device was introduced, many subsequent types of research such as Truman [137], HYPERPILL [30], Morphuzz [28], HYPER-CUBE [182], V-shuttle [163], MundoFuzz [154], HyperFuzzer [84] were proposed. Most of them focus on virtual devices, which comprise the majority of the hypervisor code base. The inputs from the virtual machine to the simulated devices consist of two main parts: the port IO and the DMA buffer. The fuzzing works utilize the processor features that the device registers accessing traps the execution into the hypervisor to inject the fuzz input into the hypervisor. Inferring the address of the DMA buffer is tackled case by case. For example, V-shuttle takes advantage of hooking the Qemu function *pci_dma_read*. Fuzzing a closed-source hypervisor is tough due to its complex run-time environment. HYPERPILL identifies the universal and general hypervisor-processor interaction behavior, takes a snapshot, and performs fuzzing on the snapshot to overcome the issue. Embedded system hypervisors such as QNX [22] have not been well studied yet. Commercial embedded systems may not allow the user to inspect device states such as physical memory, making it impossible to take a snapshot of the device. Besides, the embedded system peripherals are highly customized, and no public documentation is available. Simulating such a system is not a reasonable way to perform fuzzing. When conducting on-device fuzzing, dealing with other problems, such as coverage feedback and crash reproduction, needs to be considered. The majority of the memory corruption vulnerabilities exist in virtual device implementations. As the virtual devices are implemented purely in software, protection against hypervisor memory corruption vulnerabilities should rely on general protections such as CFI, and ASLR instead of specific methods. The Virtio [158] specification does not mention the security design or implementation that must be followed by developers. Applying mature and general protections such as enforced CFI to virtual devices might be viable in the situation.

OS Kernel. The operating system (OS) kernel has always been the main arena of memory safety attack and defense. The kernel exposes interfaces for user space applications, interacts with devices, and runs multiple threads, making it a complex low-level software. Typical kernel-specific memory corruption vulnerabilities such as double-fetch [222, 233, 223, 184] and race conditions [123, 13, 65, 71, 219, 220, 26, 110] continue to threaten the kernel security. A large amount of kernel fuzzing works target malicious input from user space applications [89, 116, 196, 234, 29, 162, 41, 197, 109, 82, 238, 243], devices [195, 166, 190, 231], or user-specified origins [183, 181]. In addition, static analysis [222, 135, 139, 85], and a mix of them [202, 138, 44] were proposed as well. They focus on different parts of the kernel, such as file systems, device drivers, error handlers, and interrupt service routines, tackling various problems such as system call sequence ordering, input seed generation, device simulation, and system call parameter format reconstruction. Due to the complexity of the kernel, memory safety issues in such an environment are hard to catch and hide in the corner. Static analysis is unsuitable for such large-scale software, and setting up a real or simulated environment is also difficult. The multi-dimensional inputs from user space and device drivers make the problems even harder. Common memory protections have already been integrated into the kernel source code base [207]. Besides, plenty of research protections have been designed for the kernel [239]. However, we believe that there is no silver bullet to handle

the problem, and memory corruption vulnerabilities will continue to exist, and new methods will come up by researchers as well.

Embedded System Firmware. There are two main types of embedded system firmware: POSIX-compatible Unix-like systems and Real-Time Operating Systems (RTOS). A Real-Time Operating System (RTOS) is used in low-power embedded devices because it is lightweight, allows for direct peripheral manipulation, and requires little memory. The Unix-like system provides a fully featured kernel and user space applications. Emulating the full or part of the system in a virtual environment poses challenges. Researchers proposed analyzing Unix-like systems by fuzzing them in a partially or fully simulated environment (FIRM-AFL [246], Firmadyne [34], Firmae [117], Greenhouse [203], EQUAFL [247], FIRMWIRE [96], FirmSolo [3]), fuzzing them on-device (IotFuzzer [38]), static analysis (Feng et al. [76], DTaint [42], Firmxray [226], Lara [244], PASAN [118]), LLM-assisted analysis (mGPTFuzz [136]), and symbolic execution (Firmusb [95]). Analyzing RTOS firmware often involves a technique called *rehosting* as the low-power devices are not designed for high-throughput executions. Researchers proposed specific rehosting-based fuzzing for RTOS firmware such as P2IM [74], HALucinator [46], DICE [144], D-Box [143], SHiFT [145], CO3 [130], Fuzzware [177], Hoedur [178], Safirefuzz [185], MultiFuzz [43] PERRY [124], Sfuzz [39]. Detecting embedded system firmware memory corruption is challenging. The complex closed-source peripheral standard makes the situation even worse. An example is the interrupt-triggering problem. If the peripheral interrupt is not triggered properly when rehosting an RTOS firmware, the fuzzing process might be hindered. In this thesis, we solve this problem by proposing an adaptive interrupt-driven firmware fuzzing technique. Moreover, the question of how to analyze the embedded system's high-level protocol logic, such as Bluetooth and TCP/IP or application layer protocol stacks, has not been well addressed yet.

TEE Application. For the Trusted Execution Environment (TEE), researchers focus more on the design and implementation of a robust TEE system. Penglai [75] and Keystone [121] provided their solutions to build a new TEE system based on the existing physical memory isolation mechanism, such as RISC-V PMP. LibOS based solutions such as Graphene [211], Occlum [189], SGX-LKL-OE [170], Fortanix [125], SCONE [6] try to port unmodified legacy applications to Intel SGX. Chen et al. [36, 35, 107, 37] proposed potential attestation issues and their countermeasures for TEE applications. However, the memory safety issue is also a non-negligible aspect. Lee [122], Bulck [216], Checkoway [33], and Biondo [21] mentioned that memory corruption vulnerabilities and the exploitation methods do not disappear within the TEE environment. Even worse, SmashEx [58] showed that the majority of the officially provided TEE SDKs are all vulnerable to a kind of exception-handling-driven attack. To detect memory corruption vulnerabilities in TEE applications, model checking [14], symbolic execution [115, 47] and fuzzing [59, 48, 240, 224, 40] were introduced by researchers. To protect the TEE applications from memory corruption, Poster [67] and Wang et al. [221] proposed to use the memory-safe Rust language to rewrite the enclave code. BesFS [192] presented an Iago-safe API specification to defend against Iago attacks. MPTEE [245] used bound check to add flexible page permission to enclave memory. SGXPecial [152] extended the Edger8r tool of the Intel SGX SDK to generate more restricted ECALL/OCALL

interfaces. Intel has abandoned SGX and will not include it in new processors, which indicates the end of application-based TEE systems. The future of mainstream TEEs will be dominated by virtual machine-integrated design systems. Memory safety issues will continue to exist in the TEE applications as long as they communicate with normal applications. However, analyzing virtual machine-integrated TEE applications leaves more challenges, such as a hard-to-setup environment, to researchers. In this thesis, we propose a fully backward-compatible secure TEE framework based on a RISC-V hypervisor extension. We address the TEE memory safety issues by combining an unmodified OS kernel and a lightweight hypervisor shim that provides the necessary interfaces for the OS.

1.3 Research Focuses

In this thesis, we focus on three typical low-level software, bootloader, embedded system firmware, and TEE application, to analyze their memory safety issues in terms of both attack and defense.

1. **Bootloader.** A bootloader is loaded into memory and gets executed at the early stage during the system boot process. During the secure boot process, it undertakes the responsibility of verifying the integrity of the next component, usually the operating system kernel image. If the kernel is not signed with a valid key, then it rejects loading. Due to the important role played by the bootloader, it is supposed to be designed and implemented securely. However, during the last decades, the bootloader has provided the end-users with rich features, and thus, its code base is getting larger and larger. However, a comprehensive memory safety analysis against the bootloader and a framework for detecting bootloader memory corruption vulnerabilities is still missing. In this thesis, we propose a comprehensive memory safety analysis against nine modern bootloaders. We thoroughly analyze the attack surfaces and how an attacker could exploit the vulnerabilities found to gain system privilege. Based on our observation, we design and implement a universal fuzzing framework to detect memory corruption vulnerabilities in various bootloaders. The framework is compatible with the majority of modern bootloaders despite their implementation details. We found 39 new vulnerabilities and five CVEs were assigned to our findings.

2. **Embedded System Firmware.** RTOS is widely used in embedded systems. Fuzzing has proven to be an efficient and effective method to detect RTOS memory corruption vulnerabilities. However, low-powered and slow-speed embedded devices are not designed for fuzzing due to their small throughput. Therefore, prior fuzzing works targeting RTOS run it in a simulated environment—a technique called re-hosting. The re-hosting environment simulates the processor, memory, and necessary peripherals to support the RTOS image execution. However, prior works rarely considered the peripheral interrupt triggering problem, which is a fundamental part of RTOS re-hosting and fuzzing. Without a proper interrupt-triggering mechanism, the RTOS may crash or get stuck in a very early stage. The fuzzing effectiveness can also be hindered due to improper interrupt triggering.

We observe that to solve the interrupt-triggering problem, identifying the RTOS run-time state is the key point. The correct execution of RTOS demands an interrupt only when it is in a specific run-time state. Based on this insight, we designed and implemented a fuzzing framework, Aidfuzzer, to tackle the interrupt-triggering problem in RTOS fuzzing. Aidfuzzer identifies the RTOS runt-time state dynamically, and triggers interrupts when necessary. Our experiments show that Aidfuzzer outperforms the state-of-the-art works in coverage and vulnerability detection. Our experiments found eight new vulnerabilities, and five CVEs were assigned.

3. **TEE Design.** Existing application-based TEE schemes such as SGX do not provide full backward compatibility, which means developers are required to design and implement the new application from scratch. Virtual machine-based TEE schemes such as CCA contain large code bases that expose more attack surfaces. In this thesis, we propose VirTEE, a full backward-compatible TEE that utilizes a RISC-V hypervisor extension and prior hardware memory isolation work to address the existing TEE shortcomings. VirTEE puts a hypervisor shim in the TEE, which exposes necessary TEE functionalities such as attestation and transparent, secure IO to the virtual machine so that an unmodified operating system can run directly in it. In the meantime, we keep the hypervisor shim as small as possible, making it less vulnerable to memory corruption vulnerabilities. In our experiments, VirTEE achieves comparable performance with the existing virtual machine-based TEE scheme.

The three targets all choose the memory-unsafe programming language C. Our analysis reveals that nowadays, memory corruption vulnerability is still a main threat to low-level software security. When designing and implementing low-level software, developers are supposed to pay more attention to it.

1.4 Thesis Roadmap

In this thesis, we present the overall high-level technical background in Chapter 2. In Chapter 3 4 5, we introduce our memory safety analysis details against the three targets. Note that the detailed technical background for each specific target is included in the specific chapter for clearance. In chapter 6, we present the related works about low-level memory safety analysis and the security mechanism that takes advantage of hardware features. In Chapter7, we conclude the thesis and propose our future works.

2
Technical Background

In this chapter, we present the hardware features provided by the processors that are closely related to the works in this thesis. In particular, the processor features TEE, hypervisor, Intel PT, Intel VT-x, and secure boot are introduced. Embedded system basic processor mechanisms such as interrupt and Memory-mapped I/O (MMIO) are related to the embedded firmware fuzzing and thus also included in the chapter. Further, the type, causes, and consequences of memory corruption vulnerability are demonstrated. Finally, a technique used to detect software memory corruption called fuzz testing (fuzzing) is presented.

2.1 Hardware Features

In this section, we introduce hardware features provided by the processors. Note that we only illustrate the general concepts of the feature. The specific feature details are presented in each target analysis chapter.

2.1.1 Trusted Execution Environment

A Trusted Execution Environment (TEE) is a secure region of a processor that ensures sensitive data and operations remain protected from unauthorized access, even if the rest of the system is compromised. Unlike traditional computing environments where the entire OS and applications share the same memory and execution resources, TEEs segregate secure operations into separate environments. This segregation protects sensitive operations from privileged software like the OS and hypervisor. TEE operates as an isolated execution area, providing confidentiality, integrity, and authentication for critical code and data. The processor vendors propose such a feature to address the threat model where the end-user application runs in an untrusted environment such as a cloud server. TEE guarantees that the end-user data is protected against a malicious attacker with the highest privilege who has full control over the system. The key characters [54] of TEE are:

- **Isolation.** TEE ensures that operations inside the TEE are separate from the rest of the system.
- **Confidentiality.** Prevents unauthorized access to data within the TEE, even with the highest privilege.
- **Integrity.** TEE ensures the correctness and protection of both code and data during execution, which means memory-write from non-TEE is automatically discarded by the processor.

There are two types of TEE design schemes: application-based TEE and virtual machine-based TEE.

Application-Based TEE. A typical application-based TEE example is Intel SGX [103]. The secure region in TEE is called the enclave. Developers wrap the code that needs to run in the trusted environment as a library. The whole SGX application runs as a single process. As shown in Figure 2.1, the enclave library runs in enclave

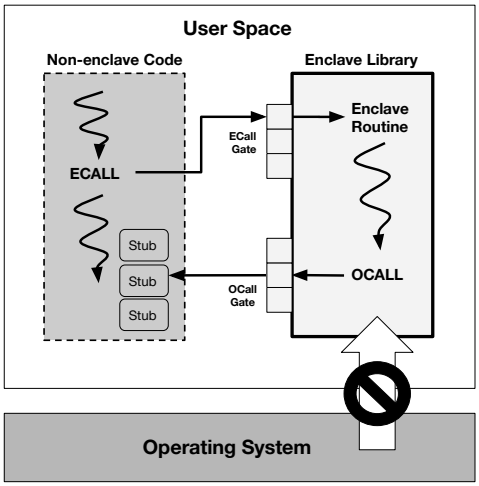

Figure 2.1: SGX design overview

memory, which is located in the user space. Compared with the conventional processor memory access permission that privileged OS can read and write any physical memory, the enclave memory is only accessible to the enclave code. Naturally, the processor does not allow a direct jump from the non-enclave to the enclave and vice versa. Especially, SGX provides a set of instructions such as *EENTER* and *EEXIT* to perform the controlled context switch between the two regions.

Similar to system calls, to use the functionalities provided by the enclave library, non-enclave code performs a special context switch from non-enclave to enclave (i.e., ECALL in Figure 2.1). The transition is confined and checked by the processor. Only when the target address is a valid entry located in the enclave metadata the transition permission will be granted.

The SGX design prevents access from the privileged OS to the enclave. On the other hand, the enclave code cannot directly invoke the system calls. To have access to OS functionalities, the enclave code has to first transit to the non-enclave (i.e., OCALL in Figure 2.1). Some predefined functions that are regarded as stubs in the non-enclave memory are responsible for forwarding requests to the OS. When the system calls return, the enclave code reads the return value and buffers from the stubs to the enclave memory.

Application-based TEE is lightweight but does not provide compatibility with the existing libraries. Developers start from scratch with the SDK provided by Intel. However, it has been proven that application-based TEE cannot be accepted by the industry.

Virtual Machine-Based TEE. Putting the whole virtual machine in the TEE eliminates the manual effort of developing the TEE applications from scratch, and it has been a popular TEE scheme nowadays. Intel TDX [104], ARM Trust Zone [5], and AMD SEV [134] are well-known virtual machine-based schemes. As shown in Figure 2.2, ARM TrustZone [5, 156] split the system into two parts: a normal world and a secure world. A

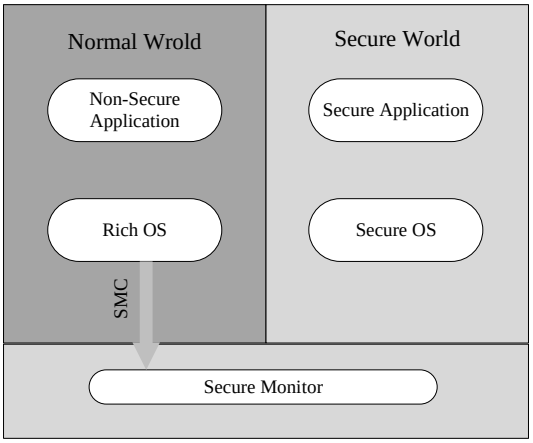

Figure 2.2: TrustZone on ARM Cortex-A

rich OS serves the normal world, while a lightweight, secure OS serves the secure world. The two worlds communicate via a secure monitor, which acts like a context switch. Normal world software can only use the services provided by the secure world by using a hardware interrupt, an external abort signal, or the software instruction *SMC*. The hardware interrupts and aborts are asynchronous and only support a full-world switch, while SMC also supports message passing without a complete changeover. Secure world software can use these methods or write directly to the CPSR. Secure monitor code executes with interrupts disabled due to volatility concerns.

Virtual machine-based TEE provides backward compatibility. Developers only need to focus on the TEE application logic implementation instead of worrying about the library dependency. It has proven to be the TEE scheme accepted by most processor vendors and communities.

2.1.2 Hypervisor

A hypervisor (also known as a Virtual Machine Monitor or VMM) is a software or hardware layer that creates and manages virtual machines (VMs) by abstracting the underlying physical hardware. It enables multiple operating systems to run concurrently on a single physical machine, sharing resources like CPU, memory, storage, and peripherals. Hypervisors are a critical component in virtualization technology and play a significant role in cloud computing, development environments, and server consolidation. There are two types of hypervisors. Type one 2.3 hypervisors such as Microsoft Hyper-V [149] and VMware ESXi [218] run directly on the hardware and manage guest operating systems. They directly manage the hardware resources and the communication between virtual machines. Type two 2.4 hypervisors such as VMware Workstation [217] and Oracle VirtualBox [159] run inside a booted operating system. They rely on the host OS to manage the hardware resources.

Second Level Page Table. The Second Level Page Table is a hardware-assisted memory virtualization feature used by hypervisors to manage guest virtual memory efficiently. In a virtualized environment, a guest operating system assumes it has

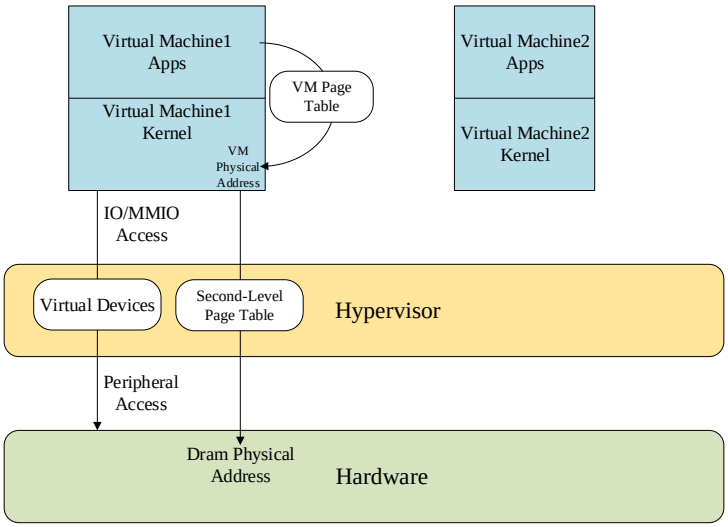

Figure 2.3: Type-1 hypervisor

direct access to the hardware memory. However, the hypervisor must manage the translation of the guest's memory access to the actual physical memory of the host machine. The second-level page table simplifies this process by adding another level of page tables, reducing the need for software-based translation. The guest's virtual address is translated into the guest's physical address via the guest page table, which is further translated into the host's physical address via the second-level page table. The translation process happens automatically in the memory management unit of the processor, making the translation overhead negligible. As reported by kAFL [183], with Intel-VTx that enables second-level page table acceleration, virtualized applications can achieve 45 times faster than QEMU software memory translation.

Device Simulation. Device simulation refers to the process of emulating or virtualizing hardware devices within a virtualized environment, allowing guest operating systems and applications to interact with virtual devices as if they were physical hardware. The hypervisor is responsible for simulating these devices and managing their interactions with the host system's actual hardware. The device access via the IO port or MMIO can be trapped into the hypervisor, which allows the hypervisor to intercept the peripheral-OS communication. A virtual machine running in a para-virtualized environment is aware of the hypervisor and communicates with it via low overhead virtio devices.

2.1.3 Intel PT and VT-x

Intel PT [102] and Intel VT-x [106] technologies are two Intel processor features that enable program trace and hypervisor.

Intel PT. Intel PT (Processor Trace) is a hardware feature for fine-grained tracing of program execution. It enables developers and security researchers to capture detailed information about how software executes on the CPU. Intel PT is widely used in

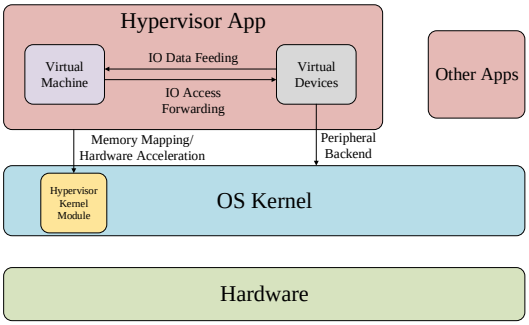

Figure 2.4: Type-2 hypervisor

debugging, performance analysis, and reverse engineering. It can be used to capture control flow changes (e.g., branches, jumps, calls, and returns) in the program execution. Due to its low overhead and selective tracing mode, coverage feedback-driven fuzzing can benefit from this feature, which we will discuss later in the chapter.

Intel VT-x. Intel VT (Virtualization Technology) is a suite of hardware-assisted virtualization features that enable the efficient execution of virtual machines. It reduces the performance overhead of traditional software-only virtualization by offloading key tasks to the CPU. It can virtualize CPU, memory, IO, and interrupt resources. Intel processors provide *VMEXIT* and *VMETER* instructions to switch into and out of a virtual environment. KVM relies on this feature in an Intel processor to create a virtual machine environment.

2.1.4 BIOS and Secure Boot

BIOS (Basic Input/Output System) is firmware embedded in a computer's motherboard (usually in the SPI flash memory in an IBM-compatible computer) that initializes hardware during the booting process and provides runtime services for operating systems and applications. It is the first code executed when a PC is powered on and plays a key role in the startup sequence. The legacy BIOS does the Power-On Self Test (POST), initializes the hardware, invokes the bootloader, and then simply exits. Modern predecessor BIOS standard UEFI provides more features than legacy BIOS. After loading the bootloader, UEFI remains in the memory and provides runtime service to the bootloader and the operating system.

Secure boot [105] is a security feature implemented in UEFI firmware to ensure that only trusted software is loaded during the boot process. It protects against malicious software, such as rootkits or bootkits, that could compromise the system at startup. Secure boot functions as a chain of trust; the former component verifies the next one and decides whether or not to load the next component.

2.2 Embedded System

An embedded system is a specialized computing system designed to perform dedicated functions or tasks. Unlike general-purpose computers, embedded systems are optimized

for specific applications, often with strict constraints on performance, power consumption, and size. Real-Time Operating System (RTOS) is a kind of lightweight OS that usually contains a simplified task scheduler, peripheral access wrapper, and necessary standard library. RTOS is designed for embedded systems due to its small code base, which can fit into a small physical memory.

2.2.1 Interrupt

An interrupt is a mechanism in computing that temporarily halts the execution of the current program to address an event or request that requires immediate attention. Once the interrupt is handled, the system resumes the execution of the interrupted program. Interrupts are essential for responsive and efficient system behavior, allowing the processor to respond promptly to asynchronous events, such as hardware signals or software exceptions. In an embedded system, interrupts are handled in a way that minimizes latency while maintaining deterministic behavior.

2.2.2 Memory-Mapped I/O

In a reduced instruction set processor, peripherals are accessed by uniformed memory access called Memory-Mapped I/O (MMIO). In MMIO, hardware devices such as peripheral controllers are mapped into the same address space as the system's RAM. This allows the CPU to interact with hardware devices using standard memory access instructions. When a large amount of data needs to be transferred from peripherals to the DRAM, Direct Memory Access (DMA) is used. When configured, DMA allows peripherals to use the bus for data transferring without CPU interception.

2.3 Memory Corruption Vulnerability

Szekeres et al. [200] mentioned that memory corruption vulnerability has existed for over 30 years. Memory corruption vulnerability begins with a memory error. Depending on the type of the error, de-referencing an out-of-bound pointer is called spatial error, while de-referencing a dangling pointer is called temporal error. The naming methodology is straightforward. An out-of-bound pointer does not point to the intended location, while the dangling pointer once points to a valid memory however, it becomes invalid when de-referencing. A typical spatial error is stack overflow, by overwriting the return address of the current function hijacks the control flow execution. Use-after-free is a common temporal memory error, which usually refers to a heap memory being used after the memory is deallocated. Attackers exploit memory corruption vulnerabilities to change the intended behavior of the target.

2.3.1 Exploitation

The attacker's goal is to change the application's normal execution behavior. The main method to achieve their goal is to alter the application control flow but not confined to it.

Shellcode. Shellcode is a sequence of machine instructions that is crafted to execute a specific payload on a target system, often as part of an exploit, giving attackers control of a compromised system. A typical shell code can be injected by the attack to the target application to swap a new shell process. In certain circumstances, shellcode might be detected by intrusion detection mechanisms, some interesting works [141] generate ASCII character shellcode, printable shellcode, or even English shellcode. However, modern shellcodes can perform a wide range of tasks beyond launching a shell, including downloading files, injecting processes, or escalating privileges. The crafting of shellcode depends on the specific platform and environment.

Code Reuse. Code reuse attacks exploit vulnerabilities in software to execute malicious actions by reusing existing code (called gadget) within the target application or system rather than injecting new code. These attacks circumvent security measures such as Data Execution Prevention (DEP), which marks certain memory regions as non-executable, preventing traditional code injection. Prominent examples of code reuse attacks include Return-to-Libc (ret2libc), Return-Oriented Programming (ROP) [169], and Jump-Oriented Programming (JOP) [24]. These techniques are widely used by attackers to compromise systems while evading detection and mitigation. Theoretically, the larger the code base is, the more gadgets the attacker could use to launch the attack.

Data Only Attack. When control flow hijacking is not possible, attackers can solely modify the application data and reuse the application logic to alter the application behavior. The attacker's goal can be achieved without modifying data that is explicitly related to control flow. For example, the attacker might exploit a buffer overwrite vulnerability to change an integer, which indicates the current user's permission to gain higher privilege than normal execution. Hu et al. proposed Data-Oriented Programming (DOP) [98] to construct expressive non-control data exploits for arbitrary x86 programs. They show that the attack can construct a Turning-complete attack in two out of nine real-world programs.

2.3.2 Mitigation

The security community has appealed to programmers to carefully consider the memory safety problems when coding. However, memory safety issues are still threatening the application execution. To enforce and eliminate the application memory safety. Several mitigation methods are proposed to prevent it from happening.

Canary. Canary protection is a security mechanism designed to detect and prevent stack-based buffer overflows. This technique is widely used in modern operating systems and compilers (e.g., GCC's Stack Smashing Protector) to protect programs from attacks that exploit stack memory to overwrite critical data such as return addresses or function pointers. A canary value is placed between the local variables and the control-sensitive data (e.g., return address, saved frame pointer) on the stack. This ensures that any overflow attempting to overwrite the return address must first modify the canary. Before a function returns, the program checks whether the canary's value has changed. If the canary is intact, the program continues execution normally otherwise, it terminates or takes predefined defensive actions.

ASLR. Address Space Layout Randomization (ASLR) is a security mechanism that

enhances program resilience against memory corruption attacks by randomizing the memory addresses of critical program components. By making the locations of the stack, heap, libraries, and other sections unpredictable, ASLR mitigates attacks that rely on a known address of the target code. Since the attacker needs to hijack the control flow of the target, ASLR makes the target address unpredictable, making the application crash immediately instead of behaving abnormally. Lu et al. proposed ASLR-Guard [133] to further augment the protection by separating the data and code region and decoding the pointer when they are translated to the other.

NX. NX (No-eXecute), also known as Data Execution Prevention (DEP), is a hardware and software-based security feature designed to prevent the execution of malicious code (intended to be data such as shell code) in non-executable memory regions (usually by putting the data in non-executable pages). By enforcing the separation of code and data, NX protection mitigates many types of memory corruption attacks, including stack-based buffer overflows and heap exploitation.

CFI. Control Flow Integrity (CFI) ensures that the program's execution adheres to a predefined, legitimate control flow. CFI effectively prevents attackers from redirecting execution to malicious or unintended code. There are many prior CFI works published in the last decades [32, 232, 114]. As claimed by Becker et al. [19], CFI has not been deployed by the majority of the binaries, making them suffer from control flow hijacking. However, most CFI rely on static compile-time data to guide the run-time control flow destination address check. It makes the CFI policy somewhat unsound and can be bypassed by carefully crafted control flow hijacking.

2.4 Fuzz Testing

Fuzz testing, or *fuzzing*, is a technique used to discover bugs in the software. The original methodology is straightforward: feed random inputs, execute the target software as an oracle, and get a report from the running result. If a crash is reported, a potential vulnerability might exist. Nowadays, fuzzing has evolved several times from black-box fuzzing (also called *dumb mode*) to the current feedback-driven fuzzing. As a popular research topic these years, various parts of fuzzing, such as input scheduling, feedback, and seed generation, have been discussed and solved.

2.4.1 Black-Box Fuzzing

During the early ages of fuzzing, fuzzing solely interacts with the target application via the command line, file, and APIs. The workflow is simple: generate the initial inputs, mutate the input, feed the input to the target application and run the target, wait for the target to exit and collect the exit result, and check if the input can cause a crash to the target. Without additional information, the method is low-efficient however, it can still detect vulnerabilities at that time.

2.4.2 Code Coverage Feedback

In 2013, AFL [147] was invented by Zalewski, which signifies a milestone in fuzzing research. The key insight of AFL is that the explored code of an input can be used as a

metric to drive the fuzzing process. When an input can trigger a path that has not been discovered previously by other inputs, it is regarded as an *interesting* input and saved in the *corpus*. The inputs in the corpus will be used as seeds to be mutated in the following rounds of fuzzing. The methodology is straightforward: a vulnerability can only be detected when the vulnerable code is executed, that said, the more code the fuzzing can explore, the more vulnerabilities a fuzzer can detect. Saving the interesting inputs in the corpus breaks the long-distance and hard-to-reach program execution point into several short-distance and easy-to-reach segments. After that, based on path coverage feedback, many works regarding seed generation [190], mutation strategies [108], input schedules [25], value predicting [9], exploration navigation [8], and a mix of them [80] were proposed.

2.4.3 Feedbacks Beyond Code Coverage

Researchers found that path coverage feedback does not always reflect how interesting the input is. For example, when fuzzing a protocol that implements a finite state machine, the state of the protocol logic is more important than how many paths an input can cover. More feedback such as state coverage [243, 224, 167, 241, 10] and data coverage [225, 78] are proposed to reflect the diversity of the quality of the input.

