A Comprehensive Memory Safety Analysis of Bootloaders

Jianqiang Wang*, Meng Wang*, Qinying Wang†, Nils Langius‡, Li Shi§, Ali Abbasi*, Thorsten Holz*
*CISPA Helmholtz Center for Information Security, †Zhejiang University, ‡Leibniz Universität Hannover, §ETH Zurich
Email: first.last@cispa.de, wangqinying@zju.edu.cn, nils@langius.de, lishil@student.ethz.ch

***Abstract*—The bootloader plays an important role during the boot process, as it connects two crucial components: the firmware and the operating system. After powering on, the bootloader takes control from the firmware, prepares the early boot environment, and then hands control over to the operating system. Modern computers often use a feature called *secure boot* to prevent malicious software from loading at startup. 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 in a secure manner. However, bootloaders have increasingly provided more features and functionalities for end users. As the code base grows, bootloaders inevitably expose more attack surfaces. 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. Despite the bootloader's critical role in the secure boot chain, a comprehensive memory safety analysis of bootloaders has yet to be conducted.**

In this paper, we present the first comprehensive and systematic memory safety analysis of bootloaders, based on a survey of previous bootloader vulnerabilities. We examine the potential attack surfaces of various bootloaders and how these surfaces lead to vulnerabilities. We observe that malicious input from peripherals such as storage devices and networks is a primary method attackers use to exploit bootloader vulnerabilities. To assist bootloader developers in detecting vulnerabilities at scale, we designed and implemented a bootloader fuzzing framework based on our analysis. In our experiments, we discovered 39 vulnerabilities in nine bootloaders, of which 38 are new vulnerabilities. In particular, 14 vulnerabilities were found in the widely used Linux standard bootloader GRUB, some of which can even lead to secure boot bypass if properly exploited. So far, five CVEs have been assigned to our findings.

I. Introduction

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 tasks. 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. As such, the bootloader is a crucial element in the booting process given that it enables a bridge between the firmware and the OS.

Modern computers commonly adopt a security mechanism called *secure boot* [29] 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.

Due to the critical nature of the bootloader, it must be designed and implemented securely. However, bootloaders are gradually providing more features and functionalities for end users. For instance, the Linux default bootloader GRUB [22] supports more than 20 types of file systems. Additionally, it allows users to customize the background image, font, and keyboard layout, as well as download files from HTTP or TFTP servers. Other bootloaders, such as Das U-Boot [16] and barebox [5], face the same situation. The larger the code base becomes, the more vulnerable it gets.

In recent years, memory corruption vulnerabilities have been found in various bootloaders. Some of these vulnerabilities can lead to denial of service or even bypass secure boot. Roee [25] 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 [1]. 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 [15]. The vulnerability originates from an image-parsing library, giving the attacker full control over the system. Similarly researchers reported an HTTP implementation vulnerability [54] in shim [44]: an

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.240330
www.ndss-symposium.org

attacker can exploit an out-of-bound memory write to compromise the entire system. Other bootloader vulnerabilities, such as BootHole [18], 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 [46] and Roee [25] 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 [13] and Starke [39] have proposed fuzzing techniques for GRUB [22] and Das U-Boot [16]. However, their analysis was limited to command-line parsing logic. The attack surfaces of bootloaders go far beyond command-line parsing.

In this paper, 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 own 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. 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:

1) Bootloaders run in a bare metal environment, which means that simple fuzzing frameworks like AFL [37] libFuzzer [34] cannot be directly deployed. Moreover, existing sanitizers cannot be used due to compatibility issues. Previous work [39] 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.
2) 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.

Contributions We make the following two key contributions:

- We perform a comprehensive analysis of bootloaders, with a focus on memory safety violations. We find that three types of peripheral inputs (storage, network, and console) are the main attack surfaces that an attacker can exploit to compromise the bootloader.

- To detect bootloader vulnerabilities on a larger scale, we developed and implemented a bootloader fuzzing framework. Using this framework, we analyzed nine bootloaders and found 38 novel vulnerabilities, 29 of which were confirmed or patched by the developers. To foster research in this area, we open-source our framework prototype at https://github.com/wjqsec/bootloader.

II. Background

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.

A. 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` [35]. 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 [43]. 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 [56] and tries to analyze the partition as a FAT file system. When successful, it searches for the boot application file [41] 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 [55] (a set of function pointers) to provide boot services [57], such as reading files and allocating memory. After initialization of the OS, the firmware remains in memory and provides the OS runtime services [58].

For the rest of the paper, 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".

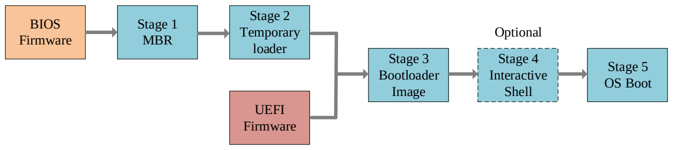

Fig. 1: Schematic overview of bootloader workflow

B. 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 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.

C. 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 [26]. It initializes a simple flat segmentation scheme and keeps paging disabled throughout its life cycle [22, 32, 16]. 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 [23], 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.

D. 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 [40] allow booting into different operating systems. Users have the option to select a location or an image file from which to start the boot process.

III. 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.

A. 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 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 2 in dark green, pink, and yellow, respectively. For example, vulnerabilities related to