2.4.4 Fuzzing Modularization

Fioraldi et al. [81] proposed LibAFL, a modularized fuzzing framework that can be customized by the developers, which we believe pinpoints the development future of fuzzing. They split a fuzzer into several interconnected components: *Observer*, *Executor*, *Feedback*, *Input*, *Corpus*, *Mutator*, *Generator*, *Stage*. An observer provides the execution information, such as how many edges an input can trigger. An executor is responsible for launching the target. The executor can be as simple as a function as well as a complex enough process such as a fork server. The fuzzer treats it like an unseparated oracle to simply run the target once. The feedback determines if an input is interesting or not. Once the input is determined interesting, it is added to the corpus. Note that several feedbacks can be assigned, and the results are unified to determine the result. Feedback is usually bound with an observer, however, it is not always. For example, edge coverage feedback needs the coverage observer to provide the edge information during the execution, however, to determine if the input is interesting, the feedback also needs to maintain the previously explored paths. An input is input fed to the target application. The format of the input can be a sequence of bytes or a grammar-based string. A corpus is a collection of inputs bound with meta-data, which can be stored in memory, disk, or disk and also cached in memory. The mutator mutates one input according to its format. A bytes stream input may involve mutation strategies such as bit-flit and byte-shuffle operations. The generator generates initial inputs for fuzzing. However, researchers commonly choose to provide their initial seeds, which gain better performance. The stage reflects a single operation on an input selected from the corpus. For example, a mutational stage mutates an input from a corpus, executes the target with the mutated input several times, and checks if the input is interesting. The modularization models the fuzzing process into several components that allow the user

to customize. It clearly defines what each component should perform and provides a universal interface to extend them. We believe LibAFL will be accepted and extended by the community, and a more robust framework will come out soon.

2.4.5 Domain Specific Fuzzing

LibAFL defines a general model for fuzzing despite their detailed implementation. However, to overcome the specific problems when fuzzing the applications that are hard to harness, tailored methods are required. For example, when fuzzing the Linux kernel, even with powerful computing resources and carefully manually crafted system call specifications [89], it is still hard to detect the hard-to-trigger race condition vulnerabilities. Therefore, prior work such as ExpRace [123] proposed interrupt-triggering to tackle this problem. When fuzzing closed-source applications, it is hard to instrument the binary to collect the path coverage. Static binary rewritings such as Retrowrite [66] and MULTIVERSE [16] either work in limited environments or are not sound enough. Dynamical run-time binary writing works such as Intel Pin [151] and DynamoRIO [68] can accurately instrument the target. However, they are not coherently designed for fuzzing and, therefore, incur high-performance overhead. Besides, they are not able to fuzz system-level applications such as operating systems and hypervisor. Embedded system software designed to be executed in low-powered MCUs is not fuzzable in a native high-performance native environment. LibAFL QEMU [140] and Afl Qemu-mode were proposed to run the target in a simulated environment to solve these problems. Thanks to the Qemu instruction translation mechanism, the target instructions are translated into intermediate code and further translated into native instructions. This allows easy instrumentation during the translation process and gains a high performance by maintaining the translation result cached in memory. In addition to instruction translation, Qemu provides detailed introspection of the running target, allowing reading and modifying any internal state of the target, such as interrupt, physical memory, and peripheral register values.

3

Bootloader: Comprehensive Attack Surfaces Analysis and Generic Fuzzing Framework

3.1 Overview

As mentioned before, low-level software is executed in the early stage of system booting or requires high hardware privilege to run. A *bootloader* is a program executed during the early stage of system booting. Its purpose is to initialize a preparatory environment for loading the operating system (OS) from a storage device into memory. After powering on a computer, the firmware is loaded first and performs the necessary power-on self-test (POST). Once the firmware completes its tasks, it hands control to the next component—the bootloader. The bootloader then continues setting up the remaining environment, including the CPU, memory, and peripheral devices, for the next component, usually the OS. Obviously, the bootloader is a typical low-level software that plays an important role during the boot process, as it connects two crucial components: the firmware and the operating system.

Modern computers commonly adopt a security mechanism called *secure boot* [105] to prevent malicious or modified software from being loaded. This mechanism functions as a chain of trust: each component checks and verifies the next component to ensure it is signed by a valid digital signature. If a component fails this check, the next layer is not loaded. During the booting process, the bootloader is responsible for examining and validating the OS. A malicious or tampered OS can break this security guarantee and make the system vulnerable, hence secure boot plays a central role when building trustworthy systems. As a key part of the secure boot chain, the bootloader is responsible for verifying the operating system, loading its image into memory, and launching it. Therefore, the bootloader must be designed and implemented securely. However, bootloaders have increasingly provided more features and functionalities for end users. As the code base grows, bootloaders inevitably expose more attack surfaces. For instance, the popular Linux bootloader GRUB [87] supports more than 20 types of file systems. Additionally, it allows users to customize the background image, font, and keyboard layout, as well as downloading files from HTTP or TFTP servers. Other bootloaders, such as Das U-Boot [63] and barebox [15], face the same situation. The larger the code base becomes, the more vulnerable it gets.

In recent years, vulnerabilities, particularly memory safety violations, have been discovered in various bootloaders. Some of these vulnerabilities can lead to denial of service or even bypass secure boot protections. Roee [93] discovered a variety of vulnerabilities in Android device bootloaders. Due to the limited physical access to mobile devices, communication with the device is confined to fast boot commands [2]. However, the command line parsing logic has caused more than ten vulnerabilities in Android bootloaders. While physically accessing a Personal Computer (PC), such as plugging in an extra USB stick, is easier than accessing mobile devices, PC bootloaders face more attack surfaces. Researchers recently reported secure boot bypass vulnerabilities in bootloaders affecting hundreds of consumer and enterprise-grade x86 and ARM models from various vendors, including Intel, Acer, and Lenovo [62]. The vulnerability originates from an image-parsing library, giving the attacker full control over the system. Similarly researchers reported an HTTP implementation vulnerability [208] in shim [173]: an attacker can exploit an out-of-bound memory write to compromise the entire system. Other bootloader vulnerabilities, such as BootHole [69], CVE-2022-30790,

CVE-2022-30552, and CVE-2023-20064, continue to threaten system security.

Although bootloaders for desktop and server computers play a security-sensitive role, a comprehensive and systematic memory safety analysis of them is still missing. Existing studies either do not address such bootloaders or only focus on a single attack vector. For example, BootStomp [174] and Roee [93] have only analyzed bootloaders for mobile devices, with a specific focus: BootStomp analyzed storage data controlled by the attacker that could compromise the bootloader, while Roee focused exclusively on command line inputs. Although bootloaders for desktop and server computers expose more attack surfaces compared to bootloaders for mobile devices, they have been less scrutinized so far. Axtens [60] and Starke [157] have proposed fuzzing techniques for GRUB [87] and Das U-Boot [63]. However, their analysis was limited to command-line parsing logic. The attack surfaces of bootloaders go far beyond command-line parsing.

In this thesis, we perform the first comprehensive and systematic memory safety analysis of bootloaders and focus on the various attack surfaces that an attacker can exploit to compromise them. We start with a survey of previous bootloader vulnerabilities, which shows that attacks can primarily originate from peripheral inputs. Without the support of a rich OS environment, bootloaders must implement their standalone infrastructures, including drivers, task schedulers, timers, network protocol stacks, and more. For example, the bootloader must perform storage device reads, partition detection, file system parsing, file handler management, and file parsing to support file parsing. Since the storage data can be controlled by attackers, each layer represents a potential point of attack. Our analysis identified three types of peripheral inputs with the most attack surfaces: *storage*, *network*, and *console*.

Storage. To allow users to boot from different storage devices and file systems and to support other custom features, bootloaders implement a whole stack of file operations, including block device drivers, file system operations, and different types of file parsers. The implementation logic at each level is complex and error-prone. An attacker can easily compromise the bootloader by inserting a malicious storage device, such as a USB flash drive.

Network. To support booting over the network, such as PXE boot, bootloaders implement a complete network operations stack, including network controller drivers, the TCP/IP protocol, and application layer protocols. Some bootloaders even allow users to test the network status by sending ICMP packets. An attacker can hijack and manipulate network traffic or corrupt the server to send malicious packets to the bootloader. These network packets are processed by each layer of the network protocol stack, again increasing the attack surface.

Console. Some bootloaders provide users with an interactive interface, such as a command line console. An attacker who has physical access to the bootloader can exploit console parsing vulnerabilities by entering malicious command line strings into the console.

In addition to these three main types of peripherals, bootloaders also support other peripherals, including LED lights, power adapters, and video controllers. However, these peripherals usually do not have complex high-level parsing logic and provide less attacker-controllable data compared to the three main types discussed above.

To address the security challenges of bootloaders identified in our analysis, we

developed an automated approach to test them for potential vulnerabilities. More specifically, we develop a fuzzing framework for bootloaders, building on the proven effectiveness of fuzzing in detecting memory corruption vulnerabilities. Unfortunately, no existing solutions can be directly applied to bootloader fuzzing and we found that two main challenges need to be solved:

- Bootloaders run in a bare metal environment, which means that simple fuzzing frameworks like AFL [147] libFuzzer [132] cannot be directly deployed. Moreover, existing sanitizers cannot be used due to compatibility issues. Previous work [157] which compiles the bootloader into a native application indicates that crashes cannot be reproduced in the real bootloader because of environmental inconsistencies. Therefore, it is necessary to fuzz the bootloader in a *real* environment.

- In contrast to common user applications, bootloaders offer numerous attack surfaces. Fuzzing some of these interfaces requires dual operations. For example, when fuzzing a file system, file operations are required to trigger the parsing of the file system, while the fuzz input must be fed by intercepting the storage device data access.

To address these challenges, we simulate a virtual machine (VM) running in a hypervisor to create a real environment for the bootloader. Using a consistent operating environment helps to reduce false positives. We assume that the bootloader source code is available. Based on this, we designed a custom heap sanitizer specifically targeting bootloaders to detect heap overflow vulnerabilities. In addition, the observation that the malicious input origins are limited allows us to identify the universal interfaces and operations to intercept peripheral access and trigger device data processing. With the help of the simulated environment, the customized heap sanitizer, and the test harnesses, we can effectively fuzz the most important attack surfaces of bootloaders.

In an empirical investigation, we analyzed nine bootloaders, including the Linux standard bootloader GRUB and two well-known bootloaders for embedded systems (Das U-Boot and barebox). We spent three weeks fuzzing each bootloader and discovered 38 new vulnerabilities. Of these, 29 were confirmed or patched by the developers, and 5 CVEs were assigned.

3.2 System Boot Process and Bootloader Features

In this section, we first introduce the two main types of firmware from which the bootloader is started. We then explain how the bootloader takes control of the system and also discuss the typical workflow of a bootloader and the runtime environment in which it operates, covering CPU state, memory layout, library support, and peripheral access. Although we use the Intel x86/x64 architecture as an example, the concepts for the workflow and runtime environment apply similarly to other architectures. Additionally, we describe the common features provided by the bootloader, which either enable a user-defined interface or assist in booting the OS.

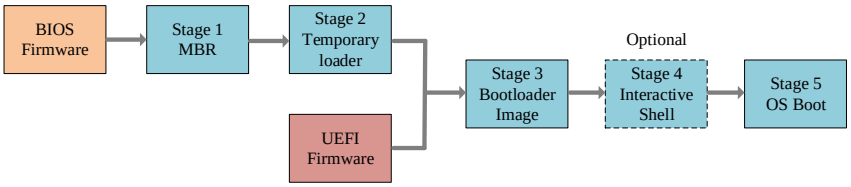

Figure 3.1: Schematic overview of bootloader workflow

3.2.1 PC Firmware

There are two main types of firmware implementations: BIOS (Basic Input/Output System) and UEFI (Unified Extensible Firmware Interface). Both perform similar tasks during the initial boot phase: when the system is switched on, the firmware flashed onto the board by the manufacturer is executed by the processor. This firmware initializes and tests the system hardware, a process known as power-on self-test (POST), which includes components such as the CPU, DRAM, motherboard, and GPU. The firmware then loads the bootloader, usually from a storage device, which further initializes the OS. The firmware adheres to either the BIOS or UEFI standard to load and communicate with the bootloader. We elaborate on the differences between these two types of firmware.

BIOS. As a legacy boot design, BIOS offers a straightforward method for loading the bootloader. It enumerates the storage devices and checks if the first sector matches the signature `0x55AA` [228]. If the firmware recognizes the first sector as a boot sector, it reads the sector into memory at a fixed address (`0x7C00` on an IBM PC-compatible computer) and then jumps to this address. The boot sector, also known as the Master Boot Record (MBR), contains only the first stage of the bootloader. The size of the MBR (512 bytes) is too small for a bootloader to perform all its functions, so the first stage bootloader loads additional sectors from the storage device and continues execution from there. The BIOS provides utilities for the bootloader, such as access to the hard disk via interrupts [171]. Once the bootloader has initialized the OS, the OS overwrites the interrupt table, and the firmware ends its life cycle.

UEFI. UEFI is the successor to BIOS and overcomes several of its limitations. For example, it supports GPT partition tables, which enables the use of large storage devices. While the BIOS firmware looks for the MBR, the UEFI firmware can recognize the partitions of storage devices and understand the FAT file system. UEFI identifies the boot partition via a specific GPT partition GUID [213] and tries to analyze the partition as a FAT file system. When successful, it searches for the boot application file [161] and loads it into memory. In this scenario, the bootloader appears as a UEFI application. While the bootloader is running, the firmware uses the EFI System Table [212] (a set of function pointers) to provide boot services [214], such as reading files and allocating memory. After initialization of the OS, the firmware remains in memory and provides the OS runtime services [215].

For the rest of the chapter, we will refer to bootloaders loaded and launched by BIOS firmware as “BIOS bootloaders” and those loaded and launched by UEFI firmware as “UEFI bootloaders”.

3.2.2 Bootloader Workflow

The bootloader typically consists of several runtime steps to achieve its final goal—booting the OS. Although the implementation may vary across different bootloaders, we can generalize the workflow into the following five stages as shown in Figure 3.1:

1. The MBR sets up a simple execution environment, such as the stack and BSS segment, and then loads the next stage data from the storage device into memory.

2. The temporary loader parses the storage device partitions to find the bootloader image file. If the file exists, the bootloader loads the image into memory and begins execution from there.

3. The UEFI firmware takes over the tasks of the first two stages. Therefore, the UEFI bootloader has the same workflow as the BIOS bootloader from the third stage onwards. In this stage, the bootloader image has been loaded into the memory and the bootloader initializes the entire execution environment. Global data structures, necessary peripheral devices, and configuration files are initialized in this phase.

4. In the fourth stage, the bootloader provides the user with an interactive shell, if available. The user can perform peripheral access tasks such as reading files, sending network packets, and changing OS parameters. In particular, the UEFI bootloader can dynamically load drivers at this stage based on user requests. These drivers can introduce additional features and functions, e.g., support for additional file systems and different types of file parsers.

5. In the final stage, the bootloader loads the OS image into memory and prepares the configuration parameters according to the user's modifications. Finally, the bootloader hands over control to the OS, thus concluding its life cycle.

3.2.3 Runtime Environment

Unlike the OS, the bootloader operates in a bare metal environment after the firmware hands over system control. Specifically, the bootloader runtime environment has the following characteristics:

CPU & Memory. The BIOS bootloader starts in real mode [101]. It initializes a simple flat segmentation scheme and keeps paging disabled throughout its life cycle [87, 129, 63]. Without paging support, the bootloader can access almost *any* memory without crashing. In contrast, the UEFI firmware provides the bootloader with a more complete environment. Once the UEFI application is loaded by the firmware, paging and segmentation are properly initialized. Consequently, any invalid memory access directly leads to a CPU exception.

Library Support. In typical applications, several helper functionalities such as the standard library [88], the operating system, and drivers facilitate a simple *Hello World* printing function. However, for a bootloader, achieving the same functionality is more challenging: without support from libraries and the OS, the bootloader must implement its own task scheduling, file system, file parser, and utility functions such

as memory copying and string comparison. For instance, a file-read operation requires the bootloader to implement the entire stack of functions, including file path parsing, file handler management, file system, block device access, and specific storage device drivers. Although UEFI firmware provides a richer environment, including FAT file system access and heap memory management, making development easier, it is still insufficient to implement the complex features described in the following section. Due to the limited environment support, the bootloader is designed and implemented as a self-contained standalone application.

Peripheral Access. While the bootloader has limited library support, it has high privileges to access peripherals. The Intel x86/x64 architecture does not allow user space applications to access IO ports. However, the bootloader runs entirely in kernel space, granting it full access to all peripherals.

3.2.4 Bootloader Features

The main goal of the bootloader is to facilitate the booting of the OS. While a simple bootloader may directly load the OS image into memory and transfer control to it, modern bootloaders offer richer features and functionalities beyond basic initialization. We summarize the end-user features provided by bootloaders into the following five categories:

UI Component Customization. The bootloader may provide end users with an interactive command line or a graphical user interface. In both cases, the bootloader allows the user to customize components such as background images, fonts, icons, language, and other UI elements. To use a customized UI component, the user usually has to place the file in a specific location specified by the bootloader. During initialization, the bootloader automatically detects the components specified by the user and displays them, improving the user interface.

Device Manipulation. The bootloader provides utilities that allow the user to access peripheral devices, providing important tools for performing early hardware and network tests. For example, these utilities allow the user to access the network card and send ICMP ping packets to test Internet connectivity or read files to check the functionality of the hard disk.

Authentication. For security reasons, certain bootloaders may require identity authentication to access certain important configuration options. When a user attempts to access sensitive functions, the bootloader prompts the user to enter a password or provide an access token for authentication.

Boot Environment Preparation. Before starting the OS booting process, the bootloader prepares the booting environment. It needs to allocate suitable memory for the OS image, prepare the kernel parameters, verify the image integrity, and perform other necessary tasks. The behavior can be adjusted according to the user's requests, such as by adding additional parameters to the kernel.

Boot Selection. Modern bootloaders support several boot methods, allowing the OS image to be obtained from different sources, such as remote servers or local storage devices. Bootloaders that implement the multiboot protocol [160] allow booting into different operating systems. Users have the option to select a location or an image file

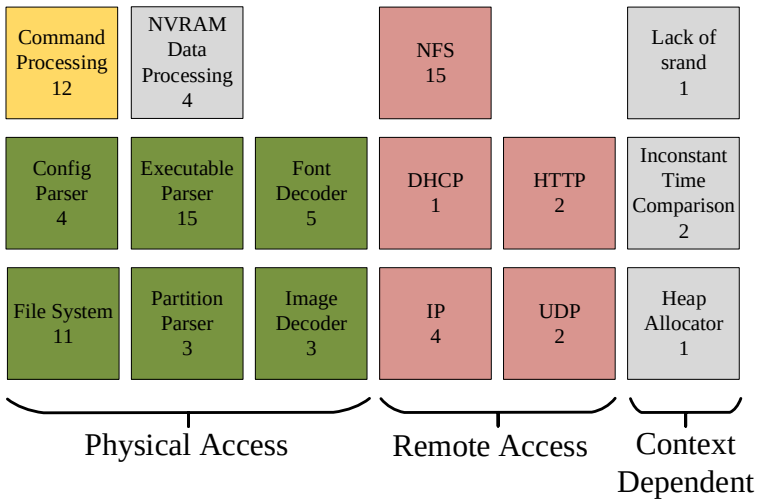

Figure 3.2: Number and root causes of collected bootloader CVEs

from which to start the boot process.

3.3 Bootloader Memory Safety Analysis

In this section, we perform a comprehensive memory safety analysis of bootloaders. We start with a survey of previous bootloader vulnerabilities to understand the historical context and patterns of exploitation. By analyzing these past vulnerabilities, we identify the main attack surfaces that an attacker can leverage to compromise a bootloader. Based on these insights, we define the threat model for bootloaders. Afterward, we present the nine bootloader targets selected for our analysis, along with the criteria used to choose them. Finally, we perform a concrete analysis of the attack surface of these nine bootloaders and use our observations to assess their security situation.

3.3.1 Survey and Lessons Learned

We exhaustively searched for all available bootloader vulnerabilities from the CVE database and manually inspected their root causes. As shown in Figure 3.2, we categorize the 85 collected vulnerabilities based on the attacker's capabilities into three categories: physical access, remote access, and context-dependent. With physical access, an attacker can modify local storage data, plug in extra devices such as LED lights and USB sticks, input commands, etc. Remote access allows an attacker to send the bootloader network packets, Bluetooth messages, radio signals, etc. Context-dependent vulnerabilities exist without specific access dependencies but still pose a risk under certain conditions. Our analysis reveals that, with physical or remote access to the bootloader, malicious inputs primarily originate from three sources: *storage*, *network*, and *console*, represented in Figure 3.2 in dark green, pink, and yellow, respectively. For example, vulnerabilities related to the file system and file parser are linked to storage, while IP packet issues are associated with the network. As shown in Table 3.1, of the 85 CVEs from four

Table 3.1: Number and distribution of collected bootloader CVEs

	Storage	Network	Console	Others	*Total*
GRUB	18	2	10	3	33
barebox	0	3	0	2	5
shim	8	3	0	2	13
Das U-Boot	15	16	2	1	34
	41 (48%)	24 (28%)	12 (14%)	8 (9%)	85

bootloaders, 48% are attributed to storage issues, 28% to network issues, and 14% to console issues. Only 9% result from other factors such as heap allocator bugs and side-channel timing attacks.

3.3.2 Threat Model

We assume that the attacker's goal is to compromise the system by exploiting memory corruption vulnerabilities, such as buffer overflows or null-pointer dereference, in the bootloader to cause it to crash or even bypass the secure boot process. Specifically, we outline the following heuristics regarding what attackers can and cannot utilize to achieve their goals.

Firmware. We always assume that the firmware flashed on the board cannot be directly modified by the attacker. Depending on the firmware implementation, modifying, replacing, or updating the firmware image typically requires it to be signed with an authorized key. If the firmware image is not properly signed, it will be immediately rejected. Given that an attacker cannot forge a valid key, it is reasonable to assume that the firmware remains intact. Furthermore, since the firmware executes before the bootloader, an attacker who could modify the firmware would be able to corrupt the system without needing to exploit bootloader vulnerabilities. With this assumption, the integrity of the bootloader can also be verified and guaranteed, ensuring that the bootloader image cannot be modified by the attacker.

CPU & Memory Access. We assume that the attacker cannot directly modify the CPU state and memory, including CPU register values, cache, and memory data. A bootloader running inside a virtual machine that can be introspected by a hypervisor might be susceptible to such an attack. However, Trusted Execution Environments (TEEs) such as Intel SGX [103], Intel TDX [104], and ARM TrustZone [5] address this vulnerability. Finding vulnerabilities in TEE software [48] [240] [224] is beyond the scope of this chapter.

Persistent Storage Access. We assume that the attacker cannot directly read or write to persistent storage, such as NVRAM variables and the UEFI signature database, as these are typically writable only by the manufacturers. Although some attacks [187] can manipulate NVRAM variables from the OS, we exclude them from our threat model. However, if the bootloader implements an NVRAM [227] variable access function that can be exploited through malicious control hijacking, we consider this a valid attack.

Peripheral Access. We assume that the attacker has limited peripheral access to the system. An attacker can plug in extra devices, such as a USB stick or hard drive, and can modify any files on existing storage devices, except for the bootloader and OS images, as we assume that their integrity has been verified during the secure

Table 3.2: Detailed overview of the nine bootloaders selected for assessment. The image size refers to the compiled bootloader's binary size, note that certain bootloaders may contain dynamic modules, which are excluded from the size. We only list the operating systems explicitly claimed by the bootloader, though they might be compatible with other operating systems. CI indicates whether the bootloader offers an interactive command line string input interface for end users. We only list the features that are supported at the time of thesis writing, developers might add more features after that.

	Version	# of Source Files	Image Size	Supported Targets	Firmware	CI	Last Update
GRUB [87]	v2.02-beta2	5411	297KB	Linux, GNU/Hurd, macOS, BSD Solaris/illumos (x86 port), Windows	BIOS,UEFI	✓	2024.05
Limine [129]	v7.x	442	105KB	Linux	BIOS	-	2024.05
Das U-Boot [63]	v2024.04-rc3	11048	1.0MB	Linux, NetBSD, VxWorks, QNX RTEMS, INTEGRITY	BIOS	✓	2024.05
barebox [15]	v2024.01	4878	681KB	RTOSes	UEFI	✓	2024.05
CloverBootloader [49]	v2-5158	9048	1.6MB	macOS	UEFI	-	2024.05
Easyboot [31]	v1.0.0	47	69KB	Linux, Windows, OpenBSD, FreeBSD, FreeDOS ReactOS, MenuetOS, KolibriOS, SerenityOS, Haiku	UEFI	-	2024.04
rEFInd [175]	v0.14.3	173	306KB	Linux, Windows, macOS, TrueOS	UEFI	-	2024.04
systemd-boot [199]	v256	92	213KB	Linux	UEFI	-	2024.04
shim [173]	v15.8	546	936KB	GRUB	UEFI	-	2024.05

boot chain. However, if a memory corruption vulnerability leads to the modification of the verification key and subsequently allows the loading of a malicious image, we consider this a valid attack. We assume that an attacker can provide any input via the bootloader peripheral access, such as malicious network packets to the network card or keyboard input string to the console. However, the attacker cannot signal an interrupt on behalf of the device, as this is relatively difficult to manipulate. In addition, we assume that the full disk encryption mechanism is not deployed. If it is used, the bootloader is usually a proprietary and close-source software to prevent physical attack. Since the decryption key is not publicly available, we consider this situation outside the scope of this chapter.

3.3.3 Target Selection

In this paper, we analyze nine bootloaders selected from a list of available bootloaders [229]. Our choice of targets is based on the following criteria:

Availability. Since proprietary bootloaders can be challenging to access and deploy, we exclusively select open-source bootloaders for our analysis. Bootloaders bundled with operating systems, those lacking available source code, or proprietary software are excluded from our targets.

Maintenance. We select only those bootloaders that have been actively maintained over the past two years. Legacy bootloaders, while still used by some users, are excluded from our target selection due to potential compatibility issues with modern machines and peripherals, as well as their lack of updated security checks and patches. Therefore, we focus solely on actively maintained bootloaders for our analysis.

Version. We select the latest version of each bootloader if multiple versions exist (e.g., GRUB [87] has several versions, but we specifically choose GRUB2 as our analysis target).

Based on these criteria, we collected nine bootloaders as shown in Table 3.2. These include widely used bootloaders such as GRUB, Das U-Boot, and systemd-boot. They

Table 3.3: Bootloader attack surface analysis for storage device input(1)

File	config, jpeg, png tga, font, mo envblock, keymap	config, png, bmp gif, psd, pic jpeg, pnm, hdr tga	fdt, slre	base64, srec, fdt bmp, png, qoi
File system	zfs, affs, bfs btrfs, cbfs, cpiofs fatfs, ext2fs, f2fs hfs, hfsplus, iso9660 jfs, minixfs, nilfs ntfs, reiserfs, sfs squashfs, udffs, ufs xfs	fatfs, iso9660	btrfs, cbfs, cramfs erofs, ext4fs, fatfs reiserfs, squashfs ubifs, yaffs2, zfs jffs2	cramfs, ext4fs, fatfs jffs2, squashfs, ubifs bpkfs, nfs
Partition	Linux/ADFS, amiga disklabel64 macintosh, GPT MS-DOS, SUN SUN PC, BSDlabel	GPT, MS-DOS	amiga, GPT MS-DOS, macintosh	GPT, MS-DOS
	GRUB	Limine	Das U-Boot	barebox

Table 3.4: Bootloader attack surface analysis for storage device input(2)

File	base64, svg, png icns, bmp, png	config	png, jpeg, bmp icns	bcd, config	csv
File system	hfs, iso9660, ext2fs ext4fs, reiserfs, fatfs	afs, befs, exfatfs ext234fs, minix3fs ntfs, ufs, xfs fatfs, fszfs	btrfs, ext2fs, ext4fs hfs, iso9660, reiserfs ntfs	-	-
Partition	-	-	-	-	-
	CloverBootloader	Easyboot	rEFInd	systemd-boot	shim

support mainstream operating systems: Windows, Linux, and macOS, and cover both BIOS and UEFI environments. All selected bootloaders have been updated regularly up to the time of writing this paper. We are confident that our selection is representative for analyzing the memory safety of bootloaders.

3.3.4 Attack Surface Analysis in Practice

In this section, we conduct a detailed memory safety analysis of the three attack surfaces *storage*, *network*, and *console* identified earlier for our nine selected targets. Although other peripherals can also contribute to vulnerabilities, they do not involve the complex processing logic found in these primary three attack vectors. Thus, we summarize them as "others" which will be further discussed in Section 3.6 and focus on the main three attack surfaces.

3.3.4.1 Storage

As shown in Table 3.3 3.4, storage device data follows a layered design. A storage device is divided into several partitions, each of which can be formatted with different file systems. The file system organizes and places various types of files in directories appropriately.

Partition. The bootloader processes local storage data by first identifying the partitions. The partition table contains metadata that allows the bootloader to identify information about each partition. MS-DOS and GPT are two widely used partition

schemes, both supported by four different bootloaders in our targets. Some bootloaders depend on the UEFI firmware to recognize partitions and operate directly on the partition. As a result, some bootloaders do not support any partitions themselves but allow file systems to be deployed. Among the nine targets, GRUB supports the largest number of partition types. If there is a vulnerability in the partition table processing logic, the bootloader could be compromised. For instance, CVE-2019-13103, targeting Das U-Boot, is an attack where a crafted self-referential MS-DOS partition table can cause infinite recursion, leading to an infinitely growing stack.

File System. After identifying the partitions, the bootloader attempts to mount file systems on them. The file system contains metadata, such as the superblock, inode tables, and directory tables, to organize the files. Only after successfully mounting a file system on a partition are subsequent file operations, such as opening, reading, and writing files, allowed. The bootloader's primary goal is to locate and launch the OS image file, so supporting various file systems is essential. Among the nine bootloaders, GRUB supports more than 20 types of file systems, the highest number. File systems are complex and involve intricate processing logic, making their implementation prone to bugs [234]. For instance, CVE-2023-4692 targets GRUB's NTFS driver and demonstrates an attack where a specially crafted NTFS file system image that contains a fragmented master file table can lead to a heap overflow.

Files. Like any other application, the bootloaders also need to handle various types of files. These can be summarized into the following three categories:

Multimedia To provide users with tailored interfaces, bootloaders allow customization of the user interface, including fonts, background images, and even themes. As shown in Table 3.3 3.4, common image types supported by bootloaders include PNG, JPEG, and BMP.

Environment Related Flies A typical environment-related file is the configuration file. This file can specify the path of the OS image, extra command line parameters passed to the kernel, the boot protocol, and more. Bootloaders that provide an interactive interface may treat the configuration file as a script and automatically execute it when the bootloader starts. For instance, GRUB allows users to define variables and execute GRUB shell commands within the configuration file. Another environment-related file is the *flat device tree* (FDT), which details the peripheral information. Users can customize this file to change the bootloader's behavior to suit their preferences and requirements.

Other Files We summarize other types of files in this category. They are occasionally used by some specific bootloaders. For instance, shim uses CSV format to parse the executable SBAT section data.

Parsers for various file formats are frequent points of attack in bootloaders. For instance, CVE-2022-2601 demonstrates that a crafted, malicious font file with an attacker-controlled size value can cause a heap overflow, ultimately circumventing the secure boot mechanism.

Table 3.5: Bootloader attack surface analysis for network device input(1). The first column represents the OSI model layers. Protocol wrapper means that the bootloader does not implement the whole protocol but processes the protocol payload data directly.

Application	HTTP, DNS TFTP	TFTP wrapper	HTTP, TFTP, NFS DNS, DHCP, SNTP	DHCP, DNS, NFS SNTP
Transport	TCP, UDP	-	TCP, UDP	UDP
Network	IP, ICMP ICMP64	-	IP, ICMP ICMP64, NDP	IP, ICMP
Data link	ETH, ARP	-	ETH, ARP CDP, RARP	ETH, ARP
	GRUB	Limine	Das U-Boot	barebox

Table 3.6: Bootloader attack surface analysis for network device input (2).

Application	-	-	-	-	HTTP wrapper
Transport	-	-	-	-	
Network	-	-	-	-	
Data link	-	-	-	-	
	CloverBootloader	Easyboot	rEFInd	systemd-boot	shim

3.3.4.2 Network

To support remote booting, such as PXE network boot and other network-related features, bootloaders might implement their own network protocol stack. As shown in Table 3.5 3.6, among the nine targets, GRUB, Das U-Boot, and barebox support a full-stack network protocol. Bootloaders like Limine and shim rely on the firmware to provide basic network protocol implementation. Limine and shim utilize the firmware's TFTP and HTTP, respectively, to download images into local memory. Each layer of the network stack could be a potentially vulnerable point. For instance, CVE-2023-40547 represents a heap overflow vulnerability in shim's HTTP protocol implementation (application layer). A crafted HTTP response containing a small value in the length field leads to a small memory allocation, and the buffer is further overwritten by the HTTP response content. Similarly, CVE-2022-30552 demonstrates an attack in the network layer where a specific range of values in the IP length field can result in a buffer overflow.

3.3.4.3 Console

Bootloaders that provide an interactive interface accept user input. Among the nine analyzed bootloaders, GRUB, Das U-Boot, and barebox implement this functionality. Note that some bootloaders provide the user with a selection list to choose which OS to boot—we do not count it as an interactive interface. User input can trigger various functions in the bootloader, such as reading a file or sending network packets. The bootloader typically accepts user input as a string and parses it into several options. This process can lead to vulnerabilities depending on the parsing implementation and

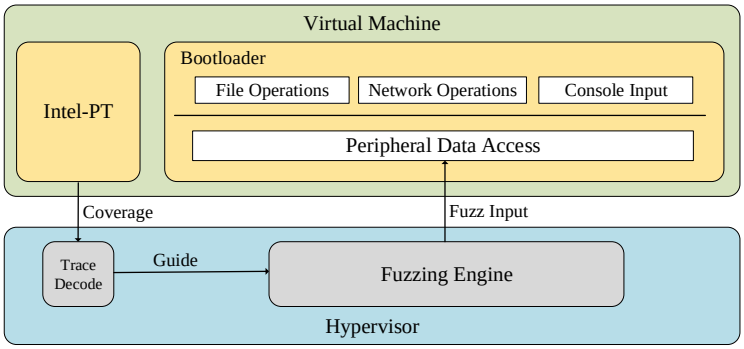

Figure 3.3: Bootloader fuzzing overview

the functionalities involved. For instance, CVE-2020-27749 demonstrates such an attack: An attacker can invoke the *i2c* command with a negative length value such as *0xffffffff* (-1 when parsed as a 32-bit signed integer). This value is treated as a signed integer, bypassing the security check. However, it is later used as an unsigned integer, leading to a stack overflow.

3.4 Generic Bootloader Fuzzing Framework Design

Our bootloader memory safety analysis revealed a wide variety of potential vulnerabilities. Based on these insights, we now present the design and implementation of a fuzzing framework to help developers detect new vulnerabilities at scale. Figure 3.3 shows a high-level overview of the design. Since the bootloader needs a real runtime environment, we simulate a virtual machine where the bootloader runs and guides the fuzzing via coverage feedback collected from Intel-PT. Our attack surface analysis informs the bootloader harness implementation. By identifying the primary attack surfaces where malicious input can be fed to the bootloader, we pinpoint the universal operations that trigger device access and the interfaces through which input is fed to the bootloader. We intercept peripheral access for three types of devices: storage, network, and console. When the bootloader operates and receives data from the device, our fuzz input is fed into it. Additionally, we trigger different operations to prompt the bootloader to read data from these devices and process it. In the following, we elaborate on each component.

3.4.1 Harness

Since we focus exclusively on open-source bootloaders, we implement the harness directly within the source code, which we must have access to.

3.4.1.1 Operations

We perform various operations to trigger peripheral access. The operations occur immediately after the bootloader initializes its execution environment, which typically

happens in the *main()* function of each bootloader. Our operations target all attack surfaces, summarized in the following categories:

- To trigger storage data processing, we perform the following sequence of operations:

  1. Discover new storage devices.
  2. Mount all supported file systems on the partitions.
  3. Open the root directory of the successfully mounted partition.
  4. Enumerate the files and directories in the root directory.
  5. Read and write a fixed length of data in the files.
  6. Close the opened files.
  7. Delete the files.
  8. Unmount the file system.

  Note that some bootloaders do not support writing operations, so we skip those in such cases.

- Fuzzing different file parsers by feeding input from the storage device generates large amounts of redundant data, which is inefficient and unnecessary. Therefore, to test file parsers, we directly invoke the target function in our harness with the fuzz input as the argument. We identify the parsing functions by searching through the source code. Typically, these functions are located in the *lib* directory. For instance, CloverBootloader uses the function *egDecodeBMP* to parse BMP images. We call this function directly in our harness, place the fuzz input in memory, and pass its pointer to the function.

- To trigger network data processing, we perform the following operations:

  1. Discover network interfaces.
  2. Assign static IP address, network mask, gateway address, and remote server address to the available interface.
  3. Send TFTP, HTTP, ICMP, and other supported network packets to the remote server.
  4. Receive and trigger the callback functions for network packets.

  Note that for some protocols such as TCP and IP, they usually make up part of the network packet. When an application layer packet is sent, they are automatically assembled and sent together.

- To trigger console data processing, we locate the console processing function for the bootloader. This function typically exists in an infinite loop within the main function and usually accepts a string as a parameter. We directly call this function with our fuzz input as the parameter. For instance, Das U-Boot accepts and processes user input via function *run_command_repeatable*.

With these operations, we are able to trigger input data processing across all three attack surfaces.

3.4.1.2 Peripheral Data Access Hook

When the bootloader reads from or writes to the peripheral, our fuzz input needs to be fed instead of directly interacting with the simulated devices. For storage data and network packets, the fuzz input cannot be directly injected into a single function since multiple functions are involved in the processing logic. We have observed that bootloaders commonly follow a layered design. This layered design ensures that all data read from or written to the devices passes through a single interface. We hook into the block device and network interface access layers so that when the bootloader tries to read from or write to the device, our fuzz input replaces the original data.

A flag is used to control the feeding of the fuzz input. When the flag is true, the fuzz input is fed; otherwise, the original data is used, allowing the bootloader to initialize its environment successfully. Once the fuzz input is exhausted, we return an error code to the caller function to indicate the end of the input, depending on the specific bootloader implementation. Additionally, when the bootloader writes to the device, we redirect the data to our fuzz input buffer. This keeps the data read from the fuzz input buffer always updated. Note that some bootloaders, such as Limine and shim, do not support full-stack network protocols, thus the universal interface does not exist. In these cases, we feed the fuzz input by hooking the functions that the bootloaders use to communicate with the firmware.

3.4.2 Fuzzing Engine

We implement our fuzzing framework based on kAFL [183, 181], which supports snapshot, fuzzing process control, and various mutation strategies. kAFL is a hypervisor-based, coverage-guided fuzzing tool designed for Intel x86 programs. It can be used to fuzz different OS kernels and user-space applications. Since kAFL relies on two important Intel CPU features, Intel PT and Intel VT, it only supports programs designed for the Intel x86 ISA. Consequently, we compiled all our targets for the Intel x86 architecture. Note that the handling of "high-level" data, such as file system or network packets, remains consistent across different architectures, so compiling the bootloader into a pure x86 architecture is not a problem. However, this does not hold for the device drivers, as the implementation of the device drivers is tightly coupled to the specific devices, which can differ on different architectures.

3.4.3 Crash Detection

We aim to detect memory corruption vulnerabilities. Our framework reports a potential vulnerability if an exception occurs in the virtual machine. Since the bootloader operates in a bare metal environment without the exception handling mechanisms found in typical applications, we have implemented and added the following features to observe crashes.

Paging. While UEFI bootloaders are executed in a paging-enabled environment, accessing invalid memory immediately triggers an exception. However, for BIOS bootloaders, paging is disabled by default. Therefore, we implement a simple paging mechanism for BIOS bootloaders. From our experience, the bootloader rarely accesses high-address memory. Thus, we map a linear 0–2GB virtual address space to the same

Table 3.7: CVEs collected from Snyk vulnerability database for reproduction

	CVE	Bootloader	Category	Reproduce
1	CVE-2023-4692	GRUB	Storage	✓
2	CVE-2023-4693	GRUB	Storage	✓
3	CVE-2020-8432	Das U-Boot	Console	✓
4	CVE-2022-33103	Das U-Boot	Storage	✓
5	CVE-2019-15937	barebox	Network	✓
6	CVE-2019-15938	barebox	Network	✓
7	CVE-2023-40547	shim	Network	✓

physical address space, leaving other memory unmapped. When the bootloader performs an arbitrary read or write operation, it might access the unmapped memory and trigger an exception.

Interrupt. With the default interrupt handling mechanism, the bootloader may enter an infinite loop or even shut down the virtual machine when it encounters a crash. To address this, we overwrite the first 16 interrupt gate vector entries with our hook functions. These vectors handle exceptions such as division-by-zero, segment faults, and invalid opcodes. When an exception occurs, such as an invalid memory access, the hook function reports the crash to the fuzzer. Afterward, the fuzzer automatically restores the snapshot and continues with the next fuzzing iteration.

Panic Hook. The bootloaders can detect invalid input by performing sanity checks. When an explicit error occurs, the bootloader may invoke a panic or hang function, which typically shuts down the virtual machine. To prevent the bootloader from terminating and to save fuzzing time, we hook these functions to report a regular exit to the fuzzer, as these errors do not lead to vulnerabilities.

Heap Sanitizer. Some vulnerabilities are caused by heap buffer overflows. To detect such cases, we design and implement a straightforward yet effective heap sanitizer. Upon heap allocation, we increase the allocation size by 8 bytes. These extra 8 bytes are used to store a magic number, which is later checked. If the magic number does not match, we report a heap overflow. We implement the sanitizer by hooking the heap allocation and deallocation functions. At the allocation stage, we record the size and allocated pointer. The magic number is stored immediately after the allocated memory, in a region not supposed to be overwritten. During deallocation, we verify if the pointer is recorded and if the magic number matches. If either condition is not met, we report an invalid free or a heap buffer overflow. Otherwise, we remove the heap memory information from the recording. Additionally, we periodically check the magic number for all allocated heap memory to detect heap overflows that occur during execution.

3.5 Evaluation

Next, we thoroughly evaluate our test framework and discuss the results. We aim to answer four research questions:

Table 3.8: Number of modified or added lines of code to bootloaders

	Paging & Interrupt	Heap Sanitizer	Harness
GRUB	240	83	376
Limine	240	83	213
Das U-Boot	240	83	358
barebox	76	83	301
CloverBootloader	76	83	214
Easyboot	76	83	70
rEFInd	76	83	258
systemd-boot	76	83	53
shim	76	83	272

Table 3.9: Detected and reported vulnerabilities compared with static analysis. *CSA*: Clang Static Analyzer. *Fuzz*: our fuzzing framework. TP: True Positive.

	CodeQL		**CSA**		**Fuzz**	
	TP	Reported	TP	Reported	TP	Reported
GRUB	1	18	1	88	14	14
Limine	0	0	0	2	4	4
Das U-Boot	2	34	0	25	3	4
barebox	0	6	3	19	5	6
CloverBootloader	0	40	0	0	3	3
Easyboot	0	0	0	1	3	5
rEFInd	0	0	0	10	7	7
systemd-boot	0	0	0	6	0	0
shim	0	7	0	0	0	0
	3	105	4	151	39	43

RQ1: Can our bootloader fuzzing framework reproduce previously identified bootloader vulnerabilities across the three main attack surfaces?

RQ2: Can our bootloader fuzzing framework detect new bootloader vulnerabilities?

RQ3: Compared to other vulnerability detection methods, what are the advantages and drawbacks of our approach?

RQ4: How much effort is required to implement an extension to the framework for a bootloader?

3.5.1 Experiment Setup

We conducted the fuzzing experiments on three servers, each equipped with a 104-core Intel Xeon Gold 5320 CPU @ 2.20GHz and 252 GB of RAM, running Ubuntu 22.04.1 LTS. For each attack surface, we assigned CPU cores with different weights. For instance, we assigned ten cores for fuzzing the file system and only one core for a specific file parser. This distribution was based on the input size–the file system inputs are larger and thus require more computing resources for exploration. In total, the fuzzing experiments lasted three weeks.

3.5.2 Reproducibility of Known Vulnerabilities (RQ1)

As shown in Table 3.7, we collected seven recently available bootloader vulnerabilities that can be compiled for the Intel x86 architecture. These vulnerabilities span the three main attack surfaces.

CVE-2023-4692 and CVE-2023-4693 The two vulnerabilities demonstrate that a crafted NTFS file system could lead to heap overwrite and potentially bypass secure boot in GRUB. The vulnerabilities exist in the NTFS attribute list parsing logic, where

the end of the attribute buffer is not checked, allowing the buffer to be accessed out of bounds.

CVE-2020-8432 This bug demonstrates a double-free vulnerability in the Das U-Boot *gpt rename* command. Das U-Boot allows users to change the GPT partition name via this command. Before changing the partition name, it collects the storage device partition information and stores it in a heap buffer. However, if the rename operation fails and returns *-1*, it deallocates the buffer and jumps to the cleanup code where the buffer is deallocated again. To trigger this vulnerability, we crafted a GPT partition table and named one of the partitions with an environment variable-like string. Das U-Boot expands this to the environment variable value, causing a sanity check failure in the rename function. Fuzzing the command line parsing logic made it easy to find the crash input by defining a specific environment string in advance.

CVE-2022-33103 This bug represents a buffer overflow vulnerability in the Das U-Boot squash file system implementation. While the regular file name for most file systems is less than 255 bytes long, the squash file system defines a two-byte-long length field for the path. When reading from a directory, Das U-Boot allocates a fixed-length buffer for the returned file name. Although the *mksquashfs* tool prevents users from generating a long file name, the fuzzer can mutate and generate such a file.

CVE-2019-15937 and CVE-2019-15938 These CVEs show two buffer overwrite vulnerabilities in the network file system implementation in barebox. When barebox tries to read a symbolic link file from a remote server, a packet that contains a length field indicating the original file path followed by the actual file name is sent to the the client. However, barebox did not check the length of the reply packet from the server and directly copied the path to a fixed-length global buffer, assuming the path length is always less than 2048 bytes. The length field is 4 bytes long in the network packet and can theoretically be large enough to overwrite the whole bootloader's physical memory.

CVE-2023-40547 This vulnerability presents a heap overwrite vulnerability in shim's HTTP content processing. Although shim does not implement a full-stack network protocol, it receives remote bootable images via HTTP. A length field in the HTTP header indicates the length of the following HTTP content, but shim did not correctly check this field and allocated a buffer of the exact length specified in the packet. While copying HTTP content from the UEFI firmware API, the actual content could exceed the allocated buffer. In our experiments, while the heap overwrite did not directly cause a crash, it modified the magic number of the heap sanitizer, causing our sanitizer to report a crash.

Our fuzzing framework was able to reproduce them within several hours successfully. One of them, CVE-2022-33103, can be triggered immediately when the initial seed is sent during the fuzzing campaign. The exception, however, is CVE-2020-8432. We found that a specially named partition is required to trigger the crash. After naming the partition accordingly, the crash could be triggered by fuzzing the command line processing function.

3.5.3 Finding New Vulnerabilities (RQ2)

During our evaluation, we found 39 vulnerabilities, of which 38 were previously unknown. Table 3.10 provides an overview of these vulnerabilities, which successfully cover all three main attack surfaces. Somewhat surprisingly, we found no vulnerabilities in the two bootloaders shim and systemd-boot. We observed that these bootloaders are rather simple and offer fewer attack surfaces compared to the other seven bootloaders.

3.5.3.1 Vulnerability Disclosure

We followed coordinated disclosure best practices and responsibly disclosed the discovered vulnerabilities to the developers. Out of the 39 vulnerabilities we found, 29 have been confirmed or patched by the developers at the time of writing. Since we evaluate active bootloader projects, the majority of the developers responded quickly to our reports.

3.5.3.2 Case Study

We present three specific patched cases in this section to illustrate different examples of the vulnerabilities we have found. We refrain from discussing vulnerabilities that have not yet been fixed by the developers.

```
#define PKTSIZE 1536
char *net_alloc_packet() {
    return dma_alloc(PKTSIZE);
}
int ping_reply(...) {
    ...
    packet = net_alloc_packet();
    if (!packet) return 0;
    // heap overflow here!
    memcpy(packet, pkt, ETHER_HDR_SIZE + len);
}
```

Listing 1: A heap overflow in barebox

Listing 1 presents an out-of-bound write in the barebox ARP implementation. The implementation copies the received packet into a fixed-length buffer, the size of which is defined by the PKTSIZE macro. However, the Ethernet packet could be larger than that in rare cases, such as with jumbo frames. In such cases, the pointer returned by *net_alloc_packet* could be overwritten by the subsequent *memcpy* operation.

```
uint32_t inodes_per_group;
void loadinode(uint32_t inode) {
    ...
    // divide by zero here!
    uint32_t block_offs = ((inode - 1)
    / inodes_per_group) * desc_size;
    uint32_t inode_offs = ((inode - 1)
```

```
    % inodes_per_group) * inode_size;
}
void _start() {
    ...
    inodes_per_group = sb->s_inodes_per_group;
}
```

Listing 2: A divide by zero in Easyboot

Listing 2 shows a divide-by-zero vulnerability in Easyboot. The variable *inodes_per_group* is directly read from the EXT file system superblock. However, without a proper sanity check, this value could be zero. In the function *loadinode*, the value is used as a divisor to calculate the value of *block_offs* and *inode_offs*.

```
EG_IMAGE* egDecodeBMP(uint8_t *FileData,
size_t FileDataLength, bool WantAlpha) {
    uint32_t RealPixelWidth;
    ...
    RealPixelWidth = BmpHeader->PixelWidth
    > 0 ? BmpHeader->PixelWidth
    : -BmpHeader->PixelWidth;
    ...
    uint32_t x = 0;
    //RealPixelWidth might be smaller than 2!
    for (; x <= RealPixelWidth - 2; x += 2)
    {
    ...
    PixelPtr->Blue = BmpColorMap[Index].Blue;
    ...
    PixelPtr++;
    }
}
```

Listing 3: A heap overflow in Cloverbootloader

Finally, Listing 3 shows an out-of-bound write caused by an integer overflow in the BMP image decoder in Cloverbootloader. The variable *RealPixelWidth* is calculated from the metadata of a BMP image file. However, in the subsequent loop, the value is subtracted by two and compared with an unsigned integer x. If *RealPixelWidth* is smaller than two, the calculated value will be huge and the loop will overwrite a large amount of memory.

3.5.4 Comparison with Other Works (RQ3)

To the best of our knowledge, there is neither a comprehensive memory safety analysis of bootloaders nor ready-to-use fuzzing tools that can be directly used to test different bootloaders. To evaluate the vulnerability detection capability of our fuzzing framework, we resort to two popular and widely used static analysis tools: *CodeQL* [11] and *Clang*

Static Analyzer [131]. CodeQL is an industry-leading semantic code analysis engine maintained by GitHub. It is now integrated with many GitHub open-source projects and runs as a Continuous Integration (CI) backend component. The Clang Static Analyzer is part of the LLVM project. It uses symbolic execution to explore bugs in C/C++/OC and has been integrated into Xcode as a default security checker. They can both target all the source code involved in the compilation. We compare our fuzzing framework against the two static analysis tools. Table 3.9 shows the vulnerability detection result of the nine bootloaders.

3.5.4.1 CodeQL

In total, CodeQL found three true positive vulnerabilities among the 105 reports it generated. After manually analyzing the true positives, we found that one of them is a previously known heap overflow in the file system in GRUB, while the other two are new vulnerabilities. One of the two new vulnerabilities was caused by an out-of-bound buffer read operation, while the other one resulted from an attacker-controlled heap allocation size. They both existed in the file system implementation of Das U-Boot. Our fuzzing framework was also able to detect the file system vulnerability in GRUB; however, it failed to detect the other two vulnerabilities. This is because the out-of-bounds read does not trigger any exception, and our fuzzing can only detect crash vulnerabilities. The other reason is that we lack a proper seed to trigger the memory allocation size control vulnerability. We inspected the false positives reported by CodeQL and found that the following reasons caused them:

Incomplete Control Flow. CodeQL does not work well in inter-file and inter-procedure analysis. A value that is checked in another file or another function would be ignored if the value is used in the analyzing point, especially if the function is invoked via a function pointer.

Missing Context Check. A typical false positive reported by CodeQL is a call to the *strcat* function. Even though the size of the destination buffer is correctly calculated and allocated, the tool still reported a potential buffer overflow.

Wrong Attacker Controlled Data Identification. CodeQL cannot identify which data can be controlled by an attacker. For instance, it reported a false positive in shim where the data is generated from a firmware-calculated string.

3.5.4.2 Clang Static Analyzer

In our experiment, the Clang Static Analyzer found four true positives among the 151 reports generated by the tool. The true positives are not found in the three main attack surfaces and, therefore, could not be detected by our fuzzing framework. We manually inspected their root causes and found that they are caused by hard code null-pointer values and misuse of Unix-like APIs that do not originate from attacker-controlled data. We investigated the false positives reported by the Clang Static Analyzer, and summarize our main findings below:

Broken Constraint. The Clang Static Analyzer could not maintain a set of consistent constraints during the symbolic execution. For instance, a value constraint

to the value zero can be assumed to be non-zero by the execution engine and continues execution. This causes some impossible paths to be reachable.

Broken Control Flow. Like CodeQL, the Clang Static Analyzer cannot perform inter-file and function pointer analysis during the symbolic execution. When the control flow is broken, it lacks enough knowledge to infer a value's constraint.

Broken Value Tracking. Lastly, the Clang Static Analyzer failed to track a field value in a struct. For instance, a pointer in a struct is deallocated and then gets overwritten with another value. The Clang Static Analyzer reported a double free when the new value gets deallocated again.

3.5.4.3 False Positives in Fuzzing

In our experiments, the fuzzer reported several false positives. These false positives were primarily due to an incorrect harness implementation by us. For instance, when fuzzing a device tree parser in Das U-Boot, the fuzzer reported an arbitrary memory write while parsing the device tree file header. Upon manual analysis of our harness, we found that this issue was due to the absence of a sanity check function: this sanity check function is supposed to report an invalid header when it detects an out-of-range value. However, in our implementation, the parser directly used the value without invoking the sanity check function, leading to an arbitrary memory write. Another false positive in barebox exists because the buffer was assumed to be allocated in the heap, however, we passed the fuzz input buffer to the parsing function. The buffer was later deallocated, thus reporting a crash.

Our framework reported in total four false positives due to the harness implementation mistakes that we subsequently fixed. We missed several necessary sanity checks or passed the wrong type of memory to the parsing functions before calling them. Nevertheless, we conclude that our fuzzing framework performs better than the state-of-the-art static analysis tools in both quantity (i.e., more new vulnerabilities) and quality (i.e., fewer false positives).

3.5.5 Manual Effort (RQ4)

The additional manual effort required to extend our framework to support a new bootloader consists mainly of three parts: (i) a paging and interrupt handler hook, (ii) a heap sanitizer, and (iii) a harness. Table 3.8 shows the number of lines of code modified or added for each of the nine bootloaders we evaluated. All bootloaders share the implementations of paging and interrupt handler hooking, and our heap sanitizer. The harness for a specific bootloader depends on the complexity of the bootloader implementation. Our goal is to help bootloader developers identify the vulnerabilities, assuming that they can efficiently implement the harness. We recommend first recognizing the peripheral data access interfaces (e.g., firmware calls or the hardware abstraction layer) to feed the fuzz input to the bootloader under test. Subsequently, the functions intended for the end applications to trigger the peripheral access should be reused. File parsers can be identified by enumerating the supported file types and the corresponding parsing functions.

3.6 Discussion

In our evaluation, we have shown that our proposed approach has successfully uncovered a variety of bugs in different bootloaders. However, there are also several shortcomings that we discuss in this section.

Device Drivers. Despite the lack of complex processing logic in some other peripheral inputs, their vulnerabilities cannot be ignored. Bootloaders communicate with peripherals through device drivers, e.g., Das U-Boot can manage more than thirty types of peripherals, with the number of device driver source code files exceeding 2,000. However, fuzzing bootloader device drivers is a challenging task. With the design of our fuzzing method, we compile all the bootloaders into x86 architecture targets. While this does not pose a problem for high-level data handling, it does not apply to device driver fuzzing. There is no universal interface, such as file operations, to manipulate different peripherals, and there is no common layer, like a data access abstraction layer, to intercept the device access. This requires a significant amount of manual effort to implement the necessary harnesses. Additionally, some peripherals rely on specific architectures incompatible with the Intel extensions, so they cannot be executed when compiled into the bootloader. Therefore, we consider the fuzzing of the device driver as a task for future work.

Beyond Peripherals. In addition to the peripheral input processing logic, other components, such as data structures, encoding and decoding algorithms, and boot management, may also contain vulnerabilities. However, these components are either implicitly used by the peripheral input processing or cannot be directly controlled by the attacker, according to our threat model. For instance, linked lists and heap management are widely used by various file parsers. Therefore, we do not consider them as an attack surface reachable via fuzzing.

Harness. In this chapter, we do not consider file operations as fuzzing input. A fixed sequence of file operations is used to trigger the file system operation. However, Janus [234] highlights that exploring the two-dimensional inputs (i.e., mutating file system metadata on a large image while emitting image-directed file operations) is efficient and effective in file system fuzzing. With our simple and fixed file operations, we might miss some potential vulnerabilities. Nevertheless, in the bootloader scenario, bootloaders typically only expose limited file operations. For instance, some bootloaders only allow file read operations, while file write and symbolic link access operations are not possible. These limited file operations confine our harness to a small range of potential actions.

Fuzzing Seeds. We have collected or generated our fuzzing seeds from both open-source corpora and created them from scratch using tools such as the *mkfs* utility. For certain components, such as image parsers, it is sufficient to use open-source corpora since they cover a wide range of corrupted images. The diversity of fuzzing seeds significantly impacts the efficiency of fuzzing. Some of our generated fuzzing seeds, such as part of the file system images and network packets, may not cover sufficient input space, potentially resulting in false negatives. Consequently, we recognize the need to generate a more diverse set of fuzzing seeds and consider this as a future work.

Heap Sanitizer. Existing sanitizer frameworks designed for bare-metal environ-

ments, such as SHiFT [145], cannot be applied directly as they rely on a specific RTOS environment, tool chain, runtime dependencies, compiler customization, or architecture-related instructions that are not commonly supported by bootloaders. Bootloaders such as GRUB and Das U-Boot also tried to integrate sanitizers into their products [61, 64]. Even with the technical efforts of experienced developers, they can only support native compilation (i.e., compile the bootloader as an ELF or EXE file that can be executed natively). This conflicts with our goal of running the bootloader in a real environment. Due to the complexity of adapting existing frameworks to bootloader fuzzing, we implemented a tailored heap sanitizer to detect out-of-bound heap buffer write vulnerabilities. However, this canary-like heap sanitizer cannot detect out-of-bound heap read vulnerabilities. To accomplish the heap sanitizer task, we made a trade-off between comprehensiveness and usability. Our method is straightforward (i.e., the design is a canary-like sanitizer) but effective given that we found many memory corruption vulnerabilities. We believe that our approach is ideally suited to sanitize heap memory for bootloader applications running in a bare-metal environment, as there are almost no runtime dependencies and compatibility issues.

Mitigation. To mitigate memory corruption vulnerabilities in bootloaders, some developers have already started to deploy static analysis tools and fuzzing in their projects [57, 172]. However, they either do not focus on the entire attack surface or cause too many false positives. To better mitigate memory corruption vulnerabilities in bootloaders, we propose the following methods:

Debloating As the bootloader code base grows, vulnerabilities may arise from the numerous features it contains. To address this, we propose debloating the bootloader at the source code or compilation level. For example, if a memory corruption vulnerability solely happens in a specific file system parsing logic, it can only compromise the bootloader when the file system feature is enabled. While this may affect user experience, a trade-off between security and user experience is necessary to ensure a more secure system.

Fuzzing Comprehensive Attack Surfaces Fuzzing has proven to be an effective method for detecting vulnerabilities. However, without a comprehensive analysis of the attack surface and tailored testing harnesses, easily detectable vulnerabilities may be missed by the fuzzer. Therefore, we propose to include a comprehensive attack surface analysis, as discussed in our thesis, in the development of fuzzing strategies to guide and improve them.

Comparison with Existing Works The two fuzzing tools introduced by Axtens [60] and Starke [157] aim to fuzz the command-line parsing logic in GRUB and Das U-Boot. However, the reasons why we did not directly compare our work to their tools are as follows: 1) They focused solely on console input, while we considered a broader range of attack surfaces. 2) They compiled the bootloader into a native application, whereas we compiled it into an x86 loader, targeting different binaries. In addition, they used AFL as fuzzing backend, while we used kAFL, which implements more advanced fuzzing mechanisms such as mutators and scheduling policies. 3) They did not publish many implementation details.

3.7 Conclusion

In this chapter, we first systematically analyze the bootloader attack surfaces in nine selected open-source bootloaders. The analysis results show that the malicious input mainly comes from three types of peripherals: storage devices, network devices, and console devices. Based on the analysis result, we designed a generic fuzzing framework that can be used to harness various bootloaders despite their implementation details. The experimental results were promising. We found 39 previously unknown vulnerabilities. The true positive and false positive all outperform the industry-leading tools that can target the bootloader. As a typical low-level software that gets executed during the early stage of system boot, bootloaders are attractive targets for attackers. Our analysis shows that dealing with data parsing (storage data, network packets, and user input strings) is still a main threat to memory-unsafe programming languages. In the next chapter, we present embedded system firmware fuzzing, another type of low-level software. By addressing an interrupt-triggering problem when re-hosting the RTOS firmware, we pinpoint the importance of domain-specific knowledge and techniques.

Table 3.10: Detected bootloader vulnerabilities information. For those without status information, we reported them to the developers. These vulnerabilities are still under their investigation.

	Bootloader	Category	Type	Status
1	GRUB	Storage, file parser	Logic bug, heap overflow	Confirmed
2	GRUB	Storage, file parser	Integer overflow, heap overflow	Confirmed
3	GRUB	Storage, file parser	Integer overflow, heap overflow	Confirmed
4	GRUB	Storage, file parser	Integer overflow, heap overflow	Confirmed
5	GRUB	Storage, file parser	Logic bug, use of uninitialized data	Confirmed
6	GRUB	Storage, file system	Lack of boundary check, heap overflow	Confirmed
7	GRUB	Storage, file system	Infinite loop, stack overflow	Confirmed
8	GRUB	Storage, file system	Integer overflow, heap overflow	Confirmed
9	GRUB	Storage, file system	Off-by-one access, heap overflow	Confirmed
10	GRUB	Storage, file system	Integer overflow, heap overflow	Confirmed
11	GRUB	Console, command parsing	Unlimited recursion, stack overflow	Confirmed
12	GRUB	Console, command parsing	Missing sanity check, null-pointer dereference	Confirmed
13	GRUB	Console, command parsing	Infinite loop, stack overflow	Confirmed
14	GRUB	Storage, file parser	Off-by-one access, heap overflow	Confirmed
15	Limine	Storage, file parser	Missing sanity check, null-pointer dereference	Patched
16	Limine	Storage, file parser	Logic bug, heap overflow	1-day
17	Limine	Storage, file system	Missing sanity check, divide by zero	Patched
18	Limine	Storage, file system	Missing sanity check, divide by zero	Patched
19	barebox	Network	Lack of length check, heap overflow	Patched
20	barebox	Network	Lack of length check, heap overflow	Patched
21	barebox	Network	Lack of length check, heap overflow	Patched
22	barebox	Network	Lack of length check, heap overflow	Patched
23	barebox	Network	Lack of length check, heap overflow	Patched
24	Easyboot	Storage, file system	Missing sanity check, global buffer overflow	Patched
25	Easyboot	Storage, file system	Missing sanity check, stack overflow	Patched
26	Easyboot	Storage, file system	Missing sanity check, divide by zero	Patched
27	rEFInd	Storage, file parser	Lack of length check, heap overflow	
28	rEFInd	Storage, file system	Logic bug, stack overflow	
29	rEFInd	Storage, file system	Missing sanity check, divide by zero	
30	rEFInd	Storage, file system	Missing sanity check, divide by zero	
31	rEFInd	Storage, file system	Missing sanity check, divide by zero	
32	rEFInd	Storage, file system	Missing sanity check, divide by zero	
33	rEFInd	Storage, file system	Missing sanity check, divide by zero	
34	Das U-Boot	Storage, file system	Implementation error, heap overflow	Patched
35	Das U-Boot	Storage, file system	Missing sanity check, divide by zero	
36	Das U-Boot	Storage, file system	Logic bug, heap overflow	
37	Cloverbootloader	Storage, file parser	Lack of length check, null-pointer dereference	Patched
38	Cloverbootloader	Storage, file parser	Lack of length check, heap overflow	Patched
39	Cloverbootloader	Storage, file parser	Implementation error, use-after-free	Patched

4

Embedded System Firmware: Adaptive Interrupt Driven Fuzzing

4.1 Overview

Real-Time Operating Systems, commonly adopted by embedded system developers, are relatively simple but effective low-level software. Compared with rich OS such as Linux, their advantage is the lightweight size and efficient communication with peripherals.

Over the past decade, embedded devices such as factory robots, medical devices, satellites, and smart fitness bands have become widespread. Despite advances in firmware development, security threats and software faults persist. Developers often prefer memory-unsafe languages such as C and C++for low-level hardware manipulation, but these languages also introduce memory corruption vulnerabilities due to their inherent direct memory access features.

Fuzzing has proven to be an effective method for discovering vulnerabilities in RTOS firmware images. Embedded devices with low performance and slow speed are not inherently designed for fuzzing. Therefore, running firmware in an emulated environment and simulating its peripherals' behaviors—a technique known as *re-hosting*—is a promising approach to improve testing. Tools like P2IM [74] and µEmu [248] attempt to model peripheral behavior by extracting information from the MCU documentation or using symbolic execution. Unfortunately, these techniques are unstable and imprecise. The Fuzzware framework [177] models the MMIO (memory-mapped I/O, the way that the firmware communicates with the peripherals) access into several categories, such as *bitextract*, *passthrough*, and *constant-value*. This MMIO modeling efficiently reduces the input overhead. For instance, the constant-value model only accepts a single specific input (i.e., the input overhead is 1), reducing fuzzer effort on mutating that MMIO access. Hoedur [178], another advanced firmware fuzzing framework, divides a single fuzzing input into multiple streams based on the MMIO access context. This so-called *multi-stream fuzzing* prevents the "avalanche effect", where a value meant for one MMIO access is mistakenly consumed by another. Recently, SafireFuzz [186] proposed binary rewriting of ARM Cortex-M firmware to make it compatible with high-performance ARM Cortex-A processors, aiming to accelerate fuzzing speed. While Fuzzware and Hoedur solely focus on the MMIO input, SafireFuzz requires the presence of a hardware abstraction layer (HAL) to inject the fuzz input into both MMIO and DMA (Direct Memory Access).

However, several hard-to-bypass obstacles still block the way for fuzzers to achieve higher code coverage in the firmware fuzzing process. Despite advances in firmware fuzzing, including state-of-the-art frameworks, the challenge of accurately triggering interrupts remains unresolved: If interrupts are triggered incorrectly, the firmware may crash or get stuck, even at an early stage. Thus, a proper mechanism for triggering and handling interrupts is a crucial yet under-researched aspect of firmware fuzzing. P2IM models all peripheral behaviors, including interrupts, by analyzing the documentation, but this approach lacks precision. Both Fuzzware and Hoedur use a round-robin mechanism for interrupt triggering by default, which involves activating an interrupt at regular, fixed time intervals. This process starts with the first enabled interrupt, triggers it, and then triggers the next enabled one. Once all enabled interrupts have been triggered, the cycle repeats with the first enabled one. Fuzzware and Hoedur also support an advanced fuzz-mode interrupt triggering mechanism, which triggers

interrupts depending on the fuzzing input. SafireFuzz uses indirect call-level counters and manual clock-update hooks. The design is similar to the round-robin mechanism for timer interrupts from a high-level perspective. While these mechanisms are effective for firmware with straightforward interrupt-triggering conditions, they fall short in more complex scenarios.

Unfortunately, interrupt processing in real-world firmware is often complicated in practice. For example, certain interrupts may be enabled but not yet ready to be triggered because the associated data, such as pointers, are not fully initialized. The round-robin and fuzz-mode interrupt triggering mechanisms do not take the status of the interrupt into account. Therefore, if the fuzzer triggers an interrupt *before* the data is initialized, this can lead to an unexpected crash that brings the fuzzing process to a halt. Even after all data is initialized, triggering some interrupts can cause the firmware to reset or get stuck in an infinite loop. These interrupts should never be triggered, as they hinder the fuzzing progress. In addition, the round-robin and fuzz-mode mechanism triggers interrupts at regular intervals, but this approach has disadvantages: If the interval is too small, the fuzzer will constantly interrupt the execution of the firmware, while too large an interval will cause the firmware to wait for the interrupt. To summarize, we need to answer three questions when dealing with the interrupt-triggering problem:

- *When* should the interrupts be triggered?
- *How often* should the interrupts be triggered?
- *Which* interrupts should be triggered?

We have found that the key observation to answer the above questions is the *run-time state transition* of the firmware. The firmware goes through an initialization phase at boot time and then transitions to a *processing state* where it processes inputs. Once the processing of the inputs is complete, it enters a *waiting state* in which it awaits certain asynchronous events or new inputs, which are usually delivered via interrupts. The cycle then repeats itself when it returns to the processing state. The interrupts should only be triggered when the firmware is in the waiting state, without intervening during the processing state. By automatically analyzing the *interrupt service routine* (ISR), we can pinpoint which interrupts are able to transition the firmware from the waiting state to the processing state. We refer to these as *effective interrupts*. When the firmware is in a waiting state, we selectively trigger only the effective interrupts that can cause a transition to the processing state. We also make sure that each interrupt we want to trigger is ready to be triggered, e.g., by checking the initialization of the associated data. AIM [73] proposes a similar interrupt analysis method for firmware testing. It is based on the same idea that ISRs can influence the behavior of the firmware. However, AIM does not model the firmware run-time state and cannot reveal the relationship between the run-time state and the interrupt. Therefore, it cannot answer the three questions accurately. In addition, the overall design and implementation of AIM is based on symbolic execution, which significantly affects the analysis speed.

In this thesis, we introduce AidFuzzer, an ***A***daptive ***I***nterrupt-***D***riven Fuzzing framework that provides a proper interrupt triggering mechanism for firmware fuzzing.

AidFuzzer identifies effective interrupts, triggers interrupt on demand and only triggers the interrupts that are required by the firmware. To evaluate the performance of AidFuzzer, we compiled a collection of 10 open-source firmware projects based on the ARM Cortex-M processor. This dataset includes open-source Github projects and popular RTOS examples, such as RT-Thread and Apache Mynewt-OS. Our experimental results show that AidFuzzer performs better than the state-of-the-art tools Fuzzware, Hoedur, and SafireFuzz in handling complex interrupt scenarios. In addition, we found eight previously unknown vulnerabilities in these open-source projects.

4.2 Arm Cortex-M NVIC Interrupt

Since we evaluate AidFuzzer on ARM Cortex-M-based firmware targets, and we focus on the interrupt handling problem, we now provide a brief introduction to the ARM Cortex-M nested vector interrupt control (*NVIC*). It enhances the reader's comprehension of our interrupt-triggering algorithm. While the specific implementation details may differ across various processors, the fundamental concepts remain consistent. Furthermore, we illustrate the QEMU soft-NVIC implementation, as it is tightly coupled to our prototype implementation.

4.2.1 IRQ and Interrupt Vector Table

An interrupt request (*IRQ*) is an asynchronous event typically initiated by peripherals. Exception handling by the processor follows a similar path to that of an interrupt, but exceptions are internally generated by the processor, such as encountering an illegal instruction or a division-by-zero fault. This chapter specifically focuses on asynchronous interrupts originating from peripherals. We will use IRQ triggering and interrupt triggering interchangeably in the chapter, as they both refer to the same concept: A device sends an asynchronous interrupt request to the processor.

When an interrupt occurs, it alters the control flow from the current processor execution context (referred to as *thread mode* in ARM Cortex-M) to the interrupt handling context (referred to as *handler mode*). Upon detecting the incoming interrupt signal, the processor automatically preserves the current register context by saving it to the stack. Subsequently, it retrieves the IRQ number and the corresponding Interrupt Service Routine (*ISR*) address from the *interrupt vector table* and starts executing the ISR.

The interrupt vector table is an array of function pointers in memory that use the IRQ number as an index to access its elements. For instance, consider an interrupt vector table located at address 0x20000000 and an IRQ with a number of 0x20, the processor finds the corresponding ISR at address 0x2000000080 (calculated as 0x20000000 + 4 * 0x20, as a function pointer in the ARM Cortex-M is 4 bytes in size). The processor then loads the memory content from 0x2000000080 into the Program Counter (PC). Upon completing the ISR, the processor loads the *EXC_RETURN* [155] value, previously saved in the link register during the context switch, into the PC, indicating an interrupt exit. The processor automatically restores the previous context and resumes execution from thread mode.

Each interrupt has a corresponding priority, and when multiple interrupts occur simultaneously, the processor gives priority to the one with the highest priority. It is noteworthy that this chapter excludes the handling of nested interrupts and tail interrupts, as interrupts are only triggered in thread mode, and only one interrupt is triggered at a time in our implementation.

Note that the interrupt vector table is subject to dynamic re-basing, and its elements can be overwritten during run-time. For instance, the firmware might alter the table address from 0x20000000 to 0x30000000 or overwrite the memory content at address 0x2000000080. Consequently, different ISRs can be employed in such scenarios to handle the same IRQ. This flexibility in re-configuring the table allows for dynamic adjustments to the interrupt handling process, enabling the system to adapt to changing requirements or respond to specific run-time conditions.

4.2.2 NVIC Configuration

NVIC is mapped as a Memory-Mapped I/O (*MMIO*) region and can be configured by writing to this designated memory space. As an example, the vector table base address can be configured by the firmware. When the firmware requires NVIC configuration, it simply writes to the pertinent field within the NVIC data structure. In this chapter, we focus on the following NVIC configurations:

- *Enable/Disable IRQ*: An array of bits indicates the enable/disable status of an IRQ. NVIC supports up to 240 interrupts [155]. However, besides the reserved interrupts that are enabled by default, only a number of the IRQs are used and enabled by the firmware. The IRQ can only be used by the peripherals and triggered if it is enabled.
- *Interrupt Vector Table Base*: Writing to this field changes the table base address. The processor fetches the subsequent ISR address based on the updated value.
- *IRQ Pending*: An array of bits indicates pending IRQ requests. Writing to this field pends a corresponding IRQ, waiting to be handled by the processor. Note that setting a pending bit to this array does not mean that this IRQ will be served immediately, it also depends on several other conditions: a) If interrupt handling is enabled globally by the processor. b) If the specific IRQ is enabled in the NVIC configuration.

One bit in the Current Program Status Register (*CPSR*) for ARM processors indicates the global interrupt enable/disable status. Setting/clearing this bit allows/prevents all the interrupts from being served.

4.2.3 Types of NVIC Interrupts

IRQ numbers 1-15 are usually reserved for core ARM Cortex-M processor functionalities such as reset, system-call, and hard fault, which are not triggered by peripherals. One exception is the SysTick IRQ: Firmware may rely on it to accomplish its functionalities such as task scheduling. Peripherals customize other IRQs mainly to handle the following

situations: a) New data is available either from DMA buffers or from device registers. b) Output data is consumed or processed by the peripherals. c) A specific time interval has passed. d) There is an internal status change in the peripherals. e) Other unexpected event happens. For example, a typical 8250 UART device [1] ISR either reads a character from the Receiver Buffer register when a new input character arrives or writes a cached character to the Transmitter Holding Buffer register when the device is ready to consume more characters.

4.2.4 QEMU NVIC Implementation

QEMU maps the NVIC as an MMIO region and maintains the NVIC status in its internal data structure (i.e., this internal structure belongs to QEMU instead of the emulated fimware). When the virtual machine accesses this memory region, a corresponding QEMU function is called. For example, when a virtual machine writes to the enable/disable IRQ bit array, the function *nvic_sysreg_write* serves this operation. The IRQ status will be updated in this function, and the status is persisted in *NVICState* structure. QEMU provides a function called *armv7m_nvic_set_pending* for the peripherals to trigger an interrupt. In this function, QEMU checks if the IRQ is allowed to be triggered, calculates the priorities among all the pending IRQs, makes the highest priority IRQ active, and notifies the execution thread about the new interrupt request. The execution thread regularly checks if there are pending interrupt requests. If any, it performs the context switch and starts executing the ISR.

4.3 Firmware Fuzzing Interrupt Triggering Analysis

4.3.1 Running Examples

In this section, we use concrete examples to show why triggering interrupts is crucial for firmware fuzzing. All examples are adapted and simplified based on real firmware. In the first example, an infinite loop dummy ISR serves an IRQ during the initial stage to prevent the firmware from running into an unintended state. The second example shows a watchdog ISR that halts the system when an unexpected error happens. These two IRQs should never be triggered since they immediately stop the firmware running, thus hindering the fuzzing process. In the third example, a UART ISR extracts characters from a ring buffer and outputs them to the terminal; it should be triggered only when necessary. By using these three examples, we illustrate what concrete problems should be considered when triggering an interrupt.

Listing 4: A commonly used dummy ISR

```
void dummy_isr() {
    while(1) { ; }
}
```

An infinite loop as shown in Listing 4 is commonly used by the firmware to implement a dummy ISR during the initial stage. The firmware usually first enables the IRQ and then initializes the actual ISR later. In practice, there is even firmware that activates

certain IRQs and allows them to be served by dummy ISRs during the entire run-time. Moreover, some developers use a function pointer in the ISR as a dummy function and initialize the pointer afterwards. The firmware directly de-references the pointers without NULL checking since the IRQ will only be triggered after the data are properly initialized in a real environment. However, if the interrupt is triggered too early before it is fully initialized, the firmware gets stuck or crashes, preventing the fuzzer from making any progress in fuzzing. The firmware works well in the real environment because no peripherals use this interrupt before they are fully initialized. The round-robin and fuzz-mode interrupt mechanisms will eventually trigger it and hinder the fuzzing process.

Listing 5: A simplified watchdog ISR

```
void watchdog_isr() {
    trace_watchdog_isr_event();
    if(wdt_handler)
        wdt_handler();
    system_hal();
}
int main() {
    do_something1();
    watchdog_tickle();
    do_something2();
    watchdog_tickle();
}
```

Even after initializing the ISR and the data, some IRQs should never be triggered during fuzzing. Listing 5 shows a watchdog ISR. The firmware needs to regularly tickle the watchdog to prevent it from triggering a watchdog interrupt. In the watchdog ISR, the watchdog interrupt event is logged, the handler provided by the firmware is called to perform the cleaning task, and finally, the system is reset. The implementation of *system_hal* depends on the specific board. It may simply go into an infinite loop or execute a breakpoint instruction. The watchdog ISR is used to prevent the firmware from corrupting the data when something unexpected happens. Normally, this interrupt is not expected to be triggered during regular execution, as the watchdog is regularly tickled. However, in a round-robin or fuzz-mode interrupt mechanism, it will be triggered at some time, thus hindering the fuzzing process.

Listing 6: A simplified UART ISR

```
struct ring_buffer output_buffer;
void uart_tx_isr() {
  if(uart_ready() && !empty(&output_buffer)) {
    char c = dequeue(&output_buffer);
    uart_write(c); // write to uart register
  }
}
void uart_tx_string(char *s) {
  while (*s) {
    while(full(&output_buffer)) {
        ;  // stuck here
```

```
    }
    enqueue(&output_buffer,*s);
    s++;
  }
}
void main() {
  uart_tx_string("before do something");
  do_something();
  uart_tx_string("hello world");
}
```

Unlike the watchdog interrupt, Listing 6 shows a simplified UART ISR that the firmware relies on to accomplish its console output functionality which should be triggered when necessary. Listing 6 defines a ring buffer *output_buffer*. The functions *full* and *empty* check whether the buffer is full or empty. When the UART interrupt gets triggered, it checks if the device is ready to consume more characters and, if so, it fetches a character from the ring buffer and writes it to the console. The function *uart_tx_string* is used to output a message string. It keeps looping until all the string characters are inserted into the ring buffer. In the *main* function, the firmware outputs several messages. However, to avoid getting stuck in the *uart_tx_string* function, the UART interrupt must be triggered to consume the characters from the ring buffer. The round-robin or fuzz-mode mechanism can work in this situation. However, a short trigger time interval interferes with the execution of the function *do_something*, while a long interval makes the firmware busy checking the ring buffer in the function *uart_tx_string*.

Recall the three questions we aim to answer in this chapter:

1. *When* should the interrupts be triggered?
2. *How often* should the interrupts be triggered?
3. *Which* interrupts should be triggered?

Learning from the above examples, we propose the following heuristics:

1. Interrupts should be triggered only after the ISR and the data are initialized. We call this IRQ status *ready*. (*When*)
2. Interrupts should only be triggered when the firmware needs them. We call this firmware run-time state *waiting*. For example, when the firmware keeps checking the status of the ring buffer in the simplified UART ISR example, it is in a waiting state. (*How often*)
3. Only the interrupts whose ISR can change the firmware run-time state from waiting to not waiting should be triggered. We call this IRQ type *effective*. For example, the ISRs in the first two examples make the firmware get stuck and are thus not effective IRQs. However, the UART ISR in the third example lets the firmware escape from the waiting state and continue running, thus it is an effective IRQ. (*Which*)

In summary, to solve the interrupt-triggering problem, we need to identify the IRQ status (ready or unready), IRQ types (effective or ineffective), and the firmware run-time state (waiting or not waiting).

Solving the problem of interrupt-triggering is a challenge in practice due to the complexity of the interrupt design. Identifying the IRQ status, IRQ types, and firmware run-time state is a non-trivial task.

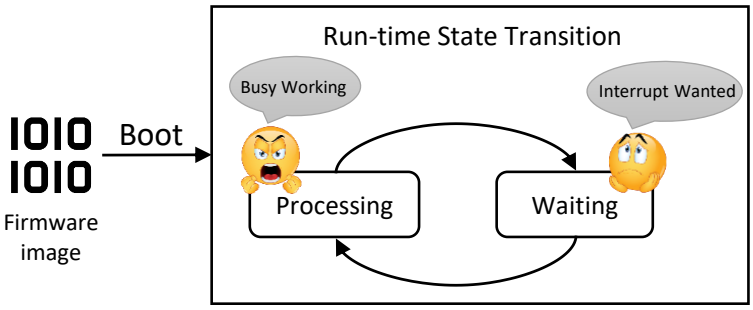

Figure 4.1: Firmware run-time state transition cycle

4.3.2 Challenges

The complicated interrupt design in real-world firmware poses two main challenges:

- When the processor serves an IRQ, it retrieves the interrupt vector table base address and indexes the ISR using the IRQ number. The vector table is subject to dynamic re-basing, and its elements can be overwritten at run-time. Thus, a single IRQ number can be served by multiple ISRs. Besides, function pointers are widely used in ISRs. A function pointer can point to different functions at firmware run-time. An ineffective IRQ may become an effective one after any of these conditions get changed. Therefore, the type and the status of the IRQ cannot be statically determined. We summarize this challenge as *run-time data dependency*.
- Manually analyzing the firmware to determine the run-time state requires a non-trivial amount of work. The firmware does not enter a waiting state at a fixed time interval but is highly dependent on the program logic. The firmware requires different interrupts in different waiting states. Statically analyzing the entire firmware to determine when the firmware enters a waiting state is a tedious task, as static analysis is not scalable. We summarize this challenge as *state recognition*.

4.3.3 Insights

To solve the run-time data dependency challenge, we monitor and intercept the changes of the interrupt vector table base, vector table entries, and the function pointers used in the ISR during the whole fuzzing campaign. If there is any change, we extract the ISR address from the vector table, dump the firmware registers and memory, analyze it, and save the analysis results in an IRQ model database. When an update is detected, e.g., when a function pointer is overwritten with a new value, we first try to find the model in our IRQ model database. If we find the corresponding model for the update, we apply it; otherwise, we perform a re-analysis. In this way, we always keep using the latest IRQ model.

For the state recognition problem, we observe that most firmware share a common run-time transition cycle. As shown in Figure 4.1, the firmware boots itself and then goes into an infinite processing-waiting loop. In the processing state, it does not require any interrupts and is busy processing data. In the waiting state, it requires interrupts

to change its state back to processing again. We have the following key observation:

> The effective IRQs change the firmware run-time state from waiting to processing by modifying global objects.

For instance, in the simplified UART ISR from Listing 6, the ISR *uart_tx_isr* makes the firmware continue running by changing the global object *output_buffer* status from full to not full. Besides, we observe that firmware widely uses specific instructions or infinite loops to enter a waiting state as well. Hence, we conclude that the firmware enters a waiting state if one of the following conditions is satisfied:

1. The firmware explicitly enables the global interrupt by setting the bit in CPSR in a frequent manner.
2. The firmware executes the *Wait for Interrupt* (WFI) or *Wait for Event* (WFE) instruction. These instructions allow the core to enter a low-power mode and stop executing code.
3. The firmware enters an infinite loop.
4. The firmware constantly checks the global objects whose value can be modified in an ISR.

With these assumptions in mind, we conclude that the firmware only requires interrupts when it is in a waiting state, and the ISRs will modify global objects to change the firmware run-time state from waiting to processing. Hence, we trigger interrupts in the following manner: When the firmware enters a waiting state via the first three conditions, we trigger all the effective interrupts one by one since we have no hints indicating which interrupt it requires. When the firmware enters a waiting state via the fourth condition, we trigger the corresponding interrupt whose ISR can change the global objects' value.

Our findings closely align with our investigation of firmware samples collected from various RTOS. More specifically, we analyzed 110 firmware samples and found that 83% of them followed our run-time state transition cycle observation. For more information about the investigation we conducted, please refer to Section 4.5.

4.4 Interrupt-Driven Firmware Fuzzing Design

We now discuss the threat model we use in this paper and then present the design and implementation of our approach.

4.4.1 Threat Model

In this paper, we assume that the attacker has full control over the MMIO data. The firmware accepts peripheral inputs (e.g., network packets, temperature, and console input characters) either from the MMIO registers or from the DMA buffer. We disregard the DMA input and focus on the data read from MMIO. We do not assume that the interrupts are controlled by the attacker, as they usually cannot be configured by an attacker. We make no assumptions about the image symbols, source code, and documentation of the firmware. However, like other re-hosting systems, we assume that we have full knowledge of the board's memory layout on which the firmware runs.

4.4.2 High-Level Overview

Figure 4.2 illustrates the design overview of AidFuzzer. It mainly consists of three components: the emulator, the IRQ modeling engine, and the fuzzing engine. The emulator maintains an emulated environment, including CPU registers and virtual memory, for re-hosting the firmware. It translates the ARM assembly code into native code and executes it. The IRQ manager triggers interrupts when the firmware enters the waiting state. The IRQ modeling engine extracts the firmware context and analyzes the ISR at a specific point. Upon completion of the analysis, the modeling results are saved in the IRQ model database, which is subsequently retrieved and used by the IRQ manager. The fuzzing engine, like other firmware fuzzers, supplies fuzzing input via MMIO and obtains coverage feedback from the emulator to guide the fuzzing process. In the following sections, we discuss the detailed functionalities of each part, in particular how the interrupt trigger mechanism works in the system.

4.4.3 IRQ Modeling Engine

Recall that we designate an IRQ as effective if it changes the firmware runtime state to processing (i.e., its ISR alters global objects), and we trigger it after its status becomes ready. The main goal of the IRQ modeling is to determine the type and status of an IRQ. The type and status can change during the runtime, therefore, we need to update the IRQ model when necessary. Specifically, we collect the following information during the IRQ modeling process.

a) Does the ISR modify any global objects? If so, collect the addresses to which the values are written.
b) Does the ISR use any function pointers? If so, collect the addresses from which the pointers are loaded.
c) Does the ISR de-reference any null pointers without checking? If so, collect the addresses from which the pointers are loaded.
d) Does the firmware always get stuck in the ISR, e.g., in an infinite loop?

We use the modified global objects information to identify the types of the IRQ (effective or ineffective), the null pointer de-reference and getting stuck information to

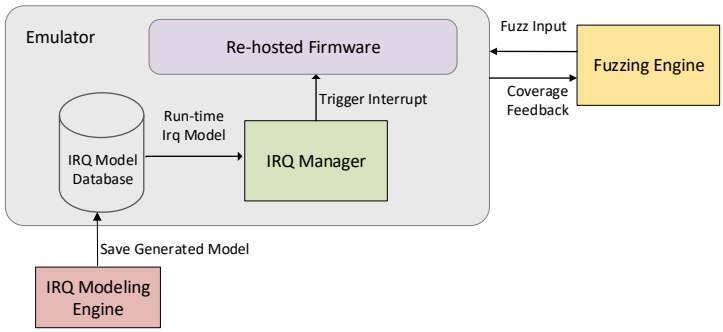

Figure 4.2: Design overview of AidFuzzer

identify the status of the IRQ (ready or unready), and the function pointer information to update the type and status of an IRQ. After finishing the modeling, the results are saved in the IRQ model database.

4.4.3.1 IRQ Modeling Workflow

The IRQ modeling engine takes firmware register values and a memory dump in combination with the memory layout configuration file as inputs (note that we assume that the memory layout is available in our threat model). The modeling engine symbolically executes the ISR by using angr [193]. During execution, the engine intercepts the memory access operations to collect the aforementioned information. However, since all data in the context dump is concrete, the global memory needs to be symbolized to explore as many paths as possible. An exception is the pointer, where further information is required to fetch more data and explore indirect functions. Specifically, we perform the following operations in the symbolic execution interception.

Interceptions. For memory read operations where the addresses from which the data is to be loaded are in the global memory space, the engine symbolizes their value and creates a mapping between the symbolic value and its concrete memory data. If the addresses are symbolic, it checks whether their corresponding concrete data is 0, and it reports a null pointer de-reference if the value is not constrained to a non-zero value. If a function pointer is used in an indirect branch, it resolves its symbolic value to concrete data so that the symbolic execution keeps running. If the function pointer is 0, it reports a function pointer usage. For memory write operation, it checks if the addresses are located in the global memory space, and if so, it reports a global object modification. For each piece of information, the engine collects the addresses at which the data was loaded or saved. After the symbolic execution is completed, it checks whether one of the paths can reach the end of the ISR; if not, it reports that it is stuck.

Special control register handling. The processor automatically loads the IRQ number into IPSR before entering the ISR. In certain firmware, a unified ISR wrapper is implemented for all IRQs, using the IPSR register value as an index to retrieve the actual ISR function pointer from a function pointer array. When conducting symbolic execution and encountering reads from this register, we provide the concrete IRQ number. For other control registers, we opt to symbolize their values to facilitate the exploration of additional execution paths.

Path explosion handling. To mitigate path explosion, we first explore the newly discovered basic blocks and set a timeout for the loops when modeling the ISR. We set a two minute timeout for the AidFuzzer prototype.

4.4.3.2 IRQ Modeling Example

We take the simplified UART ISR as a modeling example. During the symbolic execution, the entire global memory and MMIO data are symbolized. Therefore, the function *uart_ready* can return both true and false, and the ring buffer could also be empty and full. If one of the two conditions is unsatisfied, the execution ends, making it possible to reach the end of the ISR. If the two conditions are satisfied, the execution enters the `if` branch. When it tries to write the *output_buffer* object, the engine infers that it is

a global object, and therefore it collects the address of the field that is being written to. Finally, we conclude that the UART ISR has the following modeling results: It modifies a global object *output_buffer*, it does not contain null pointer dereferencing, and does not cause the firmware to get stuck. Thus, the UART IRQ is effective and ready.

4.4.4 Emulator

We implement our emulator based on QEMU [20]. The emulator functions as a dynamic runtime environment for the firmware, with the original ARM assembly code dynamically compiled into intermediate language code (TCG code in QEMU). This intermediate code is further translated into native code and executed. The physical addresses of the re-hosted firmware are translated into native addresses through the use of the softMMU, enabling interception of every memory access. To feed the fuzz input, we implement an ARM Cortex-M-based board that supports full customization for the memory regions. Originally, the emulated peripherals are responsible for triggering interrupts; however, the IRQ manager takes over the interrupt triggering and can decide when and what interrupts are to be triggered with the help of the IRQ model database. After each fuzzing run, the emulator restores the re-hosted firmware and IRQ manager state. The coordination between the IRQ manager and the re-hosted firmware is explained in more detail next.

4.4.4.1 IRQ Manager

State monitor. Recall that we established four conditions, and if any of them is satisfied, the firmware enters the waiting state. Upon the satisfaction of any of these conditions, our callback function is invoked. In this callback function, the IRQ manager determines whether and what interrupts are to be triggered. We explain how AidFuzzer checks the satisfaction of the conditions and infers the firmware runtime state:

1. The firmware frequently enables the interrupt. We monitor the global interrupt enable/disable state by intercepting the execution of all *CPSIE I* instructions. This instruction sets the CPSR bit so that the processor can serve the interrupts. When writing to this register, our callback function is invoked.
2. The firmware executes WFI or WFE instructions. We intercept all the WFI and WFE instructions. Once the firmware executes these two instructions, the firmware stops execution and our callback function gets invoked.
3. The firmware enters an infinite loop. We search for all infinite loops in the firmware before fuzzing. Beginning with each branch instruction, we conduct symbolic execution of the subsequent instructions. If we determine that the execution can reach the same branch instruction without encountering an opportunity to exit the loop, we categorize it as an infinite loop. For each basic block initiating an infinite loop, we register a callback function. Consequently, should the firmware enter an infinite loop during runtime, the IRQ manager receives a notification.
4. The firmware constantly checks the global objects whose values can be modified in an ISR. We set memory read breakpoints to all the global objects whose values can be modified in the effective IRQs' ISRs. Whenever any of the global objects is read by the firmware, our callback functions are invoked.

We set a counter for each condition. If any callback function gets invoked, we increase the corresponding counter by one. Once a counter surpasses a predefined threshold, we trigger the specific interrupts. For example, we set the counter threshold for condition 1 to 10. If the firmware enables the interrupt 10 times, we trigger all the effective IRQs one by one and reset the counter to 0. Note that we trigger all the effective IRQs one by one when the first three condition counters surpass the threshold, while we only trigger the corresponding IRQ in the fourth condition. In AidFuzzer, we set the enable interrupt counter threshold to 32, the WFI/WFE instruction counter threshold to 1, the infinite loop counter threshold to 7, and the global object check counter threshold to 10 according to our empirical analysis.

To avoid recursive interrupt triggering, we do not increase the counter when the firmware is handling an exception, which means when the firmware is executing an ISR, even if the conditions are satisfied, no counter is increased.

IRQ triggering. Once the IRQ manager decides to trigger an interrupt, we set a bit in the NVIC IRQ pending field. Specifically, we call the QEMU function *armv7m_nvic_set_pending* with the IRQ number as an argument. The emulator checks the pending request and does the actual interrupt handling.

IRQ model switching. Maintaining the latest IRQ model is crucial. We achieve this by registering several event hooks to trigger model updating. Initially, we generate a model whose status is not ready for all the IRQs. During the fuzzing process, we analyze the ISR, save the results to the model database, and fetch the model when the currently used model needs to be updated. Specifically, we update the IRQ model in the following situations: a) When an IRQ is enabled. IRQ manager requests the IRQ modeling engine to analyze the newly enabled IRQ and switch the IRQ model to the generated one. b) When the vector table is re-based. We check all the enabled IRQ vector table entries to see if it is a new value. If so, the IRQ manager requests a re-analysis and switches to the new model. c) When an enabled IRQ table entry is overwritten with a new value. We re-analyze the ISR and switch to the new model. d) When a function pointer is overwritten with a new value. We re-analyze the ISR and switch to the new model. We assign a unique ID to each generated model according to its ISR address and the function pointer values. When an existing model is available in the database, we switch to the existing one instead of requesting a re-analysis.

4.4.4.2 Snapshot

We use snapshots to speed up the fuzzing process. When the firmware executes the first MMIO read instruction, we take a snapshot, as we rely on the assumption that the firmware's control flow will not change if it is not affected by the MMIO input. This holds for almost all the firmware, and it works well for all our test cases. Besides the memory and the registers, we snapshot and restore the internal NVIC state as well.

4.4.5 Fuzzing Engine

Multi-stream fuzzing input. We adopt Hoedur's [178] multi-stream input and Fuzzware's fine-grained input model [177]. Whenever the firmware tries to read from MMIO memory, the emulator generates a corresponding ID by calculating the instruction

and MMIO address hash result. If it is the first time it encounters the access ID, it invokes the Fuzzware interface to generate an input model for the ID, then the emulator notifies the fuzzer about the newly generated input stream. For each newly generated stream, the fuzzing engine assigns a random length of data for it. The firmware consumes the stream data from the MMIO read when the stream is not exhausted; otherwise, it reports an out-of-stream exit. For now, we ignore the MMIO writes and redirect them to a dummy function.

Coverage feedback. Edge coverage is widely adopted in fuzzing [147] [127] [81]; however, the asynchronous interrupt leads to noisy coverage. An edge that starts from the current basic block to the beginning of the ISR does not exist. To eliminate the noise, we choose to use basic block coverage. We intercept every basic block execution. Before the basic block gets executed, we increase the corresponding coverage byte by one.

Crash detection. We do not have sanitizers integrated into our emulator; therefore, we only detect invalid memory access crashes. When an invalid memory access happens, such as a null pointer de-reference when address 0 is not mapped, our exception hook is notified and gets called with the exception code as an argument. We check the exception code to see if it is a real crash since, for example, a syscall is also regarded as an exception in ARM Cortex-M. Moreover, the firmware may write to an NVIC field to reset the system. Fuzzware regards this as a crash; however, we filter out such cases since they do not incur a security issue.

4.5 Evaluation

In this section, we comprehensively evaluate AidFuzzer, demonstrating its effectiveness on firmware fuzzing. We aim to answer the following research questions.

RQ1: Is AidFuzzer more effective compared to the previous works for fuzzing firmware in terms of coverage and bug finding?

RQ2: How sound is the IRQ modeling?

RQ3: How computationally expensive is the implemented IRQ modeling?

RQ4: Does the IRQ modeling perform better than existing methods?

4.5.1 Experiment Setup

Experiment settings. We performed our experiments on a 104 core Intel Xeon Gold 5320 CPU @ 2.20GHz with a 252 GB RAM server running a Ubuntu 22.04.1 LTS OS. We evaluated our prototype against the two state-of-the-art firmware fuzzers Fuzzware and Hoedur. For each target, we gave each fuzzer one physical CPU core. Moreover, we evaluated against SafireFuzz.

Target firmware selection. Our evaluation targets consist of 10 firmware projects. We collected them from open-source GitHub projects, well-known RTOS examples, and the targets that have been used by previous evaluation experiments [248]. The details can be found in Table 4.1.

[1]Microchip Advanced Software Framework

Table 4.1: Number of basic blocks, board, and the OS/framework information of the firmware we collected.

	Bbls	Board	OS/framework
Blehci [83]	5441	nrf52840	Apache Mynewt
AnnePro2-Shine [50]	1117	AnnePro2 keyboard	ChibiOS
TauLabs [206]	4644	pipxtreme	ChibiOS
3Dprinter [204]	8032	Marlin printers	bare-metal
bcn_rfd_ncp [148]	3590	Atmel SAM	ASF[1]
coord_ncp [148]	4247	Atmel SAM	ASF
mac_no_beacon_sleep [148]	2940	Atmel SAM	ASF
nobcn_ffd_ncp [148]	3510	Atmel SAM	ASF
sam4l_qtouch [148]	1799	Atmel SAM	ASF
nmea_parser [209]	8415	STM32F411RE	RT-Thread

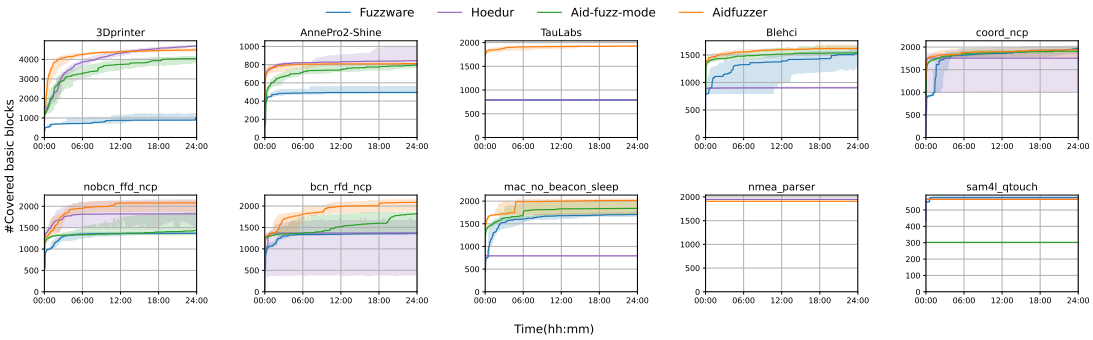

Figure 4.3: Basic block coverage achieved by Fuzzware, Hoedur, AidFuzzer-fuzz-mode, and AidFuzzer over the course of 24 hours for 10 times. We plot the median and confidence interval.

Evaluation metrics. We evaluate the effectiveness of AidFuzzer in two aspects: a) We count the number of unique basic blocks discovered by fuzzers. We measure if AidFuzzer can discover more unique basic blocks or can discover basic blocks faster. b) We count the number of discovered unique crashes and the number of confirmed vulnerabilities. We measure if AidFuzzer can discover more vulnerabilities while having fewer false positives.

4.5.2 Effectiveness of AidFuzzer (RQ1)

We compare AidFuzzer against the two state-of-the-art tools Fuzzware and Hoedur with their advanced fuzz-mode interrupt triggering mechanism integrated. Each is configured with the default 1000 basic blocks interval. To eliminate the effects originating from the fuzzer, we implemented a fuzz-mode interrupt triggering for AidFuzzer as well for comparison. The implemented AidFuzzer-fuzz-mode has the same interrupt triggering settings, such as interval, as Fuzzware and Hoedur. We fuzzed each target for 24 hours 10 times as recommended by Klees et al. [119] and Schloegel et al. [180]. Figure 4.3 visualizes the median and confidence interval of discovered basic blocks, and Table 4.2 presents the number of unique reported crashes and the confirmed vulnerabilities.

4.5.2.1 Coverage Analysis

As shown in Figure 4.3, AidFuzzer achieved higher and faster coverage than Fuzzware, Hoedur, and AidFuzzer-fuzz-mode for the majority of the targets, while for other targets, AnnePro2-Shine, nmea_parser, and sam4l_qtouch, AidFuzzer achieved similar coverage.

Hoedur had a bug when handling a UART interrupt priority in Blehci and could not continue before the firmware started processing data. For TauLabs, due to ineffective IRQs, Hoedur triggered the watchdog interrupt and got stuck in an infinite loop, and no fuzzing progress was made. We did not plot Fuzzware and AidFuzzer-fuzz-mode for TauLabs, as both fuzzers crashed before discovering any valid input due to triggering unready interrupts. The same problem also happened for target nmea_parser. The nmea_parser SysTick interrupt ISR used uninitialized pointers, which was triggered by Fuzzware and AidFuzzer-fuzz-mode in the early stage. For sam4l_qtouch, AidFuzzer-fuzz-mode kept triggering the SysTick interrupt whose ISR involves an infinite loop before the second interrupt was enabled. An interesting target is mac_no_beacon_sleep. As recommended by Hoedur, we disabled the interrupt triggering time interval because the firmware contains WFI instructions(which means Hoedur only triggers an interrupt when it encounters a WFI instruction). However, the firmware did not reach the instruction during the execution. We identified four conditions that can make the firmware enter a waiting state. However, Hoedur cannot fully identify all of these conditions.

As shown in Figure 4.3, although the interrupts did not make the firmware get stuck or crash in the early stage, triggering the interrupts in a proper frequency was crucial for firmware fuzzing as well and made the fuzzer achieve faster basic block coverage. We take the 3Dprinter as an example to illustrate the reason behind it.

The 3Dprinter firmware takes a string—called GCode instruction—as input. The GCode instruction is read one character at a time when the UART interrupt is triggered. This character is stored in a buffer and is only processed later on when a whole line is read. For the fuzz-mode interrupt triggering strategy, it can be an issue to associate the coverage of the GCode instruction execution with triggering the UART interrupt multiple times in combination with a meaningful GCode instruction input. A fuzzer with such an interrupt strategy may only rarely raise the UART interrupt due to the mentioned coverage feedback disconnect and therefore greatly decreases the chance of reaching deep into the 3Dprinter logic. AidFuzzer identifies the UART interrupt as an effective and ready IRQ. Its ISR modifies the number of characters stored in the buffer. The firmware keeps checking the number of elements in the buffer when it does not receive enough characters from the UART. AidFuzzer's state monitor recognizes the firmware run-time state as waiting and triggers the UART interrupt accordingly. This way, it increases the chances for the firmware to receive a whole line of GCode and continue processing. We noticed that Hoedur achieved higher coverage than AidFuzzer after 14 hours. We found that an interrupt was not triggered by AidFuzzer. This interrupt is enabled after a failure occurs and the firmware enters a throb function. The interrupt is not used by the firmware. AidFuzzer identifies the IRQ as unready as it contains uninitialized pointers that persist throughout the throb function, thus AidFuzzer did not trigger this interrupt, incurring lower coverage than Hoedur.

Besides Fuzzware and Hoedur, we also compared AidFuzzer to SafireFuzz using the

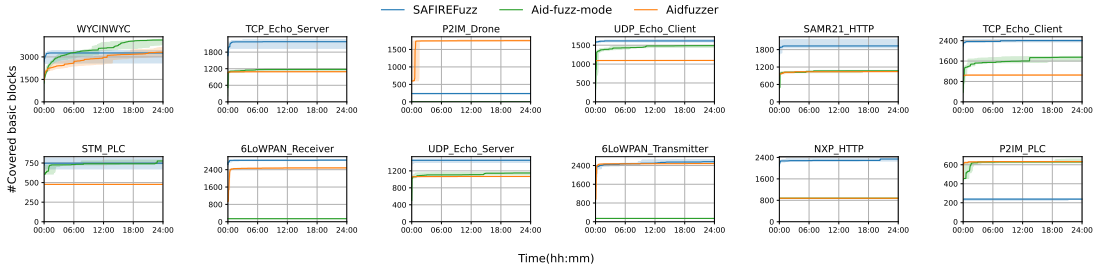

Figure 4.4: Basic block coverage achieved by SafireFuzz, AidFuzzer-fuzz-mode, and AidFuzzer over the course of 24 hours for 5 times. We plot the median and confidence interval. AidFuzzer and SafireFuzz work on different architectures; therefore, we reused the evaluation data from the SafireFuzz experiments.

Table 4.2: Unique crashes, confirmed vulnerabilities, and vulnerability types found by Fuzzware, Hoedur, AidFuzzer-fuzz-mode and AidFuzzer. AidFuzzer discovered more vulnerabilities while reporting 0 false positives.

	Fuzzware		Hoedur		Aid-fuzz-mode		AidFuzzer	
	Reported	Confirmed	Reported	Confirmed	Reported	Confirmed	Reported	Confirmed
Blehci	1	0	0	0	0	0	0	0
AnnePro2-Shine	3	3	3	3	3	3	3	3
TauLabs	1	0	0	0	0	0	1	1
3Dprinter	1	0	1	0	1	0	0	0
bcn_rfd_ncp	1	1	1	1	1	1	1	1
coord_ncp	1	1	1	1	1	1	1	1
mac_no_beacon_sleep	1	1	0	0	1	1	1	1
nobcn_ffd_ncp	1	1	1	1	1	1	1	1
sam4l_qtouch	0	0	0	0	0	0	0	0
nmea_parser	1	0	1	0	1	0	0	0
total	11	7	8	6	9	7	**8**	**8**

12 samples from the SafireFuzz experiments [186]. Since SafireFuzz requires a manual HAL function hook, we reused the data from their paper and plotted them separately. Figure 4.4 shows the unique basic blocks discovered by AidFuzzer, AidFuzzer-fuzz-mode, and SafireFuzz. Notably, six of the firmware samples (6LoWPAN_Receiver, 6LoWPAN_Transmitter, P2IM_Drone, P2IM_PLC, STM_PLC, WYCINWYC) do not use DMA to transfer data, hence these samples are fully supported by AidFuzzer. The other samples use DMA, which is an orthogonal challenge not addressed by AidFuzzer. In the WYCINWYC, P2IM_Drone, 6LoWPAN_Transmitter, and P2IM_PLC samples, AidFuzzer achieves similar or better coverage compared to SafireFuzz. However, AidFuzzer discovered fewer unique basic blocks than SafireFuzz in STM_PLC and 6LoWPAN_Receiver. We observed that the STM_PLC requires a nested interrupt to be triggered, which AidFuzzer does not support. Additionally, AidFuzzer could not successfully recognize the global objects in 6LoWPAN_Receiver due to a bottleneck in symbolic execution. We emphasize that while AidFuzzer and SafireFuzz address orthogonal firmware fuzzing challenges, our methodology could be adapted to enhance the fuzzing efficiency of SafireFuzz.

4.5.2.2 Crash Analysis

AidFuzzer found in a total of 8 vulnerabilities in the 10 firmware targets shown in Table 4.2, including 1 buffer over-read control flow hijacking in TauLabs, 3 buffer

Table 4.3: IRQ Modeling Result. Effective ISRs refer to the ISRs that make the corresponding IRQ effective. Global objects refer to the global objects that the ISRs modify. We count the overall numbers for all enabled IRQs in the firmware. The methods to enter the waiting state are: ❶ constantly enable global interrupt, ❷ execute WFI/WFE instructions, ❸ infinite loop, ❹ constantly check global objects.

	# of enabled IRQs	# of unique ISRs	# of effective ISRs	# of global objects	# of NULL data pointers	# of function pointers	enter waiting state
Blehci	7	10	6	84	4	4	❷❶
AnnePro2-Shine	3	3	3	57	5	1	❸
TauLabs	10	10	8	186	13	11	❸❹
3Dprinter	7	7	6	66	3	13	❹
bcn_rfd_ncp	4	5	3	11	0	16	❶❹
coord_ncp	4	4	3	11	0	15	❶❹
mac_no_beacon_sleep	3	3	2	16	0	45	❶❹
nobcn_ffd_ncp	4	4	3	13	0	19	❶❹
sam4l_qtouch	3	3	2	5	0	1	❶❹
nmea_parser	2	2	2	45	4	25	❶❹

over-writes control flow hijacking in AnnePro2-Shine, and 4 arbitrary memory writes in bcn_rfd_ncp, coord_ncp, mac_no_beacon_sleep, and nobcn_ffd_ncp. It is worth noting that AidFuzzer did not report any false positives. Hoedur, Fuzzware, and AidFuzzer-fuzz-mode correctly reported part of the vulnerabilities; however, they reported false positives as well. We reported all the vulnerabilities to the vendors.

Fuzzware misreported a reset in Blehci as a crash. Due to the implementation bug, Hoedur did not correctly handle the interrupt priority in Blehci and therefore got stuck. Fuzzware, Hoedur, and AidFuzzer-fuzz-mode reported null pointer de-references in 3Dprinter. We manually checked the firmware, and we found that the crashes all happened in an ISR that has not been fully initialized. When a failure occurs, this IRQ is enabled by accident and should not be triggered in the real environment. AidFuzzer successfully identified it as an unready IRQ and did not trigger it. Fuzzware, Hoedur, and AidFuzzer-fuzz-mode all got stuck in the early stage when fuzzing the TauLabs firmware due to ineffective and unready IRQs. AidFuzzer avoided triggering the IRQs that cause the firmware to get stuck and successfully found the buffer over-read vulnerabilities which were not covered by other fuzzers.

AidFuzzer successfully found all the vulnerabilities that the state-of-the-art tools could also find. Moreover, AidFuzzer found more vulnerabilities that were not found by others and reported fewer false positives, which saves manual effort for crash analysis. Note that we only counted the number of unique crashes. Hoedur and Fuzzware reported a large number of false positive crashes in these targets which is why verification requires a non-trivial manual effort. Five CVEs have been assigned to our findings.

4.5.3 Soundness of IRQ Modeling (RQ2)

We collected the following IRQ information for each firmware presented in Table 4.3: the number of enabled IRQs, the number of unique ISRs, the number of effective ISRs (effective ISR means the ISR makes the corresponding IRQ effective), the number of monitored global objects, the number of null data pointers, the number of function pointers, and the mechanisms employed by the firmware to enter the waiting state. The presented table reveals that firmware deploys complex interrupt services for their

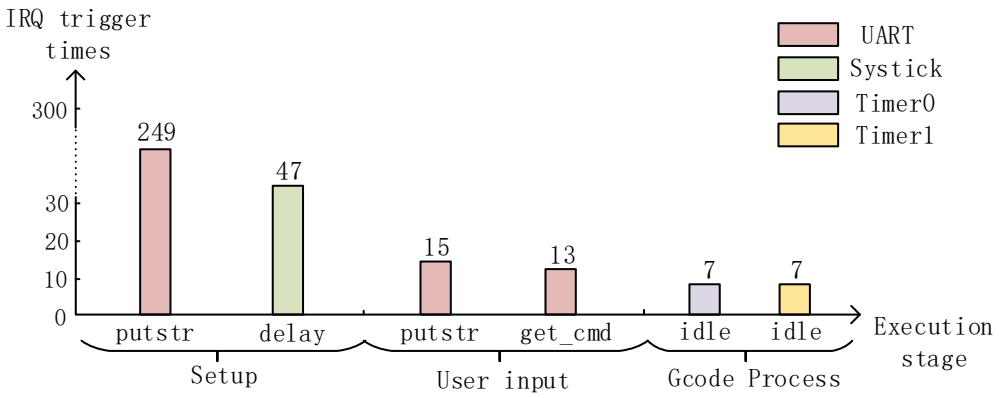

Figure 4.5: Interrupt triggering in 3Dprinter

functionalities. A portion of the ISRs renders the corresponding IRQs ineffective. The range of monitored global objects spans from 5 to 186. The number of function pointers and null-data pointers is consistently below 50 for all targets. Regarding the conditions to enter the waiting state, the firmware utilizes all mentioned four conditions. However, the preference for specific methods varies depending on the implementation of each firmware.

4.5.3.1 Ineffective IRQ Case Study

Taking the Blehci firmware as an example, it enables 7 IRQs and employs 10 ISRs to handle these IRQs. However, 4 of these ISRs render their corresponding IRQs ineffective. One such IRQ is associated with the watchdog. The implementation of the watchdog ISR involves halting the system by executing a break-point instruction and then entering an infinite loop. If triggered, this interrupt results in the termination of the fuzzing process. Another interrupt is tied to the SysTick, and its ISR is an infinite loop. Activation of this interrupt also leads to the cessation of the fuzzing process. Our IRQ modeling result correctly identifies the watchdog and SysTick IRQ as ineffective.

4.5.3.2 Effective IRQ Case Study

Manually verifying the soundness of each modeling result for effective IRQs in the target set can be a cumbersome task. Nonetheless, it is worth illustrating the result with the example of the 3Dprinter. We manually analyzed the firmware code logic and understood the ISR functionalities. Then we checked if the modeling results fit our manual analysis results. To have a clear representation of the AidFuzzer interrupt-triggering for 3Dprinter, we executed the firmware with a discovered input and visualized the type and frequency of interrupts triggered during the execution, as depicted in Figure 4.5. The x-axis is the firmware execution stage, the differently colored columns represent the number and type of the interrupts triggered. We mark the function names where the interrupts get triggered for intuitive understanding.

In the 3Dprinter firmware program logic, IRQ 15 serves the SysTick, and its ISR increases a counter by one. IRQ 53 serves a UART device, and its ISR either consumes one character from the output buffer or reads a character from the UART register into the input buffer. IRQ 44 and 66 serve two timers.

In the setup stage, the UART interrupt is only triggered in the function *usart_putstr*. This function keeps looping until all the characters are consumed. AidFuzzer triggers the UART interrupt to consume the output buffer and lets the firmware continue running. The SysTick interrupt is only triggered in function *delay*. This function is used to set up the temperature management environment and it checks whether the counter exceeds a limit and then continues execution. AidFuzzer triggers the SysTick interrupt to increase the counter and therefore bypass the check quickly. In the user input stage, besides being triggered in the *usart_putstr* function, the UART interrupt is also triggered in function *GCodeQueue_get_serial_commands*. This function checks if there are enough characters in the input buffer and retrieves the characters to a command buffer. During this stage, the firmware reads the input from UART, and thus AidFuzzer triggers the UART interrupt to fill the input buffer. In the last Gcode process stage, two timer interrupts are triggered in the function *idle*. In this stage, the firmware is busy processing the user input, therefore AidFuzzer does not trigger any other interrupts. The firmware waits for some tasks to be completed; consequently, AidFuzzer triggers the timer interrupt to notify the firmware about the completion of the task. The overall IRQ modeling result fits our manual analysis result well.

4.5.3.3 Heuristic Study

To verify if our firmware run-time transition cycle applies to the majority of the firmware, we conducted a heuristic study. Analyzing firmware binaries without access to the source code requires considerable manual effort and is prone to errors. Given that firmware is typically closed-source, we collected 110 firmware samples from well-known open-source RTOS examples. We manually analyzed their implementation logic and the corresponding models based on our observations. We found that 19 (17%) of the samples do not perform the run-time transition cycle discussed in our paper, while 91 (83%) adhere closely to our heuristics: The firmware performs a waiting-processing run-time state transition cycle. For the 91 samples that follow our heuristics, we analyzed the functions used for the waiting and processing logic, the interrupt service routines, and the global objects involved in changing the run-time state. Regarding the methods used to transition to the waiting state, 7 (6%), 19 (17%), 8 (7%), and 82 (74%) of the samples use one of the four identified methods, respectively. Note that firmware can employ multiple methods to enter the waiting state. Our results indicate that continuous checking of global objects is a common method in the analyzed firmware samples. However, the use of different methods to enter the waiting state is mostly independent of each other and highly dependent on the RTOS design logic. For instance, the ChibiOS samples exclusively use an infinite loop, a method that is not commonly used. In contrast, the RIOT samples use both global object checks and WFI/WFE instructions to enter the waiting state. From this heuristic study, we conclude that our observation applies to the majority of firmware samples.

We proposed four conditions that firmware can use to enter the waiting state and

Table 4.4: Extra overhead caused by IRQ modeling

	Infinite loop searching (s)	IRQ Modeling times	Total time consuming (s)
Blehci	133	11	525
AnnePro2-Shine	32	7	286
TauLabs	51	15	1433
3Dprinter	136	7	397
bcn_rfd_ncp	76	7	524
coord_ncp	83	7	508
mac_no_beacon_sleep	66	6	390
nobcn_rfd	64	7	519
sam4l_qtouch	51	3	6
nmea_parser	232	4	489

applied empirical counter values to each threshold for the four conditions. Depending on the design logic and implementation of the firmware, the values we have chosen may not be optimal. For example, in the case of the 3D printer firmware, which uses two fixed-length buffers to send and receive characters over UART, both reading and sending characters take place within a single ISR. The ISR checks the value of the UART register to determine readiness for sending or receiving characters. When the sending buffer is full, the firmware continuously checks the buffer status until it is no longer full, then writes the character to the buffer. Compared to the sending request, the reading request is less frequent. If the threshold for checking global objects is set too low, the ISR will be triggered excessively, making it easier to consume characters from the sending buffer, but also increasing the data read for processing, thereby expanding the input space. Conversely, if the threshold is set too high, the ISR will be triggered infrequently, reducing the input space but failing to meet the character-sending requests. This can lead to the coverage feedback being interrupted if loops are executed without new basic blocks being detected.

4.5.4 Overhead (RQ3)

The additional overhead primarily stems from three sources. The first source is the memory read/write breakpoints. Despite our efforts to optimize the code and minimize the impact, it still incurs a 20%-25% overhead throughout the entire fuzzing process, as every memory access in the firmware undergoes scrutiny.

The second overhead arises from the search for infinite loops in the firmware image. Table 4.4 details the time consumed by AidFuzzer in locating infinite loops for all targets. The majority of these searches can be completed within 250 seconds. It is important to note that we conducted this search only once for each target, and the results can be reused in subsequent fuzzing.

The third contributor to overhead is the IRQ modeling. When the IRQ model needs to be updated and the model is not found in the model database, an IRQ modeling is conducted. Table 4.4 provides the overall times and time used for the IRQ modeling. In our experiments, IRQ modeling occurred mostly in the initial half-hour, and the modeling results were reused in subsequent fuzzing. Therefore, the overhead associated with IRQ modeling is deemed acceptable when compared to the overall fuzzing time.

Table 4.5: IRQ modeling comparison with AIM. *Ident* refers to the global objects identified by the tool. *TP* refers to the correctly identified global objects.

	AIM		AidFuzzer		Time(s)	
	Ident	TP	Ident	TP	AIM	AidFuzzer
cnc_r1	1	1	20	15	28	126
gateway_r2	13	13	12	12	1270	375
plc_r1	13	13	11	11	590	71
robot_r1_hardfpu	1	1	19	19	1784	117
reflow_oven_r1	13	13	1	1	699	135

4.5.5 IRQ Modeling Comparison with AIM (RQ4)

Since AidFuzzer and AIM use different underlying methods to analyze firmware—fuzzing and symbolic execution, respectively—a direct comparison of the number of discovered basic blocks is not meaningful. Instead, we conducted a quantitative comparison of AidFuzzer and AIM in terms of IRQ modeling in the firmware samples analyzed in the AIM paper experiment [73]. Both AidFuzzer and AIM share the insight that firmware changes global objects in the ISR to change execution behavior. Therefore, we counted the number of unique global objects identified by both methods and manually inspected their correctness as well as the analysis time spent on IRQ modeling. As shown in Table 4.5, AidFuzzer and AIM both identified the global objects correctly. However, AidFuzzer reported five false positives in cnc_r1. AidFuzzer aims to find the global objects that can be modified as many as possible. Therefore, AidFuzzer tries to symbolize the variable values thus it can explore all possible paths. As a result, due to the symbolic execution mechanism, some unreachable paths can be explored by our modeling engine, and the corresponding global objects that are modified within the paths are incorrectly identified. This comprehensive path exploration can lead to false positive global objects that cannot change the execution behavior of the firmware. The time required for IRQ modeling varied between a few seconds and minutes for the two methods, depending on the firmware logic. These variations in AidFuzzer are acceptable, as the modeling process only occurs once during fuzzing and can be reused in subsequent fuzzing runs. We observed that AidFuzzer in general spent less time on the IRQ modeling compared to AIM. More specifically, AidFuzzer only spent an average of 33% of the analysis time on the five samples used in the AIM experiments. For cnc_r1, AidFuzzer spent more time on analysis than AIM. The reason is that a register indicates the status of the device for a TIM IRQ ISR. While AidFuzzer explores all possible paths by symbolizing the register values, AIM only analyzes one path by giving the register a concrete value.

4.6 Discussion

Although the majority of the firmware follows the run-time state proposed in our thesis, according to the heuristic study result, there is still a portion of firmware that does not align with this assumption. Even though some firmware does not rely on the interrupt to accomplish its tasks, accurately modeling the firmware interrupt in a broad range requires additional work that depends on a more generic assumption.

AidFuzzer triggers interrupts only after their data are fully initialized in the ISR. Some may consider crashes caused by uninitialized data as bugs. While AidFuzzer may lead to false negatives in such scenarios, we assert that these bugs are less closely tied to security vulnerabilities. Our primary focus is on fuzzing the deep logic within the firmware, prompting us to strike a balance between deep logic exploration and the potential for false negatives. Although certain ineffective interrupts are not triggered by AidFuzzer, resulting in slightly lower coverage, this impact is deemed trivial when compared to overall coverage.

For real-time monitoring of the firmware state, AidFuzzer requires interception of every memory access, incurring extra overhead during the whole fuzzing process, which, while noticeable, is not negligible. We have optimized the interception function to minimize this overhead as much as possible.

Given the well-known bottleneck associated with symbolic execution, AidFuzzer faces challenges in effectively handling loops and intricate mathematical operations. To address issues related to control flow explosion, we chose to first explore the newly discovered basic blocks, albeit at the expense of potential false negatives. Moreover, when handling complicated nested structures, AidFuzzer cannot fully model the ISR, which may generate an incorrect result. When a data pointer is updated in the ISR that may alter the model result, we do not re-analyze it, resulting in an incorrect result as well. However, we need to make a trade-off between the fuzzing performance and the modeling soundness.

Compared with round-robin and fuzz-mode interrupt triggering mechanisms, AidFuzzer's adaptive method outperforms them in fuzzing, but may not correspond to real situations. For instance, the SysTick interrupt is only triggered at a fixed time interval in a real device, however, it is triggered multiple times within a short time window in fuzzing. We urge that as long as the discovered vulnerabilities can be reproduced in real devices, we can prioritize the fuzzing effectiveness.

4.7 Conclusion

In this chapter, we observed that the interrupt-triggering problem may hinder the fuzzing process when targeting RTOS firmware. We found that the key insight to solve the problem is to identify the run-time transition cycle of the firmware execution. The firmware run-time state can be categorized into two stages: the waiting state and the processing state. The interrupt should only be triggered when it is in a waiting state while the simulator should let it run in the processing state. We designed and implemented an adaptive interrupt-driven firmware fuzzing framework to tackle the interrupt-triggering problem. Our experiment results showed that when the interrupts are properly triggered, the discovered basic block and false positives caused by fuzzing can outperform the state-of-the-art works. We found eight previously unknown vulnerabilities and five of them were assigned CVEs by using our fuzzing framework in well-known RTOSes. In the next chapter, we focus on memory safety defense instead of the attack side. We chose to design a fully backward-compatible and secure TEE framework. Learning from the attack surfaces analysis, we found that the more interfaces the low-level software exposes, the more vulnerable it becomes. Therefore, when designing the TEE framework, we

consider both how the industry would accept the design (compatibility) and minimize the attack surfaces (security).

5

TEE Application: Backward Compatible and Secure TEE Design

5.1 Overview

Trusted Execution Environment (TEE) applications require special permission for specific resources, such as secure memory access, to run in the system. TEE applications must preserve and restore the context before entering and leaving the secure region (called *enclave*) to not leak secret data. As the software is tightly coupled with the platform, TEE applications are obviously low-level software. To manipulate the processor-provided isolation features, the TEE applications benefit from direct memory access languages such as C and C++. As we have seen before, the larger the code base is, the more gadgets the attacker could use to exploit memory corruption vulnerabilities.

However, the seamless integration of existing TEEs into the cloud is hindered, as they require substantial adaptation of the software executing inside an enclave as well as the cloud management software to handle enclaved workloads. TEEs enforce memory access control mechanisms to the protected ranges of memory using enclaves that are inaccessible even to the high-privilege software. The major platform vendors provide their proprietary enclave security architectures such as Intel SGX [103], Intel TDX [104], AMD SEV [134], ARM TrustZone [5] and ARM CCA [4]. Similarly, academic research has also proposed a variety of enclave architectures using CPU features or customized hardware, such as Keystone [121], Penglai [75], Sanctum [55], CURE [12] and Komodo [77]. However, these solutions have several shortcomings such as a lack of full backward compatibility, native live migration and secure I/O.

In Intel SGX [103], system calls are not allowed to be directly used inside the enclave. Therefore, programmers cannot develop SGX applications using normal toolchains. Although the provided SDK can facilitate the coding process, the cost of manual development for the specific CPU feature from scratch is still high. Recent works have attempted to port unmodified legacy applications into SGX such as Panoply [191], Haven [17], Scone [6] and Graphene-SGX [211]. However, these solutions suffer from scalability and compatibility problems. Keystone [121], Penglai [75], Komodo [77], and Sanctum [55] also provide their SDK for developers to develop enclaved applications and, as a result, slow down the acceptance of the TEE solutions by the industry.

To provide application-level level compatibility, virtual machine-based TEEs such as Intel TDX [104], AMD SEV [134] and ARM CCA [4] have been developed by the major industry players. However, the hypervisor is assumed to be untrusted in their threat model. Consequently, the hypervisor is deployed outside the enclave memory. The virtual machine OS kernel needs to be modified to adapt to the untrusted hypervisor. For instance, when it traps into the hypervisor, the virtual machine OS kernel is responsible for cleaning the secret data in the general purpose registers. Since the hypervisor and the virtual machine are not in the same enclave memory, the virtual machine is supposed to implement its secure I/O which requires extra developing effort for the programmer. ARM TrustZone [5] is another virtual machine-based TEE, however, it does not support multiple enclaves, so it cannot be deployed on cloud infrastructure. Hence, currently, virtual machine-based TEEs do not provide full backward compatibility.

To take full advantage of hardware resources, the cloud server commonly needs to migrate virtual machines to other platforms. However, existing virtual machine-based TEEs hardly provide simple and native migration features as the enclave memory is not

accessible to other software even for the hypervisor in the platform. Previous works such as [165] [90] [164] [91] aimed to provide third-party migration support to those TEEs. However, they either require additional hardware extension or an understanding of the enclave applications.

To tackle the problems of the existing solutions, we propose VirTEE, the first TEE architecture that allows strongly isolated execution of unmodified virtual machines (VMs) in enclaves, as well as secure live migration of VM enclaves between VirTEE-enabled servers. It is a full backward-compatible TEE on RISC-V architecture enabling native live migration and secure I/O by utilizing the RISC-V hypervisor extension and VirTEE hardware. Combined with its secure I/O capabilities, VirTEE enables the integration of enclaved computing in today's complex cloud infrastructure. VirTEE hardware allows both strong enclave memory isolation and large-size enclave while incurring small performance overhead. Facilitated by large-size enclave support, VirTEE runs unmodified kernel and applications on top of an enclave monitor in one enclave. The enclave monitor is located in the same enclave as the virtual machine and consequently can provide transparent native migration and secure I/O for the virtual machines. We thoroughly evaluate our RISC-V-based prototype and show its effectiveness and efficiency. The evaluation results show that VirTEE only imposes moderate overhead on standard benchmarks such as rv8, CoreMark, as well as on real-world software such as SQLite and OpenSSL.

5.2 RISC-V Hypervisor Extension

In this section, we give an overview of the aspects that are helpful to understand the remainder of the chapter. Specifically, we elaborate on the RISC-V privilege levels and the RISC-V hypervisor extension.

5.2.1 RISC-V Architecture Privilege Levels.

The RISC-V architecture defines four privilege levels. The firmware runs in machine mode, the most privileged mode (PL0), its memory integrity is protected by a secure boot and the Physical Memory Protection (PMP) unit. The operating system kernel runs in PL1, and user-space applications run in PL2. The PL3 privilege level is introduced by the RISC-V hypervisor extension which is elaborated next.

5.2.2 RISC-V Hypervisor Extension.

RISC-V introduced a hypervisor extension for virtualization. Instead of the operating system kernel, the hypervisor runs in PL1 and the virtual machine that contains the virtual machine kernel and virtual machine applications runs in PL2 and PL3. Any memory access from the virtual machine is further translated by a second-level page table to form the real physical address. The hypervisor can decide which physical memory page is mapped to the virtual machine by manipulating the second-level page table. Since RISC-V uses memory-mapped I/O (MMIO) to access device registers, the hypervisor is also able to intervene in the virtual machine I/O process. Similar to the system call, the firmware which is running in PL0, provides low-level functionality

interfaces called environment call (ECALL) for the operating system. Any ECALL from the virtual machine is handled by the hypervisor first. The hypervisor determines whether to forward the ECALL requests to the firmware or returns with fake values.

5.3 Secure TEE Design Analysis

When designing a secure TEE framework, two aspects need to be considered: compatibility and security.

5.3.1 Backward Compatibility

Compatibility determines how the industry would accept it. As we mentioned in this thesis, the application-based TEE design SGX [103] has been abandoned by Intel. Even with the vendor-provided fully featured SDK [100], developers still need to start from scratch to learn how to use it, making it hard for the developer to accept the application-based TEE design. To overcome the application-based TEE, the virtual machine-based TEE design is the mainstream industrial standard nowadays. By putting the whole TEE component into a virtual machine, the developers program in a familiar development environment, increasing the programmer's interest in accepting it. However, virtual machine-based TEE is not the silver bullet, as the size of the virtual machine is relatively bigger than an application, it inevitably exposes more attack surfaces. We mentioned in the thesis that due to the presence of code reuse attacks, the less code base it contains, the more secure environment it gains. This led to the second consideration of TEE design: how to minimize the code base and the exposed interfaces of the TEE framework.

5.3.2 Less Attack Surfaces Security

Virtual machine-based TEE designs such as TDX [104], and Arm CCA [4] all adopted the mechanism that the hypervisor should be put in the TEE. This design on the one hand treats the hypervisor as an untrusted component, making the OS in the virtual machine more resilient to attacks from the hypervisor, on the other hand, does not provide full backward compatibility for the users. An unmodified OS cannot run directly inside the virtual machine otherwise the secret data can be leaked by hypervisor introspection. A hypervisor has full control over the virtual machine it governs. A single-step breakpoint makes the register value leak through a VM exit. In addition, since the hypervisor is not a trusted component, the OS needs to implement the transparent secure IO for the device simulation, otherwise the IO data will be accessible to the hypervisor. To provide full backward compatibility and expose fewer attack surfaces, we need to design a TEE that makes the hypervisor a trusted component.

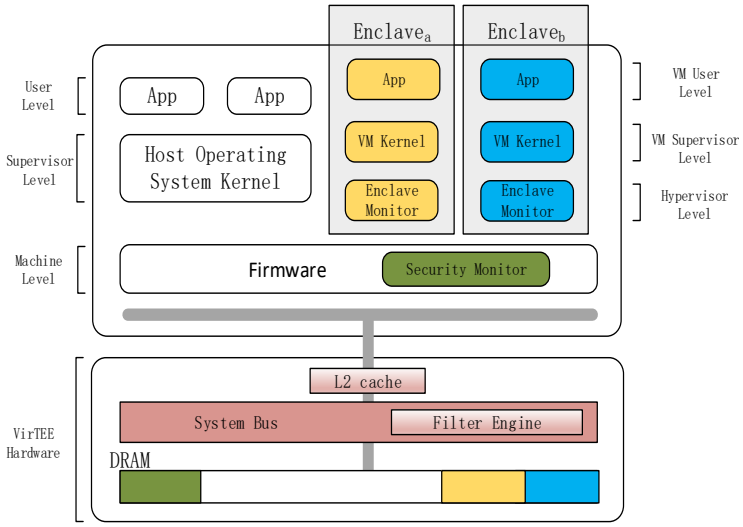

Figure 5.1: Design overview of VirTEE

5.4 Backward Compatible and Secure TEE Design

5.4.1 Adversary Model

We consider the adversary model along the line of related works [5, 12, 55, 27]. The Trusted Computing Base (TCB) consists of three components, i.e., 1) the underlying VirTEE hardware, 2)the security monitor, a privileged component in PL0 that can configure the VirTEE hardware, and 3) the enclave monitor running in the enclave. We assume that the adversary controls the whole OS. The adversary can leak secret data from the enclave using cache side-channel attacks, forge enclaves, etc. However, VirTEE, as for other TEE architectures, does not protect enclaves against memory corruption attacks. We also assume that the peripherals such as the hard drive are also accessible to the adversary. DoS attacks are orthogonal to the scope of this paper, as most TEEs do not give guarantees on availability.

5.4.2 Design Overview

VirTEE is a novel security architecture that allows the execution of unmodified virtual machines (VMs) in strongly isolated enclaves. Further, VirTEE completes this design with secure live migration and secure I/O. VirTEE design is shown in Figure 5.1. The VirTEE hardware provides strong physical enclave memory isolation and cache side-channel attack resilience. Based on the VirTEE hardware memory access control mechanism, the security monitor provides enclave memory management (e.g., creating new enclave memory, increasing and shirking enclave size) as well as enclave metadata management (e.g., measurement, header address, and size), attestation primitives and context switching for the host and the enclaves. Inside the enclave, the unmodified VMs run on top of the enclave monitor. Since they are in the same enclave, the enclave monitor can directly access the VM memory. The enclave monitor provides live

migration and secure I/O support for the VM that facilitates the enclave user a lot. In section 5.4.6, we will elaborate on the main workflow of VirTEE.

In the following, we will present the respective components in more detail.

5.4.3 VirTEE Hardware

The VirTEE hardware is the secure platform infrastructure of VirTEE. It divides the physical memory into enclaves (different colors in Figure 5.1). Several pairs of registers which are only accessible by the security monitor are used to record the enclave header address and size and are further used by the filter engine to grant access permissions. If and only if the instruction and target data are located in the same enclave, permission will be granted. The VirTEE hardware uses registers instead of the page table to manage enclave memory so that it can support large-size enclave memory. To provide cache side-channel attack resilience, when a CPU core is executing one enclave, it has its last-level cache partition which is not shared by other cores. The CPU core will clean its cache information before exiting the enclave. In this way, cache information is locked into the specific core, preventing cache side-channel attacks.

5.4.4 Security Monitor

The security monitor is a part of firmware and runs in PL0. It manages the enclave memory by manipulating the registers provided by the VirTEE hardware. Since it can change the whole enclave layout, it has full access to the physical memory. During platform boot, the integrity of the security monitor is verified by secure boot [105] so that we assume that the security monitor is not compromised at load time. In summary, the security monitor provides the following functionalities.

Enclave Metadata Management. During enclave initialization, the security monitor generates an enclave instance that contains the enclave id, header address, size, derived local attestation key, and its measurement report (i.e., enclave fingerprint), etc. The instance is stored in a list protected by the security monitor.

Enclave Memory Management. The security monitor can modify the registers to change the enclave memory layout. At enclave initialization, the security monitor allocates new enclave memory according to the request arguments. When it receives a request from the enclave to modify the enclave size, the security monitor first checks if the requested memory will overlap with other enclave memory. If so, the security monitor rejects the request. If not, the security monitor changes the corresponding registers and notifies the enclave.

Attestation Primitives. Attestation is used by the enclave application to prove that it is the genuine entity assumed by the verifier. Based on the prover's and the verifier's identity, the security monitor generates cryptographic reports and quota for local attestation and remote attestation respectively. A device key is hardcoded in the security monitor and is used to generate attestation keys. Since the security monitor has full access to the physical memory, it can implement a zero-copy (in place) report-generating mechanism which significantly reduces the overhead.

Context Switching. To enter an enclave, the security monitor first checks if the target enclave is ready to be executed. If the check passes, the security monitor prepares

the context for the enclave such as arguments, and sets up the control registers. Then, it jumps to the enclave. When the enclave terminates, the security monitor updates the enclave status and switches back to the host kernel.

5.4.5 Enclave Monitor

The enclave monitor running in the hypervisor privilege level is the core of VirTEE. We regard the enclave monitor as part of the TCB, therefore, the VM kernel is not required to clear sensitive data before trapping into the enclave monitor. The RISC-V hypervisor extension guarantees that every memory access including I/O registers access is automatically handled by the enclave monitor first. By intercepting the I/O process, the enclave monitor provides transparent secure I/O for the VM. Since the enclave monitor is located in the same enclave memory as the VM, it can directly read the VM memory and migrate the VM to another platform without third-party support. By using the enclave monitor, VirTEE achieves full backward compatibility, native live migration and secure I/O. However, note that the enclave monitor is small so that adds only a little attack surface to the enclave. We summarize the enclave monitor's main functionalities as follows.

Enclave Memory Management. In principle, without the enclave monitor, a kernel running in PL1 can access any physical address including the non-enclave memory even though the access will be blocked by the VirTEE hardware. However, we prevent such unintentional memory access by using the enclave monitor memory protection scheme. Every physical address that the VM accesses will be further translated by a second-level page table to the real physical address. We initialize a second-level page table at the enclave monitor initialization process. At run time, the enclave monitor communicates with the security monitor to allocate new enclave pages and maps the pages for the VM. The enclave monitor can prevent the VM from accessing non-enclave memory by mapping the enclave memory pages to an out-of-enclave VM address. RISC-V architecture uses memory-mapped I/O (MMIO) to access device registers. If the physical address is located in the MMIO memory range, it means that the VM kernel is accessing the device registers. We handle MMIO access by simulating the specific devices. The device virtualization details will be discussed next.

Device Virtualization and Secure I/O. For device-registers accesses, we parse the register values to get the I/O-request arguments such as the buffer address, size, and hard driver sector number. After extracting the arguments, the enclave monitor forwards the I/O requests to the real devices. During the forwarding process, the enclave monitor can encrypt and decrypt the I/O data transparently so that it achieves I/O data confidentiality. In VirTEE, we implemented a serial port as the console and a block device as the hard drive for their limited registers. Note that we can virtualize any device as long as the register accesses are properly handled.

In our threat model, the peripherals such as the hard drive can be readable for the attacker. For example, the VM kernel commonly uses a partition in the hard drive as a swap space and writes the memory into the partition. In this way, the secret memory will be leaked to the attacker. Without the enclave monitor, the VM kernel needs to be modified to encrypt the memory before writing them to the hard drive. With

the enclave monitor, the I/O requests are first handled by the enclave monitor. After receiving an I/O request, the enclave monitor looks up the device tree to infer which device registers the VM accesses. If it is the hard drive, the enclave monitor parses the arguments to get the sector number to see if it is in the swap partition, then the enclave monitor decides whether to encrypt or decrypt the data. In this way, VirTEE achieves transparent secure I/O.

Attestation Service. Attestation implementations are commonly deployed as separate enclaves by other TEEs. However, we encapsulate the attestation implementation in the enclave monitor, and the enclave monitor provides interfaces to the VM. For example, a VM, say the verifier, starts to attest another enclave VM, the prover, on the same platform. It then calls the local attestation interface exposed by the enclave monitor of the corresponding enclave. Then, the enclave monitor creates an attestation report of the enclave. The enclave monitor uses attestation primitives provided by the security monitor to complete the attestation process with the verifier. The Diffie–Hellman key exchange data are encapsulated in the report so that the secure channel is established and returned to the VM.

Live Migration Service. VirTEE provides a live migration service for the source enclave VM to migrate itself to a target trusted platform. VirTEE presents a stub enclave as the migration target. In the stub enclave, the enclave monitor keeps listening to remote migration requests and continues the VM execution after finishing the migration process. Similar to [45], VirTEE takes three steps to finish the migration process. First, the source enclave uses remote attestation to verify the target and establishes a secure channel with the target. Secondly, the source enclave monitor keeps the VM running, clears the dirty bit in the whole second-level page table, and transfers the VM memory to the target. Finally, the source enclave monitor stops the VM and checks the page-table dirty bit. Dirty pages are transferred again to the target platform as well as the virtual devices status and pending I/O requests. In this way, we keep the VM downtime small.

5.4.6 Enclave Setup

VirTEE works mainly as the following steps:

1. The host allocates contiguous physical memory and fills the memory with the enclave monitor and the virtual machine binary file and other necessary metadata in a predefined memory layout.

2. The host notifies the security monitor about the creation of one enclave with its physical address and initial size. The security monitor assigns a unique id to this enclave and binds the id to its header address and size.

3. The host finds an available CPU core and assigns the core to one enclave, then switches to the core, invokes the security monitor's entering enclave function with the specific enclave id as an argument.

4. The security monitor finds the corresponding enclave via its id and assigns the enclave memory by manipulating the VirTEE hardware registers. Afterward, the security monitor verifies the enclave signature by using a private key. If it fails,

the security monitor refuses to launch the enclave, otherwise, it performs context switching and hands over the control flow to the enclave monitor's entry point.

5. The enclave monitor receives three arguments from the security monitor: CPU core id, device tree which contains all device information, and the enclave memory header address.

6. The enclave monitor initializes the necessary virtual machine environment, e.g., second-level page table and virtual devices. Peripherals are allocated and bound to the virtual machine.

7. The enclave monitor hands over the control flow to the virtual machine's entry point,i.e., the virtual machine kernel entry point.

8. The virtual machine runs under the control of the enclave monitor. The virtual machine can use the migration or attestation services by calling the interfaces exposed by the enclave monitor.

5.5 Evaluation

We thoroughly evaluate our test framework and discuss the results. We aim to answer two research questions:

RQ1: How much extra performance overhead does it cause?

RQ2: How much extra performance overhead does the security feature cause?

This section presents our thorough evaluation of VirTEE's effectiveness and efficiency regarding 1) Benchmark run-time performance overhead, and 2) VirTEE features' overheads on microbenchmarks including enclave initialization, attestation, and live migration. For the run-time performance overhead, we selected standard benchmarks (rv8 [146] and CoreMark [70]), which have been used by CURE [12], the security architecture we build on, and real-world software (OpenSSL 3.0.0 [205], Binutils [86] and SQLite [97]). For VirTEE's features, we evaluated enclave initialization, attestation, and migration overhead by measuring the average running time.

5.5.1 Experiment Setup

We did our experiments on an Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz laptop with 16 GB RAM. The QEMU simulator version is 6.0.50.

Implementation We implemented the security monitor on top of OpenSBI. The hsm[1] handlers are extended to support security monitor environment call interfaces. Fourteen Maros are defined in the security monitor to perform enclave creation, enclave memory management, and attestation primitives. We emulated a UART8250 serial port as a console and a virtio block device as the hard drive in the enclave monitor. In summary, enclave monitor, the security monitor, and the host kernel required approximately 6700, 720, and 250 lines of C code respectively.

[1]Hart State Management SBI extension

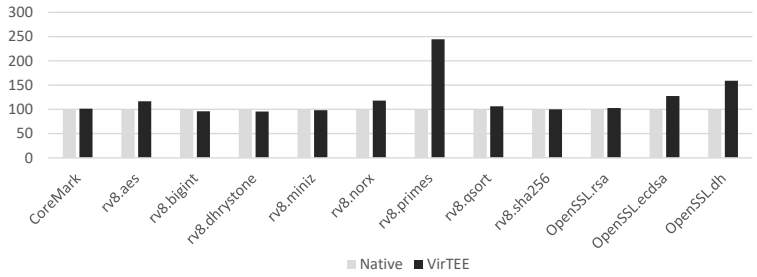

Figure 5.2: VirTEE's run-time performance overhead relative to native process

5.5.2 Run-time Performance Overhead (RQ1)

The run-time performance overhead mainly comes from RISC-V second-level page-table address translation, device virtualization (secure I/O), and VirTEE hardware. We selected standard benchmarks and real-world software to test VirTEE's performance overhead. The selected benchmarks include I/O-intensive workloads (Binutils, SQLite) and CPU-intensive workloads (rv8, CoreMark, OpenSSL). The hardware performance overhead has already been discussed in CURE [12], therefore we treat performance overhead induced by the CURE hardware as orthogonal. For each program, we run it five times with and without VirTEE in the same simulator and calculate the final average overhead.

I/O Intensive Workloads. We selected Binutils-*strings* and Binutils-*readelf*, two popular binary analysis tools, to extract information from an unstripped binary. The main performance overhead comes from the console virtualization. We used the SQLite *kvtest* program to generate a 1 GB test database file for SQLite. Then we ran *kvtest* to read blobs from the database file. The main performance overhead comes from the hard-drive virtualization and secure I/O. The experiment results show that VirTEE incurs 51% and 52% overhead in *strings* and *readelf* respectively. For SQLite, *kvtest* can read the database file in 54.7 MB/s and 87.7 MB/s with and without VirTEE respectively. That means VirTEE incurs 61% hard drive I/O overhead induced by hard-drive virtualization and secure I/O.

We conducted the same experiments in an AMD SEV-enabled platform with an AMD EPYC 7262 8-Core 3.2GHz Processor and 32GB RAM. The results show that AMD SEV incurs 532% and 214% overhead for *strings* and *readelf* respectively. For SQLite, AMD SEV incurs 25% hard-drive virtualization overhead. The reason for VirTEE's better performance for *strings* and *readelf* is that, for console virtualization, VirTEE does not need to intercept the I/O process and directly writes the data to the output buffer. However, for hard drive virtualization, VirTEE needs to intercept every I/O process, parse the I/O arguments, and performance encryption or decryption while AMD SEV does not support secure I/O. Therefore, VirTEE has a better virtual console performance and a worse virtual hard-drive performance than AMD SEV.

CPU Intensive Workloads. We selected three asymmetric crypto algorithm tests (dh, ecdsa and rsa) in OpenSSL's test vectors. CoreMark and rv8 use internal test data, therefore, we ran those binaries directly. As shown in Figure 5.2, for the majority of the

benchmarks, VirTEE only incurs less than 15% overhead. An exception is rv8-primes. The reason for that is that VirTEE does not support floating-point registers yet, and rv8-primes performs a lot of division operations. VirTEE has to use a soft-floating point to simulate the much slower calculation. This also happens in OpenSSL tests as well. We did the same experiment in AMD SEV, the results show that the SEV incurs an average of 15% overhead on the whole test programs as SEV supports real floating-point registers.

5.5.3 VirTEE Feature Overhead (RQ2)

Enclave Initialization. The enclave initialization process mainly includes continuous physical memory allocation, VM and enclave monitor binary-files loading, metadata filling, and integrity verification. In our experiment, the initial enclave size is 500MB, consisting of 100MB enclave monitor heap memory, 10MB enclave monitor and 490MB VM. The enclave monitor binary file size is 59KB and the VM kernel is 17MB. We filled the binary files to each part of the memory head and zeroed the remainder. We run an enclave VM five times and calculate the average time. The result shows that the enclave VM initialization approximately takes less than 150ms.

Location Attestation. We launched two enclave VMs running in parallel in a simulator. They run the same VM and enclave monitor binary. Starting from the verifier's first attestation call to the final secure channel setup, we counted the time elapsed and repeated the process five times. The result shows that the location attestation only takes 144 ms.

Remote Attestation. We simulated a remote attestation server by directly feeding the verification result back to the platform. We left the complete remote attestation system as future work. Starting from receiving the verifier's request to the final secure channel setup, we counted the time elapsed and repeated the process five times. The result shows that the remote attestation only takes less than 50ms.

Live Migration. We launched two QEMU simulators in parallel. The two simulators are bridged in one network card in the laptop. Now the two simulators are running in the same LAN. One is the target which contains a stub enclave and the other one is the migration source which contains a normal enclave VM. We set the bandwidth to 5MB/s. It takes 100 seconds to finish the second migration step. The third step depends on how many dirty pages are in the VM. In our experiment, we took two Binutils programs as examples, *readelf* and *objdump* contain a maximum of 1441 and 1652 dirty pages respectively which means they take approximately 1s and 1.2s to finish the final migration step.

5.6 Discussion

Although VirTEE achieves full backward compatibility, it may still suffer from the following problems during continuous development.

Large Code Base. We have implemented live migration, attestation, and transparent secure IO features in the hypervisor. They do not expose too many attack surfaces and can be used for the virtual machine without extra effort. However, while the

end-users require more features to be presented in the hypervisor, it inevitably becomes larger and larger. If the code base grows, it exposes more attack surfaces and leaves more code gadgets for attackers to perform code reuse attacks. One way to tackle this problem is to offload the tasks to the virtual machine. However, CCA and TDX have shown that this method considers the hypervisor as an untrusted component, making the design not fully backward-compatible. Another method is to de-bloat the hypervisor code base while maintaining the functionalities. However, this method requires a deep understanding of the design and implementation of the framework. Keeping a balance between the attack surfaces and security guarantees is still an open question when designing the secure TEE framework.

Transparent Secure IO. We implemented a secure IO feature in the hypervisor. For now, we adopted a lightweight TEA encryption mechanism. To provide extra data integrity protection, we need to implement a checksum mechanism, which may further slow down the IO performance. We see this feature as a future work.

5.7 Conclusion

In this chapter, we propose VirTEE. Based on RISC-V hypervisor extension and VirTEE hardware, we overcome the disadvantages that most state-of-the-art TEE solutions have. VirTEE can run unmodified kernels and applications in a virtual machine in enclave memory, providing full backward compatibility. With its novel design architecture, VirTEE supports native live migration and secure I/O. The evaluation indicates that VirTEE only incurs moderate performance overhead. This chapter proposes our memory safety defense consideration when designing a low-level software—TEE application framework. With the three works we mentioned in the thesis, we conduct a comprehensive and systematical memory safety analysis of low-level software. In the next chapter, we present the related works that are closely related to low-level software memory safety analysis as well as the security protection schemes that utilize the hardware features.

6
Related Work

6.1 Secure TEE Design

Existing non-virtual machine-based TEEs all require non-trivial porting effort to support legacy applications. Intel SGX [103] provides an SDK for programmers to develop SGX applications from scratch. Programmers are supposed to manually define the trusted part, the untrusted part, and their calling interfaces in a so-called EDL file. Although the SDK facilitates the development process, complex commercial software is still hard to adapt to SGX. LibOS based solutions such as Graphene [211], Occlum [189], SGX-LKL-OE [170], Fortanix [125], SCONE [6] try to port unmodified legacy applications to SGX. However, they only support limited system call interfaces and thus suffer from compatibility problems. Shim library-based solutions such as Haven [17] and Panoply [191] forward system call requests to the operating system kernel by shielding the enclave applications. Previous works show that they are prone to attacks through system call return values [33] [115]. Other TEEs such as Penglai [75] and Keystone [121] provide their development kit like SGX. The porting effort hinders the TEEs from being widely adopted. In contrast, VirTEE can run unmodified applications in a virtual machine and does not lead to the extra attack surface.

Virtual machine-based TEEs run unmodified applications in an enclave. Compared with VirTEE, Intel TDX [104], AMD SEV [134], and ARM CCA [4] isolate the virtual machine from untrusted parts, including the hypervisor. In this design, the hypervisor has no access to the virtual machine, which means the virtual machine kernel has to be modified to support native live migration and secure I/O. In addition, the virtual machine kernel is required to clear the sensitive data (e.g., general registers) before exiting the virtual environment. As this design does not support native migration and secure I/O, several works such as [165] [90] [164] [91] presented third-party solutions by using new instructions or security hardware modules. VirTEE deploys a enclave monitor inside the enclave. The enclave monitor is assumed to be a trusted component. It encapsulates the secure I/O and live migration functionalities for the virtual machine so that VirTEE can seamlessly support unmodified kernel. In this thesis, we implemented VirTEE hardware based on CURE [12], which provides a strong physical enclave memory isolation. However, since there is no enclave monitor in CURE's design, it does not support unmodified kernel, native live migration, and secure I/O.

6.2 Fuzzing

We split the wide range of fuzzing topics into two parts: general fuzzing and domain-specific fuzzing. General fuzzing does not focus on a specific target or environment. Instead, it tries to solve the problems that all fuzzing faces. Domain-specific fuzzing relies on general fuzzing methodology and focuses on specific targets such as OS kernel and firmware. Domain-specific fuzzing tackles more about other aspects other than the fuzzing itself. For example, when fuzzing an embedded firmware, efficiently re-hosting the firmware matters more than fuzzing.

6.2.1 General Fuzzing

General fuzzing targets various aspects of fuzzing, which include: value predicting [9], input format referring [7, 168, 23, 79, 176], input scheduling [80, 25], mutation policies [108], and manual annotation-based interesting input [8]. Various general fuzzing techniques have been integrated into fuzzing tools [80] and fuzzing frameworks [81]. In this thesis, we focus on domain-specific fuzzing: Bootloader and embedded firmware. Both of them benefit from prior general fuzzing ideas and implementations.

6.2.2 Domain Specific Fuzzing

Domain-specific fuzzing handles the obstacles that are tightly related to the specific target or environment. When applying the LibAFL [81] fuzzer model, domain-specific fuzzing spends more effort on the executor component. The executor, treated as a black box by LibAFL, however, is a non-negligible part of fuzzing.

Embedded Firmware Fuzzing Instead of re-hosting the firmware in an emulator environment, black-box fuzzing tools, such as Iotfuzzer [38], feed the fuzzing data from real devices (e.g., mobile phone Apps). Although black-box fuzzing mitigates the cumbersome effort to set up the emulator environment without having direct access to the firmware memory, it suffers from no coverage guidance. The same problem also happens to drone fuzzing [179] and [224]. Semi-simulation approaches [112] [112] [113] [120] [153] [201] [242] implement hardware in the loop method to forward the hardware access to the real physical devices, while the firmware itself runs in a simulated environment. This approach, however, requires lots of physical devices. Besides, as the generation of the data from physical devices is slow and cannot be easily controlled, this approach is not suitable for fuzzing. Due to the presence of physical devices, it is difficult to deploy a parallel analysis. Running the whole firmware in an emulator environment enables direct memory access to the firmware, turning it into a grey-box fuzzing. Qemu [20] has been widely used in full-system emulation. Qemu-based fuzzers [34] [56] [117] [246] heavily rely on the target-specific information that a large corpus of general firmware cannot deploy. HALucinator [46] and [126] propose to identify the hardware abstraction layer in the firmware for re-hosting. Unfortunately, the hardware abstraction layer has not been widely adopted by firmware development yet. Modeling the hardware abstract layer still requires much manual effort, such as reverse engineering. Without making too many knowledge assumptions about the firmware, full system emulation-based fuzzing is more scalable and requires less manual effort. PRETENDER [92] runs the original firmware independently in a simulated CPU. However, it still needs the hardware to model the peripheral behaviors. The peripheral behaviors are then solved by P2IM [74] by extracting the knowledge from the documentation or manuals and later further addressed by µEmu [248] and Fuzzerware [177] by symbolic execution modeling. Extracting the peripheral behavior model from the documentation is not stable and sometimes not reliable. Symbolic execution is known to be limited by its low scalability and confined to a small control flow scope. For example, it cannot model the MMIO data that is copied to global variables. Hoedur [178] splits the single-stream fuzzing data into multi-stream data that is identified by its MMIO address and its instruction address, making the fuzzing

data field aware. It avoids the avalanche caused by asynchronous interrupts. Recently, SafireFuzz [186] proposed to use dynamic binary re-writing to run the firmware on high-performance hardware. However, the hardware abstraction layer (HAL) needs to be present. Although HAL is becoming more and more popular in firmware development, there is still a large number of firmware that does not support it. Only two of the samples in our dataset support HAL. In addition, manually hooking HAL requires a lot of manual effort, which is also error-prone. All the full system emulation-based fuzzing either implements a simple round-robin or fuzz-mode interrupt triggering mechanism or relies on the unstable model extracted from the documentation and, therefore, cannot handle the complex interrupt situation. Concurrent to our work, AIM [73] proposed an idea similar to AidFuzzer for modeling firmware interrupts. However, our method has several advantages compared to AIM: First, AidFuzzer systematically investigates the relationship between interrupts and firmware run-time state. The global variables bridge them. In contrast, AIM does not recognize the firmware run-time state in a general and high-level view. Second, we propose four conditions that the firmware can use to require an interrupt. In contrast, AIM considers only one of them. Third, AIM does not consider the interrupt status and, thus, cannot prevent the firmware from crashing prematurely. Finally, AIM has to analyze the ISR by using dynamic symbolic execution when an event is enabled, which means that it has to perform symbolic execution frequently. Moreover, its firmware emulation is based on symbolic execution. The overall design has a strong impact on the execution speed.

UEFI and Bootloader Fuzzing Yang et al. [235] proposed the first fuzzing framework for UEFI firmware. They leverage the SIMICS virtual platform to emulate an environment for running UEFI firmware. This framework forces the CPU counter to point to the System Management Interrupt (SMI) handler function and directly places the fuzz input in the simulator memory to detect SMI out-of-bound memory access vulnerabilities. However, different SMI handlers may communicate with each other via variables, a complexity prior fuzzing tools could not handle, limiting their ability to test deeper logic. This issue was addressed by RSFuzzer [236]. RSFuzzer employs a two-stage fuzzing process. It begins by fuzzing a single SMI handler with randomly generated inputs. Before adding any new seed to the corpus, it extracts knowledge to infer the input structure. Cross-handler variables are identified by recording their handling behaviors. In the second stage, RSFuzzer performs cross-handler fuzzing using the knowledge extracted in the first stage. Bazhaniuk et al. [18] targeted SMM interrupt handler variables using symbolic execution but faced common challenges such as path explosion. Surve et al. [198] summarized the attack surfaces that UEFI firmware faces. The bootloader is launched after the firmware. As discussed in this paper, it faces a wide array of attack surfaces. Previous research targeting bootloaders has primarily focused on mobile devices, e.g., Android devices allow users to enter an interactive fastboot interface. An attacker with physical access to the device can boot it into fastboot mode by pressing a key combination upon boot or by connecting the device to a PC via ADB. The interactive command line interface accepts user input and processes requests accordingly. Roee [93] identified several command line parser vulnerabilities in commercial Android devices. BootStomp [174] performs taint analysis on bootloader binaries of mobile devices. Unlike Roee, BootStomp aims to find vulnerabilities caused

by the use of attacker-controlled storage data. In addition to identifying vulnerabilities, BootStomp designs and implements a framework for analyzing closed-source Android bootloaders. In comparison, our work presents a comprehensive attack surface analysis of PC bootloaders, along with a fuzzing tool. Unlike the studies by Roee and BootStomp, which focused solely on mobile devices and a single attack surface, our research covers a broader range of vulnerabilities in PC bootloaders.

OS Kernel Fuzzing Kernel fuzzing has always been a hot topic. They target system call generation [89, 29, 41, 197, 243] and device malicious inputs [195, 166, 190, 231]. The kernel is complex enough for fuzzers to reach the deep state of it. Researchers came up with ideas such as triggering an intended interrupt [123] to help the fuzzing process. Manual system call specification [89], on the one hand, is useful, on the other hand, it limits the fuzzing to a small range. This is why the automated system call arguments calibration [29] was proposed.

Hypervisor Fuzzing Hypervisor fuzzing works [94, 30, 28, 182, 163, 154, 84] were proposed to fuzz the simulated devices which also comprise the majority of the hypervisor code base. The malicious inputs come from port I/O, MMIO, and DMA buffer. Prior works utilize the specific processor or software features such as Intel-VTx hypervisor control structure and *pci_dma_read* function in Qemu. This corresponds to what we mentioned, that domain-specific fuzzing faces and deals with target-specific problems instead of general fuzzing problems.

6.3 Hardware Feature Assisted Memory Safety Protection

Hardware features do not always incur memory safety issues. They can also be applied to enhance the memory safety protection. kAFL[183] benefit from Intel PT and Intel VT-x to fuzz closed-source system-level software. KextFuzz [238] collects code coverage by replacing ARM Pointer authentication instructions with coverage collecting instructions. The debug port in ARM Cortex chip [224] functions similarly to Intel-PT, which has also been used to collect code coverage. ARM pointer authentication was used by [128] to defend against run-time attacks. CRYPTOMPK [111] presented an Intel Memory Protection Keys (MPK) based critical memory region protections.

7

Conclusion and Future Work

This thesis focuses on low-level software memory safety regarding both attack and defense sides. We focus on three typical low-level software: bootloader, embedded system firmware, and TEE applications, as our analysis targets. We proposed a comprehensive memory corruption vulnerability analysis of the bootloader, an adaptive interrupt-driven fuzzing for embedded firmware, and a full backward-compatible TEE with secure I/O. For the bootloader, our experiments uncovered 39 vulnerabilities, with 29 of these being confirmed or patched by the bootloader developers. Additionally, we have been assigned five CVEs so far. Overall, our findings highlight critical areas of concern in bootloader security and demonstrate the effectiveness of our fuzzing approach in identifying vulnerabilities. For the embedded firmware, in our collected firmware targets, we achieved higher and faster coverage when dealing with complex interrupt firmware compared with the state-of-the-art approaches. Besides, we found eight previously unknown security issues in real-world firmware. For the secure TEE design, we can run unmodified kernels and applications in enclave memory on a virtual machine, providing full backward compatibility. With its novel design architecture, we support native live migration and secure I/O. The evaluation indicates that our design and implementation only incur moderate performance overhead.

Memory safety issues have been and will be one of the paramount threats to system security due to their severe impacts. In this thesis, we choose three typical low-level software as our analysis targets and reveal that memory corruption vulnerabilities still exist in widely deployed and well-tested applications. Memory safety is supposed to gain more attention from low-level software developers.

For the bootloader, our fuzzing framework can be deployed to fuzz different bootloader designs despite their implementation. However, our fuzzing strategy is standardized and not tailored for bootloader fuzzing. For example, automatically generating harnesses for different bootloaders by using a popular large language model could be a promising topic. Nevertheless, it requires researchers to manually correct and test the harness, which requires extra manual effort. Another shortcoming of our work is that the framework does not handle closed-source bootloaders and, therefore, cannot fuzz mobile device bootloaders. Without the source code, the execution environment is opaque, and we do not know which address to load the bootloader image and how to boot it. These problems are potential research topics for future work. For firmware fuzzing, besides the interrupt triggering problem, there are still many obstacles waiting to be solved. For example, the high-level protocol implementation, such as the TCP/IP Bluetooth protocol stack, has not been well fuzzed due to the fuzzing bottleneck caused by inaccurate peripheral simulation and DMA data transferring. These are promising topics that deserve more research efforts. For TEE applications, SGX has been abandoned by Intel, which proves that incompatible TEE applications cannot be accepted by the developers. Therefore, virtual machine-based TEE will be the mainstream scheme. On the one hand, unmodified applications can be directly executed in the TEE, however, on the other hand, the large code base exposes more attack surfaces, especially memory corruption vulnerabilities. Finding a balance between offloading the tasks from the trusted to the untrusted execution environment and backward compatibility is still an open question.

Memory-unsafe programming languages such as C and C++are still the backbone of modern low-level software. Linus once held a nondeterministic attitude towards embracing Rust in the Linux kernel. However, he did not see a memory-safe programming language like Rust as a silver bullet to solve the problem. Though the data lifetime and ownership design saves Rust from the majority of memory safety issues, the *unsafe* feature provided by Rust for the developers to connect their Rust applications with native applications leaves a potential flaw. Nevertheless, we believe that it will take a long time for the community to accept Rust as their first programming language, not only because of the steep learning curve but also because of the immature library management. Therefore, memory safety issues will not disappear in the low-level software soon.

We mainly focus on the problems on x86 and ARM platforms. Nowadays, RISC-V has gained much attention. It may introduce more new features which require more low-level software to drive. These applications can lead to more memory corruption issues. Similar methodologies can be applied to the new processors and platforms.

Bibliography

Other references

[1] *8250 UART Programming Serial Programming*. https://en.wikibooks.org/wiki/SerialProgramming/8250_UARTProgramming.

[2] Android. *Android Debug Bridge (adb)*. https://developer.android.com/tools/adb?.

[3] Angelakopoulos, I., Stringhini, G., and Egele, M. {Firmsolo}: enabling dynamic analysis of binary linux-based {iot} kernel modules. In: *USENIX Security Symposium*. 2023.

[4] ARM. *Arm Confidential Compute Architecture*. https://documentation-service.arm.com/static/61825631f45f0b1fbf3a7a7d?token=.

[5] ARM. *Security technology: building a secure system using TrustZone technology*. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/TrustZone-and-FIDO-white-paper.pdf?revision=98e6ae26-92ca-4ffd-ac4e-3329b7f8a23e.

[6] Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'keeffe, D., Stillwell, M. L., et al. {Scone}: secure linux containers with intel {sgx}. In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2016.

[7] Aschermann, C., Frassetto, T., Holz, T., Jauernig, P., Sadeghi, A.-R., and Teuchert, D. Nautilus: fishing for deep bugs with grammars. In: 2019.

[8] Aschermann, C., Schumilo, S., Abbasi, A., and Holz, T. Ijon: Exploring deep state spaces via fuzzing. In: *IEEE Symposium on Security and Privacy (S&P)*. 2020.

[9] Aschermann, C., Schumilo, S., Blazytko, T., Gawlik, R., and Holz, T. Redqueen: Fuzzing with Input-to-State Correspondence. In: *Symposium on Network and Distributed System Security (NDSS)*. 2019.

[10] Atlidakis, V., Godefroid, P., and Polishchuk, M. Restler: stateful rest api fuzzing. In: *International Conference on Software Engineering (ICSE)*. 2019.

[11] Avgustinov, P., De Moor, O., Jones, M. P., and Schäfer, M. QL: Object-oriented queries on relational data. In: *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. 2016.

[12] Bahmani, R., Brasser, F., Dessouky, G., Jauernig, P., Klimmek, M., Sadeghi, A.-R., and Stapf, E. {Cure}: a security architecture with customizable and resilient enclaves. In: *USENIX Security Symposium*. 2021.

[13] Bai, J.-J., Lawall, J., Chen, Q.-L., and Hu, S.-M. Effective static analysis of concurrency {use-after-free} bugs in linux device drivers. In: *USENIX Annual Technical Conference (ATC)*. 2019.

[14] Baidu. *SGXRay: Automated Vulnerability Finding in SGX Enclave Application*. `https://github.com/baidu/sgxray`.

[15] barebox developers. *Barebox*. `https://www.barebox.org/`.

[16] Bauman, E., Lin, Z., Hamlen, K. W., et al. Superset disassembly: statically rewriting x86 binaries without heuristics. In: 2018.

[17] Baumann, A., Peinado, M., and Hunt, G. Shielding applications from an untrusted cloud with haven. In: 2015.

[18] Bazhaniuk, O., Loucaides, J., Rosenbaum, L., Tuttle, M. R., and Zimmer, V. Symbolic Execution for BIOS Security. In: *USENIX Workshop on Offensive Technologies (WOOT)*. 2015.

[19] Becker, L., Hollick, M., and Classen, J. {Sok}: on the effectiveness of {control-flow} integrity in practice. In: *USENIX Workshop on Offensive Technologies (WOOT)*. 2024.

[20] Bellard, F. QEMU, a fast and portable dynamic translator. In: *USENIX Annual Technical Conference (ATC)*. 2005.

[21] Biondo, A., Conti, M., Davi, L., Frassetto, T., and Sadeghi, A.-R. The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX. In: *USENIX Security Symposium*. 2018.

[22] blackberry. *QNX Hypervisor*. `https://blackberry.qnx.com/en/products/foundation-software/qnx-hypervisor`.

[23] Blazytko, T., Bishop, M., Aschermann, C., Cappos, J., Schlögel, M., Korshun, N., Abbasi, A., Schweighauser, M., Schinzel, S., Schumilo, S., et al. {Grimoire}: synthesizing structure while fuzzing. In: *USENIX Security Symposium*. 2019.

[24] Bletsch, T., Jiang, X., Freeh, V. W., and Liang, Z. Jump-oriented programming: a new class of code-reuse attack. In: *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. 2011.

[25] Böhme, M., Pham, V.-T., and Roychoudhury, A. Coverage-based greybox fuzzing as markov chain. In: *ACM Conference on Computer and Communications Security (CCS)*. 2016.

[26] Bond, M. D., Coons, K. E., and McKinley, K. S. Pacer: proportional detection of data races. In: *ACM Sigplan Notices*. 2010.

[27] Brasser, F., Gens, D., Jauernig, P., Sadeghi, A.-R., and Stapf, E. Sanctuary: arming trustzone with user-space enclaves. In: *Symposium on Network and Distributed System Security (NDSS)*. 2019.

[28] Bulekov, A., Das, B., Hajnoczi, S., and Egele, M. Morphuzz: Bending (input) space to fuzz virtual devices. In: *USENIX Security Symposium*. 2022.

[29] Bulekov, A., Das, B., Hajnoczi, S., and Egele, M. No grammar, no problem: towards fuzzing the linux kernel without system-call descriptions. In: 2023.

[30] Bulekov, A., Liu, Q., Egele, M., and Payer, M. {Hyperpill}: fuzzing for hypervisor-bugs by leveraging the hardware virtualization interface. In: *USENIX Security Symposium*. 2024.

[31] bzt. *easyboot*. `https://gitlab.com/bztsrc/easyboot/`.

[32] Carlini, N., Barresi, A., Payer, M., Wagner, D., and Gross, T. R. {Control-flow} bending: on the effectiveness of {control-flow} integrity. In: *USENIX Security Symposium*. 2015.

[33] Checkoway, S. and Shacham, H. Iago attacks: why the system call api is a bad untrusted rpc interface. In: *ACM SIGARCH Computer Architecture News*. 2013.

[34] Chen, D. D., Woo, M., Brumley, D., and Egele, M. Towards automated dynamic analysis for linux-based embedded firmware. In: *Symposium on Network and Distributed System Security (NDSS)*. 2016.

[35] Chen, G. and Zhang, Y. {Mage}: mutual attestation for a group of enclaves without trusted third parties. In: *USENIX Security Symposium*. 2022.

[36] Chen, G. and Zhang, Y. Securing tees with verifiable execution contracts. In: *IEEE Transactions on Dependable and Secure Computing*. 2022.

[37] Chen, G., Zhang, Y., and Lai, T.-H. Opera: open remote attestation for intel's secure enclaves. In: *ACM Conference on Computer and Communications Security (CCS)*. 2019.

[38] Chen, J., Diao, W., Zhao, Q., Zuo, C., Lin, Z., Wang, X., Lau, W. C., Sun, M., Yang, R., and Zhang, K. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In: *Symposium on Network and Distributed System Security (NDSS)*. 2018.

[39] Chen, L., Cai, Q., Ma, Z., Wang, Y., Hu, H., Shen, M., Liu, Y., Guo, S., Duan, H., Jiang, K., et al. Sfuzz: slice-based fuzzing for real-time operating systems. In: *ACM Conference on Computer and Communications Security (CCS)*. 2022.

[40] Chen, L., Li, Z., Ma, Z., Li, Y., Chen, B., and Zhang, C. Enclavefuzz: finding vulnerabilities in sgx applications. In: 2024.

[41] Chen, W., Wang, Y., Zhang, Z., and Qian, Z. Syzgen: automated generation of syscall specification of closed-source macos drivers. In: *ACM Conference on Computer and Communications Security (CCS)*. 2021.

[42] Cheng, K., Li, Q., Wang, L., Chen, Q., Zheng, Y., Sun, L., and Liang, Z. Dtaint: detecting the taint-style vulnerability in embedded device firmware. In: *Conference on Dependable Systems and Networks (DSN)*. 2018.

[43] Chesser, M., Nepal, S., and Ranasinghe, D. C. {Multifuzz}: a {multi-stream} fuzzer for testing monolithic firmware. In: *USENIX Security Symposium*. 2024.

[44] Choi, J., Kim, K., Lee, D., and Cha, S. K. NTFuzz: Enabling type-aware kernel fuzzing on windows with static binary analysis. In: *IEEE Symposium on Security and Privacy (S&P)*. 2021.

[45] Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A. Live migration of virtual machines. In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2005.

[46] Clements, A. A., Gustafson, E., Scharnowski, T., Grosen, P., Fritz, D., Kruegel, C., Vigna, G., Bagchi, S., and Payer, M. HALucinator: Firmware re-hosting through abstraction layer emulation. In: *USENIX Security Symposium*. 2020.

[47] Cloosters, T., Rodler, M., and Davi, L. TeeRex: Discovery and Exploitation of Memory Corruption Vulnerabilities in SGX Enclaves. In: *USENIX Security Symposium*. 2020.

[48] Cloosters, T., Willbold, J., Holz, T., and Davi, L. SGXFuzz: Efficiently synthesizing nested structures for SGX enclave fuzzing. In: *USENIX Security Symposium*. 2022.

[49] CloverBootloader developers. *CloverBootloader*. `https://github.com/CloverHackyColor/CloverBootloader/`.

[50] Codetector1374. *Anne Pro 2 Shine!* `https://github.com/OpenAnnePro/AnnePro2-Shine/tree/master`.

[51] Company, B. *efiXplorer*. `https://i.blackhat.com/eu-20/Wednesday/eu-20-Labunets-efiXplorer-Hunting-For-UEFI-Firmware-Vulnerabilities-At-Scale-With-Automated-Static-Analysis.pdf`.

[52] Company, B. *Static analysis-based recovery of service function calls and type information in UEFI firmware*. `https://raw.githubusercontent.com/binarlyio/Research_Publications/main/EKO_2020/EKO_2020_efiXplorer.pdf`.

[53] Company, T. *TIOBE Software Ranking*. `https://www.tiobe.com/tiobe-index/`.

[54] Costan, V. Intel sgx explained. In: *IACR Cryptol, EPrint Arch*. 2016.

[55] Costan, V., Lebedev, I., and Devadas, S. Sanctum: minimal hardware extensions for strong software isolation. In: *USENIX Security Symposium*. 2016.

[56] Costin, A., Zarras, A., and Francillon, A. Automated dynamic firmware analysis at scale: a case study on embedded web interfaces. In: *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. 2016.

[57] Coverity. *Coverity Scan: barebox*. `https://scan.coverity.com/projects/barebox`.

[58] Cui, J., Yu, J. Z., Shinde, S., Saxena, P., and Cai, Z. SmashEx: Smashing SGX Enclaves Using Exceptions. In: *ACM Conference on Computer and Communications Security (CCS)*. 2021.

[59] Cui, R., Zhao, L., and Lie, D. Emilia: Catching Iago in Legacy Code. In: *Symposium on Network and Distributed System Security (NDSS)*. 2021.

[60] Daniel Axtens. *Fuzzing grub*. https://sthbrx.github.io/blog/2021/03/04/fuzzing-grub-part-1/.

[61] Daniel Axtens. *Fuzzing grub, part 2: going faster*. https://sthbrx.github.io/blog/2021/06/14/fuzzing\protect\discretionary{\char\hyphenchar\font}{}{}grub\protect\discretionary{\char\hyphenchar\font}{}{}part\protect\discretionary{\char\hyphenchar\font}{}{}2\protect\discretionary{\char\hyphenchar\font}{}{}going-faster/.

[62] Darkreading. *Critical 'LogoFAIL' Bugs Offer Secure Boot Bypass for Millions of PCs*. https://www.darkreading.com/endpoint-security/critical-logofail-bugs-secure-boot-bypass-millions-pcs.

[63] Das U-Boot developer. *The U-Boot Documentation*. https://docs.u-boot.org/en/latest/.

[64] Das U-Boot Developers. *Das U-Boot Sanitizer Compilation Kconfig*. https://github.com/u\protect\discretionary{\char\hyphenchar\font}{}{}boot/u-boot/blob/master/Kconfig#L157.

[65] Deligiannis, P., Donaldson, A. F., and Rakamaric, Z. Fast and precise symbolic analysis of concurrency bugs in device drivers (t). In: *ACM/IEEE International Conference on Automated Software Engineering (ASE)*. 2015.

[66] Dinesh, S., Burow, N., Xu, D., and Payer, M. Retrowrite: statically instrumenting cots binaries for fuzzing and sanitization. In: *IEEE Symposium on Security and Privacy (S&P)*. 2020.

[67] Ding, Y., Duan, R., Li, L., Cheng, Y., Zhang, Y., Chen, T., Wei, T., and Wang, H. Poster: Rust SGX SDK: Towards memory safety in Intel SGX enclave. In: *ACM Conference on Computer and Communications Security (CCS)*. 2017.

[68] DynamoRIO Team. *Dynamic Instrumentation Tool Platform*. https://dynamorio.org/.

[69] Eclypsium. *There's A Hole In The Boot*. https://eclypsium.com/blog/theres-a-hole-in-the-boot/.

[70] EEMBC. *EMBC. Coremark*. https://www.eembc.org/coremark/.

[71] Engler, D. and Ashcraft, K. Racerx: effective, static detection of race conditions and deadlocks. In: *ACM SIGOPS operating systems review*. 2003.

[72] Enrique Nissim, K. O. *AMD Sinkclose Universal SMM Privilege Escalation*. https://media.defcon.org/DEF%20CON%2032/DEF%20CON%2032%20presentations/DEF%20CON%2032%20-%20Enrique%20Nissim%20Krzysztof%20Okupski%20-%20AMD%20Sinkclose%20Universal%20Ring-2%20Privilege%20Escalation%20Redacted.pdf.

[73] Feng, B., Luo, M., Liu, C., Lu, L., and Kirda, E. AIM: Automatic Interrupt Modeling for Dynamic Firmware Analysis. In: *Conference on Dependable Systems and Networks (DSN)*. 2023.

[74] Feng, B., Mera, A., and Lu, L. P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In: *USENIX Security Symposium*. 2020.

[75] Feng, E., Lu, X., Du, D., Yang, B., Jiang, X., Xia, Y., Zang, B., and Chen, H. Scalable Memory Protection in the PENGLAI Enclave. In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2021.

[76] Feng, Q., Zhou, R., Xu, C., Cheng, Y., Testa, B., and Yin, H. Scalable graph-based bug search for firmware images. In: *ACM Conference on Computer and Communications Security (CCS)*. 2016.

[77] Ferraiuolo, A., Baumann, A., Hawblitzel, C., and Parno, B. Komodo: using verification to disentangle secure-enclave hardware from software. In: *Symposium on Operating Systems Principles (SOSP)*. 2017.

[78] Fioraldi, A. Program state abstraction for feedback-driven fuzz testing using likely invariants. In: *arXiv preprint arXiv:2012.11182*. 2020.

[79] Fioraldi, A., D'Elia, D. C., and Coppa, E. Weizz: automatic grey-box fuzzing for structured binary formats. In: *International Symposium on Software Testing and Analysis (ISSTA)*. 2020.

[80] Fioraldi, A., Maier, D., Eißfeldt, H., and Heuse, M. AFL++: Combining incremental steps of fuzzing research. In: *USENIX Workshop on Offensive Technologies (WOOT)*. 2020.

[81] Fioraldi, A., Maier, D., Zhang, D., and Balzarotti, D. LibAFL: A Framework to Build Modular and Reusable Fuzzers. In: *ACM Conference on Computer and Communications Security (CCS)*. 2022.

[82] Fleischer, M., Das, D., Bose, P., Bai, W., Lu, K., Payer, M., Kruegel, C., and Vigna, G. {Actor}:{action-guided} kernel fuzzing. In: *USENIX Security Symposium*. 2023.

[83] Foundation, T. A. S. *Apache Blehci example*. `https://mynewt.apache.org/v1_5_0/tutorials/ble/blehci_project.html`.

[84] Ge, X., Niu, B., Brotzman, R., Chen, Y., Han, H., Godefroid, P., and Cui, W. HYPERFUZZER: An efficient hybrid fuzzer for virtual cpus. In: *ACM Conference on Computer and Communications Security (CCS)*. 2021.

[85] Gens, D., Schmitt, S., Davi, L., and Sadeghi, A.-R. K-miner: Uncovering Memory Corruption in Linux. In: 2018.

[86] GNU. *GNU Binutils*. `https://www.gnu.org/software/binutils/`.

[87] GNU community. *GNU GRUB*. `https://www.gnu.org/software/grub/index.html`.

[88] GNU community. *The GNU C Library*. `https://www.gnu.org/software/libc/`.

[89] Google. *syzkaller*. `Linuxsyscallfuzzer.https://github.com/google/syzkaller`.

[90] Gu, J., Hua, Z., Xia, Y., Chen, H., Zang, B., Guan, H., and Li, J. Secure live migration of sgx enclaves on untrusted cloud. In: *Conference on Dependable Systems and Networks (DSN)*. 2017.

[91] Guerreiro, J., Moura, R., and Silva, J. N. Teender: sgx enclave migration using hsms. In: *Computers & Security*. 2020.

[92] Gustafson, E., Muench, M., Spensky, C., Redini, N., Machiry, A., Fratantonio, Y., Balzarotti, D., Francillon, A., Choe, Y. R., Kruegel, C., et al. Toward the analysis of embedded firmware through automated re-hosting. In: *Symposium on Recent Advances in Intrusion Detection (RAID)*. 2019.

[93] Hay, R. fastboot oem vuln: Android bootloader vulnerabilities in vendor customizations. In: *USENIX Workshop on Offensive Technologies (WOOT)*. 2017.

[94] Henderson, A., Yin, H., Jin, G., Han, H., and Deng, H. Vdf: targeted evolutionary fuzz testing of virtual devices. In: *Symposium on Recent Advances in Intrusion Detection (RAID)*. 2017.

[95] Hernandez, G., Fowze, F., Tian, D., Yavuz, T., and Butler, K. R. Firmusb: vetting usb device firmware using domain informed symbolic execution. In: *ACM Conference on Computer and Communications Security (CCS)*. 2017.

[96] Hernandez, G., Muench, M., Maier, D., Milburn, A., Park, S., Scharnowski, T., Tucker, T., Traynor, P., and Butler, K. Firmwire: transparent dynamic analysis for cellular baseband firmware. In: 2022.

[97] Hipp, R. D. *SQLite*. `https://www.sqlite.org/index.html`.

[98] Hu, H., Shinde, S., Adrian, S., Chua, Z. L., Saxena, P., and Liang, Z. Data-oriented programming: on the expressiveness of non-control data attacks. In: *IEEE Symposium on Security and Privacy (S&P)*. 2016.

[99] Intel. *Finding BIOS Vulnerabilities with Symbolic Execution and Virtual Platforms*. `https://www.intel.com/content/www/us/en/developer/inproceedingss/technical/finding-bios-vulnerabilities-with-symbolic-execution-and-virtual-platforms.html`.

[100] Intel. *Intel-SGX SDK*. `https://github.com/intel/linux-sgx`.

[101] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1-Real mode*. `https://cdrdv2.intel.com/v1/dl/getContent/671436`.

[102] Intel. *Intel® Processor Trace*. `https://edc.intel.com/content/www/de/de/design/ipla/software-development-platforms/client/platforms/alder-lake-desktop/12th-generation-intel-core-processors-datasheet-volume-1-of-2/004/intel-processor-trace/`.

[103] Intel. *Intel® Software Guard Extensions*. `https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf`.

[104] Intel. *Intel® Trust Domain Extensions*. `https://www.intel.com/content/dam/develop/external/us/en/documents/intel-tdx-module-1.5-base-spec-348549001.pdf`.

[105] Intel. *UEFI Secure Boot*. https://www.intel.com/content/dam/doc/product-specification/efi-v1-10-specification.pdf.

[106] Intel. *What Is Virtualization Security?* https://www.intel.com/content/www/us/en/business/enterprise-computers/resources/virtualization-security.html.

[107] Jangid, M. K., Chen, G., Zhang, Y., and Lin, Z. Towards formal verification of state continuity for enclave programs. In: *USENIX Security Symposium*. 2021.

[108] Jauernig, P., Jakobovic, D., Picek, S., Stapf, E., and Sadeghi, A.-R. Darwin: survival of the fittest fuzzing mutators. In: *arXiv preprint arXiv:2210.11783*. 2022.

[109] Jeong, D. R., Lee, B., Shin, I., and Kwon, Y. Segfuzz: segmentizing thread interleaving to discover kernel concurrency bugs through fuzzing. In: *IEEE Symposium on Security and Privacy (S&P)*. 2023.

[110] Jiang, Y., Yang, Y., Xiao, T., Sheng, T., and Chen, W. Drddr: a lightweight method to detect data races in linux kernel. In: *The Journal of Supercomputing*. 2016.

[111] Jin, X., Xiao, X., Jia, S., Gao, W., Gu, D., Zhang, H., Ma, S., Qian, Z., and Li, J. Annotating, tracking, and protecting cryptographic secrets with cryptompk. In: *IEEE Symposium on Security and Privacy (S&P)*. 2022.

[112] Kammerstetter, M., Burian, D., and Kastner, W. Embedded security testing with peripheral device caching and runtime program state approximation. In: *10th International Conference on Emerging Security Information, Systems and Technologies (SECUWARE)*. 2016.

[113] Kammerstetter, M., Platzer, C., and Kastner, W. Prospect: peripheral proxying supported embedded code testing. In: *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. 2014.

[114] Kasten, F., Zieris, P., and Horsch, J. Integrating static analyses for high-precision control-flow integrity. In: *Symposium on Recent Advances in Intrusion Detection (RAID)*. 2024.

[115] Khandaker, M. R., Cheng, Y., Wang, Z., and Wei, T. Coin attacks: on insecurity of enclave untrusted interfaces in sgx. In: *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2020.

[116] Kim, K., Jeong, D. R., Kim, C. H., Jang, Y., Shin, I., and Lee, B. Hfl: hybrid fuzzing on the linux kernel. In: 2020.

[117] Kim, M., Kim, D., Kim, E., Kim, S., Jang, Y., and Kim, Y. Firmae: Towards large-scale emulation of iot firmware for dynamic analysis. In: *Annual Computer Security Applications Conference (ACSAC)*. 2020.

[118] Kim, T., Kumar, V., Rhee, J., Chen, J., Kim, K., Kim, C. H., Xu, D., and Tian, D. J. {Pasan}: detecting peripheral access concurrency bugs within {bare-metal} embedded applications. In: *USENIX Security Symposium*. 2021.

[119] Klees, G., Ruef, A., Cooper, B., Wei, S., and Hicks, M. Evaluating fuzz testing. In: *ACM Conference on Computer and Communications Security (CCS)*. 2018.

[120] Koscher, K., Kohno, T., and Molnar, D. SURROGATES: Enabling Near-Real-Time dynamic analyses of embedded systems. In: *USENIX Workshop on Offensive Technologies (WOOT)*. 2015.

[121] Lee, D., Kohlbrenner, D., Shinde, S., Asanović, K., and Song, D. Keystone: An open framework for architecting trusted execution environments. In: *European Conference on Computer Systems (EuroSys)*. 2020.

[122] Lee, J., Jang, J., Jang, Y., Kwak, N., Choi, Y., Choi, C., Kim, T., Peinado, M., and Kang, B. B. Hacking in darkness: Return-oriented programming against secure enclaves. In: *USENIX Security Symposium*. 2017.

[123] Lee, Y., Min, C., and Lee, B. {Exprace}: exploiting kernel races through raising interrupts. In: *USENIX Security Symposium*. 2021.

[124] Lei, C., Ling, Z., Zhang, Y., Yang, Y., Luo, J., and Fu, X. A friend's eye is a good mirror: synthesizing {mcu} peripheral models from peripheral drivers. In: *USENIX Security Symposium*. 2024.

[125] Leiserson, A. *Side Channels and Runtime Encryption Solutions with Intel® SGX*. `https://www.fortanix.com/assets/Fortanix_Side_Channel_Whitepaper.pdf`.

[126] Li, W., Guan, L., Lin, J., Shi, J., and Li, F. From library portability to para-rehosting: Natively executing microcontroller software on commodity hardware. In: *Symposium on Network and Distributed System Security (NDSS)*. 2021.

[127] *libFuzzer – a library for coverage-guided fuzz testing*. `https://llvm.org/docs/LibFuzzer.html`.

[128] Liljestrand, H., Nyman, T., Wang, K., Perez, C. C., Ekberg, J.-E., and Asokan, N. {Pac} it up: towards pointer integrity using {arm} pointer authentication. In: *USENIX Security Symposium*. 2019.

[129] Limine developers. *Limine*. `https://limine-bootloader.org/`.

[130] Liu, C., Mera, A., Kirda, E., Xu, M., and Lu, L. {Co3}: concolic co-execution for firmware. In: *USENIX Security Symposium*. 2024.

[131] LLVM developers. *Clang Static Analyzer*. `https://clang-analyzer.llvm.org/`.

[132] LLVM developers. *libFuzzer – a library for coverage-guided fuzz testing*. `https://llvm.org/docs/LibFuzzer.html`.

[133] Lu, K., Song, C., Lee, B., Chung, S. P., Kim, T., and Lee, W. Aslr-guard: stopping address space leakage for code reuse attacks. In: *ACM Conference on Computer and Communications Security (CCS)*. 2015.

[134] Lütkebohle, I. *AMD Secure Encrypted Virtualization*. `https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf`.

[135] Lyu, Y., Fang, Y., Zhang, Y., Sun, Q., Ma, S., Bertino, E., Lu, K., and Li, J. Goshawk: hunting memory corruptions via structure-aware and object-centric memory operation synopsis. In: *IEEE Symposium on Security and Privacy (S&P)*. 2022.

[136] Ma, X., Luo, L., and Zeng, Q. From one thousand pages of specification to unveiling hidden bugs: large language model assisted fuzzing of matter {iot} devices. In: *USENIX Security Symposium*. 2024.

[137] Ma, Z., Liu, Q., Li, Z., Yin, T., Tan, W., Zhang, C., and Payer, M. Truman: constructing device behavior models from os drivers to fuzz virtual devices. In: *Symposium on Network and Distributed System Security (NDSS)*. 2025.

[138] Ma, Z., Zhao, B., Ren, L., Li, Z., Ma, S., Luo, X., and Zhang, C. Printfuzz: fuzzing linux drivers via automated virtual device simulation. In: *International Symposium on Software Testing and Analysis (ISSTA)*. 2022.

[139] Machiry, A., Spensky, C., Corina, J., Stephens, N., Kruegel, C., and Vigna, G. DR.CHECKER: A Soundy Analysis for Linux Kernel Drivers. In: *USENIX Security Symposium*. 2017.

[140] Malmain, R., Fioraldi, A., and Aurélien, F. Libafl qemu: a library for fuzzing-oriented emulation. In: 2024.

[141] Mason, J., Small, S., Monrose, F., and MacManus, G. English shellcode. In: *ACM Conference on Computer and Communications Security (CCS)*. 2009.

[142] Matsuo, K. *You've Already Been Hacked What if There Is a Backdoor in Your UEFI OROM?* https://i.blackhat.com/BH-US-24/Presentations/US24-Matsuo-Youve-Already-Been-Hacked-What-if-There-Is-a-Backdoor-in-Your-UEFI-OROM-Thursday.pdf.

[143] Mera, A., Chen, Y. H., Sun, R., Kirda, E., and Lu, L. D-box: dma-enabled compartmentalization for embedded applications. In: 2022.

[144] Mera, A., Feng, B., Lu, L., and Kirda, E. Dice: automatic emulation of dma input channels for dynamic firmware analysis. In: *IEEE Symposium on Security and Privacy (S&P)*. 2021.

[145] Mera, A., Liu, C., Sun, R., Kirda, E., and Lu, L. SHiFT: Semi-hosted Fuzz Testing for Embedded Applications. In: *USENIX Security Symposium*. 2024.

[146] michaeljclark. *rv8-bench*. https://github.com/michaeljclark/rv8-bench.

[147] Michał Zalewski. *american fuzzy lop*. https://lcamtuf.coredump.cx/afl/.

[148] Microchip®. *Advanced Software Framework*. https://asf.microchip.com/.

[149] Microsoft. *Hyper-V Technology Overview*. https://learn.microsoft.com/en-us/windows-server/virtualization/hyper-v/hyper-v-overview?pivots=windows-server.

[150] Microsoft. *Microsoft: 70 percent of all security bugs are memory safety issues.* https://www.zdnet.com/inproceedings/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/.

[151] Microsoft. *Pin - A Dynamic Binary Instrumentation Tool.* https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html.

[152] Mishra, S. and Polychronakis, M. SGXPecial: Specializing SGX Interfaces against Code Reuse Attacks. In: *European Workshop on Systems Security*. 2021.

[153] Muench, M., Nisi, D., Francillon, A., and Balzarotti, D. Avatar 2: A multi-target orchestration platform. In: *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research*. 2018.

[154] Myung, C., Lee, G., and Lee, B. MundoFuzz: Hypervisor fuzzing with statistical coverage testing and grammar inference. In: *USENIX Security Symposium*. 2022.

[155] *Nested Vectored Interrupt Controller.* https://developer.arm.com/documentation/ddi0439/b/Nested-Vectored-Interrupt-Controller.

[156] Ngabonziza, B., Martin, D., Bailey, A., Cho, H., and Martin, S. Trustzone explained: architectural features and use cases. In: *International Conference on Collaboration and Internet Computing (CIC)*. 2016.

[157] Nicholas Starke. *U-Boot Fuzzing.* https://starkeblog.com/qemu/u-boot/bootloader/fuzzing/negative-result/2021/03/12/u-boot-fuzzing.html.

[158] OASIS OPEN. *Virtual I/O Device (VIRTIO) Version 1.2.* https://docs.oasis-open.org/virtio/virtio/v1.2/csd01/virtio-v1.2-csd01.html.

[159] Oracle. *Powerful open source virtualization For personal and enterprise use.* https://www.virtualbox.org/.

[160] OSDev Wiki. *Multiboot.* https://wiki.osdev.org/Multiboot.

[161] OSDev Wiki. *UEFI.* https://wiki.osdev.org/UEFI.

[162] Pailoor, S., Aday, A., and Jana, S. {Moonshine}: optimizing {os} fuzzer seed selection with trace distillation. In: *USENIX Security Symposium*. 2018.

[163] Pan, G., Lin, X., Zhang, X., Jia, Y., Ji, S., Wu, C., Ying, X., Wang, J., and Wu, Y. V-shuttle: Scalable and semantics-aware hypervisor virtual device fuzzing. In: *ACM Conference on Computer and Communications Security (CCS)*. 2021.

[164] Park, J., Park, S., Kang, B. B., and Kim, K. Emotion: an sgx extension for migrating enclaves. In: *Computers & Security*. 2019.

[165] Park, J., Park, S., Oh, J., and Won, J.-J. Toward live migration of sgx-enabled virtual machines. In: *IEEE World Congress on Services (SERVICES)*. 2016.

[166] Peng, H. and Payer, M. {Usbfuzz}: a framework for fuzzing {usb} drivers by device emulation. In: *USENIX Security Symposium*. 2020.

[167] Pham, V.-T., Böhme, M., and Roychoudhury, A. Aflnet: a greybox fuzzer for network protocols. In: *International Conference on Software Testing, Validation and Verification (ICST)*. 2020.

[168] Pham, V.-T., Böhme, M., Santosa, A. E., Căciulescu, A. R., and Roychoudhury, A. Smart greybox fuzzing. In: *IEEE Transactions on Software Engineering*. 2019.

[169] Prandini, M. and Ramilli, M. Return-oriented programming. In: *IEEE Symposium on Security and Privacy (S&P)*. 2012.

[170] Priebe, C., Muthukumaran, D., Lind, J., Zhu, H., Cui, S., Sartakov, V. A., and Pietzuch, P. Sgx-lkl: securing the host os interface for trusted execution. In: *arXiv preprint arXiv:1908.11143*. 2019.

[171] Ralf Brown. *BIOS interrupt call*. `https://www.cs.cmu.edu/~ralf/files.html`.

[172] Red Hat Bootloader Team. *shim CSV file fuzzer*. `https://github.com/rhboot/shim/blob/main/fuzz-csv.c`.

[173] Red Hat Bootloader Team. *shim, a first-stage UEFI bootloader*. `https://github.com/rhboot/shim/tree/main`.

[174] Redini, N., Machiry, A., Das, D., Fratantonio, Y., Bianchi, A., Gustafson, E., Shoshitaishvili, Y., Kruegel, C., and Vigna, G. BootStomp: On the security of bootloaders in mobile devices. In: *USENIX Security Symposium*. 2017.

[175] Roderick W. Smith. *The rEFInd Boot Manager*. `https://www.rodsbooks.com/refind/`.

[176] Salls, C., Jindal, C., Corina, J., Kruegel, C., and Vigna, G. {Token-level} fuzzing. In: *USENIX Security Symposium*. 2021.

[177] Scharnowski, T., Bars, N., Schloegel, M., Gustafson, E., Muench, M., Vigna, G., Kruegel, C., Holz, T., and Abbasi, A. Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing. In: *USENIX Security Symposium*. 2022.

[178] Scharnowski, T., Woerner, S., Buchmann, F., Bars, N., Schloegel, M., and Holz, T. Hoedur: Embedded Firmware Fuzzing using Multi-Stream Inputs. In: *USENIX Security Symposium*. 2023.

[179] Schiller, N., Chlosta, M., Schloegel, M., Bars, N., Eisenhofer, T., Scharnowski, T., Domke, F., Schonherr, L., and Holz, T. Drone Security and the Mysterious Case of DJI's DroneID. In: *Symposium on Network and Distributed System Security (NDSS)*. 2023.

[180] Schloegel, M., Bars, N., Schiller, N., Bernhard, L., Scharnowski, T., Crump, A., Ale-Ebrahim, A., Bissantz, N., Muench, M., and Holz, T. SoK: Prudent Evaluation Practices for Fuzzing. In: *IEEE Symposium on Security and Privacy (S&P)*. 2024.

[181] Schumilo, S., Aschermann, C., Abbasi, A., Wör-ner, S., and Holz, T. Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types. In: *USENIX Security Symposium*. 2021.

[182] Schumilo, S., Aschermann, C., Abbasi, A., Wörner, S., and Holz, T. HYPER-CUBE: High-Dimensional Hypervisor Fuzzing. In: *Symposium on Network and Distributed System Security (NDSS)*. 2020.

[183] Schumilo, S., Aschermann, C., Gawlik, R., Schinzel, S., and Holz, T. kAFL:Hardware-Assisted feedback fuzzing for OS kernels. In: *USENIX Security Symposium*. 2017.

[184] Schwarz, M., Gruss, D., Lipp, M., Maurice, C., Schuster, T., Fogh, A., and Mangard, S. Automated detection, exploitation, and elimination of double-fetch bugs using modern cpu features. In: *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. 2018.

[185] Seidel, L., Maier, D., and Muench, M. Forming Faster Firmware Fuzzers. In: *USENIX Security Symposium*. 2023.

[186] Seidel, L., Maier, D., and Muench, M. Forming faster firmware fuzzers. In: *USENIX Security Symposium*. 2023.

[187] Sentinel-One. *efi_fuzz*. `https://github.com/Sentinel-One/efi_fuzz`.

[188] Shafiuzzaman, M., Desai, A., Sarker, L., and Bultan, T. Stase: static analysis guided symbolic execution for uefi vulnerability signature generation. In: *ACM/IEEE International Conference on Automated Software Engineering (ASE)*. 2024.

[189] Shen, Y., Tian, H., Chen, Y., Chen, K., Wang, R., Xu, Y., Xia, Y., and Yan, S. Occlum: secure and efficient multitasking inside a single enclave of intel sgx. In: *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2020.

[190] Shen, Z., Roongta, R., and Dolan-Gavitt, B. Drifuzz: harvesting bugs in device drivers from golden seeds. In: *USENIX Security Symposium*. 2022.

[191] Shinde, S., Le Tien, D., Tople, S., and Saxena, P. Panoply: low-tcb linux applications with sgx enclaves. In: *Symposium on Network and Distributed System Security (NDSS)*. 2017.

[192] Shinde, S., Wang, S., Yuan, P., Hobor, A., Roychoudhury, A., and Saxena, P. Besfs: Mechanized proof of an iago-safe filesystem for enclaves. In: 2018.

[193] Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., and Vigna, G. SoK: State of The Art of War: Offensive Techniques in Binary Analysis. In: *IEEE Symposium on Security and Privacy (S&P)*. 2016.

[194] Smolár, M. *BlackLotus UEFI bootkit: Myth confirmed*. `https://www.welivesecurity.com/2023/03/01/blacklotus-uefi-bootkit-myth-confirmed/`.

[195] Song, D., Hetzelt, F., Das, D., Spensky, C., Na, Y., Volckaert, S., Vigna, G., Kruegel, C., Seifert, J.-P., and Franz, M. Periscope: an effective probing and fuzzing framework for the hardware-os boundary. In: 2019.

[196] Song, D., Hetzelt, F., Kim, J., Kang, B. B., Seifert, J.-P., and Franz, M. Agamotto: accelerating kernel driver fuzzing with lightweight virtual machine checkpoints. In: *USENIX Security Symposium*. 2020.

[197] Sun, H., Shen, Y., Liu, J., Xu, Y., and Jiang, Y. {Ksg}: augmenting kernel fuzzing with system call specification generation. In: *USENIX Annual Technical Conference (ATC)*. 2022.

[198] Surve, P. P., Brodt, O., Yampolskiy, M., Elovici, Y., and Shabtai, A. SoK: Security Below the OS–A Security Analysis of UEFI. In: 2023.

[199] systemd developers. *System and Service Manager*. https://systemd.io/.

[200] Szekeres, L., Payer, M., Wei, T., and Song, D. Sok: eternal war in memory. In: *IEEE Symposium on Security and Privacy (S&P)*. 2013.

[201] Talebi, S. M. S., Tavakoli, H., Zhang, H., Zhang, Z., Sani, A. A., and Qian, Z. Charm: Facilitating dynamic analysis of device drivers of mobile systems. In: *USENIX Security Symposium*. 2018.

[202] Tan, X., Zhang, Y., Lu, J., Xiong, X., Liu, Z., and Yang, M. Syzdirect: directed greybox fuzzing for linux kernel. In: *ACM Conference on Computer and Communications Security (CCS)*. 2023.

[203] Tay, H. J., Zeng, K., Vadayath, J. M., Raj, A. S., Dutcher, A., Reddy, T., Gibbs, W., Basque, Z. L., Dong, F., Smith, Z., et al. Greenhouse:{single-service} rehosting of {linux-based} firmware binaries in {user-space} emulation. In: *USENIX Security Symposium*. 2023.

[204] Team, M. F. *Marlin Firmware*. https://marlinfw.org/.

[205] Team, O. *OpenSSL*. https://www.openssl.org/.

[206] Team, taulab. *high quality open source code for autopilots*. https://github.com/TauLabs/TauLabs.

[207] The kernel development community. *Kernel Self-Protection*. https://www.kernel.org/doc/html/v4.18/security/self-protection.html.

[208] TheHackNews. *Critical Boot Loader Vulnerability in Shim Impacts Nearly All Linux Distros*. https://thehackernews.com/2024/02/critical-bootloader-vulnerability-in.html.

[209] RT-Thread. *A Tiny and Elegant IoT Operating System*. https://www.rt-thread.io//.

[210] tianocore. *EDK II*. https://github.com/tianocore/edk2.

[211] Tsai, C.-C., Porter, D. E., and Vij, M. Graphene-sgx: a practical library {os} for unmodified applications on {sgx}. In: *USENIX Annual Technical Conference (ATC)*. 2017.

[212] UEFI community. *EFI System Table*. https://uefi.org/specs/UEFI/2.10/04_EFI_System_Table.html.

[213] UEFI community. *GUID Partition Table (GPT) Disk Layout*. https://uefi.org/specs/UEFI/2.10/05_GUID_Partition_Table_Format.html.

[214] UEFI community. *Services — Boot Services*. https://uefi.org/specs/UEFI/2.9_A/07_Services_Boot_Services.html.

[215] UEFI community. *Services — Runtime Services*. https://uefi.org/specs/UEFI/2.9_A/08_Services_Runtime_Services.html.

[216] Van Bulck, J., Oswald, D., Marin, E., Aldoseri, A., Garcia, F. D., and Piessens, F. A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. In: *ACM Conference on Computer and Communications Security (CCS)*. 2019.

[217] Vmware. *Desktop Hypervisor*. https://www.vmware.com/products/desktop-hypervisor/workstation-and-fusion.

[218] Vmware. *vSphere*. https://www.vmware.com/products/cloud-infrastructure/vsphere.

[219] Vojdani, V., Apinis, K., Rõtov, V., Seidl, H., Vene, V., and Vogler, R. Static race detection for device drivers: the goblint approach. In: *ACM/IEEE International Conference on Automated Software Engineering (ASE)*. 2016.

[220] Voung, J. W., Jhala, R., and Lerner, S. Relay: static race detection on millions of lines of code. In: *Joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 2007.

[221] Wang, H., Wang, P., Ding, Y., Sun, M., Jing, Y., Duan, R., Li, L., Zhang, Y., Wei, T., and Lin, Z. Towards memory safe enclave programming with rust-sgx. In: *ACM Conference on Computer and Communications Security (CCS)*. 2019.

[222] Wang, P., Krinke, J., Lu, K., Li, G., and Dodier-Lazaro, S. How {double-fetch} situations turn into {double-fetch} vulnerabilities: a study of double fetches in the linux kernel. In: *USENIX Security Symposium*. 2017.

[223] Wang, P., Lu, K., Li, G., and Zhou, X. Dftracker: detecting double-fetch bugs by multi-taint parallel tracking. In: *Frontiers of Computer Science*. 2019.

[224] Wang, Q., Chang, B., Ji, S., Tian, Y., Zhang, X., Zhao, B., Pan, G., Lyu, C., Payer, M., Wang, W., et al. SyzTrust: State-aware Fuzzing on Trusted OS Designed for IoT Devices. In: *IEEE Symposium on Security and Privacy (S&P)*. 2023.

[225] Wen, C., Wang, H., Li, Y., Qin, S., Liu, Y., Xu, Z., Chen, H., Xie, X., Pu, G., and Liu, T. Memlock: memory usage guided fuzzing. In: *International Conference on Software Engineering (ICSE)*. 2020.

[226] Wen, H., Lin, Z., and Zhang, Y. Firmxray: detecting bluetooth link layer vulnerabilities from bare-metal firmware. In: *ACM Conference on Computer and Communications Security (CCS)*. 2020.

[227] WikiLeaks. *EFI Basics: NVRAM Variables*. https://wikileaks.org/ciav7p1/cms/page_26968084.html.

[228] Wikipedia. *Boot sector*. https://en.wikipedia.org/wiki/Boot_sector.

[229] Wikipedia. *Comparison of bootloaders*. https://en.wikipedia.org/wiki/Comparison_of_bootloaders.

[230] Wook Shin Junghwan Kang, H. K. *I Don't Want to Sleep Tonight: Subverting Intel TXT with S3 Sleep*. https://i.blackhat.com/briefings/asia/2018/asia-18-Seunghun-I_Dont_Want_to_Sleep_Tonight_Subverting_Intel_TXT_with_S3_Sleep.pdf.

[231] Wu, Y., Zhang, T., Jung, C., and Lee, D. Devfuzz: automatic device model-guided device driver fuzzing. In: *IEEE Symposium on Security and Privacy (S&P)*. 2023.

[232] Xiang, H., Cheng, Z., Li, J., Ma, J., and Lu, K. Boosting practical control-flow integrity with complete field sensitivity and origin awareness. In: *ACM Conference on Computer and Communications Security (CCS)*. 2024.

[233] Xu, M., Qian, C., Lu, K., Backes, M., and Kim, T. Precise and scalable detection of double-fetch bugs in os kernels. In: *IEEE Symposium on Security and Privacy (S&P)*. 2018.

[234] Xu, W., Moon, H., Kashyap, S., Tseng, P.-N., and Kim, T. Fuzzing file systems via two-dimensional input space exploration. In: *IEEE Symposium on Security and Privacy (S&P)*. 2019.

[235] Yang, Z., Viktorov, Y., Yang, J., Yao, J., and Zimmer, V. Uefi firmware fuzzing with simics virtual platform. In: *Design Automation Conference (DAC)*. 2020.

[236] Yin, J., Li, M., Li, Y., Yu, Y., Lin, B., Zou, Y., Liu, Y., Huo, W., and Xue, J. RSFuzzer: Discovering Deep SMI Handler Vulnerabilities in UEFI Firmware with Hybrid Fuzzing. In: *IEEE Symposium on Security and Privacy (S&P)*. 2023.

[237] Yin, J., Li, M., Wu, W., Sun, D., Zhou, J., Huo, W., and Xue, J. Finding smm privilege-escalation vulnerabilities in uefi firmware with protocol-centric static analysis. In: *IEEE Symposium on Security and Privacy (S&P)*. 2022.

[238] Yin, T., Gao, Z., Xiao, Z., Ma, Z., Zheng, M., and Zhang, C. {Kextfuzz}: fuzzing {macos} kernel {extensions} on apple silicon via exploiting mitigations. In: *USENIX Security Symposium*. 2023.

[239] Yoo, S., Park, J., Kim, S., Kim, Y., and Kim, T. {In-kernel}{control-flow} integrity on commodity {oses} using {arm} pointer authentication. In: *USENIX Security Symposium*. 2022.

[240] Yu, D., Wang, J., Fang, H., Fang, Y., and Zhang, Y. SEnFuzzer: Detecting SGX Memory Corruption via Information Feedback and Tailored Interface Analysis. In: *Symposium on Recent Advances in Intrusion Detection (RAID)*. 2023.

[241] Yu, Y., Chen, Z., Gan, S., and Wang, X. Sgpfuzzer: a state-driven smart graybox protocol fuzzer for network protocol implementations. In: *IEEE Access*. 2020.

[242] Zaddach, J., Bruno, L., Francillon, A., Balzarotti, D., et al. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In: *Symposium on Network and Distributed System Security (NDSS)*. 2014.

[243] Zhao, B., Li, Z., Qin, S., Ma, Z., Yuan, M., Zhu, W., Tian, Z., and Zhang, C. {Statefuzz}: system {call-based}{state-aware} linux driver fuzzing. In: *USENIX Security Symposium*. 2022.

[244] Zhao, J., Li, Y., Zou, Y., Liang, Z., Xiao, Y., Li, Y., Peng, B., Zhong, N., Wang, X., Wang, W., et al. Leveraging semantic relations in code and data to enhance taint analysis of embedded systems. In: *USENIX Security Symposium*. 2024.

[245] Zhao, W., Lu, K., Qi, Y., and Qi, S. Mptee: Bringing flexible and efficient memory protection to intel sgx. In: *European Conference on Computer Systems (EuroSys)*. 2020.

[246] Zheng, Y., Davanian, A., Yin, H., Song, C., Zhu, H., and Sun, L. FIRM-AFL:High-Throughput greybox fuzzing of IoT firmware via augmented process emulation. In: *USENIX Security Symposium*. 2019.

[247] Zheng, Y., Li, Y., Zhang, C., Zhu, H., Liu, Y., and Sun, L. Efficient greybox fuzzing of applications in linux-based iot devices via enhanced user-mode emulation. In: *International Symposium on Software Testing and Analysis (ISSTA)*. 2022.

[248] Zhou, W., Guan, L., Liu, P., and Zhang, Y. Automatic firmware emulation through invalidity-guided knowledge inference. In: *USENIX Security Symposium*. 2021.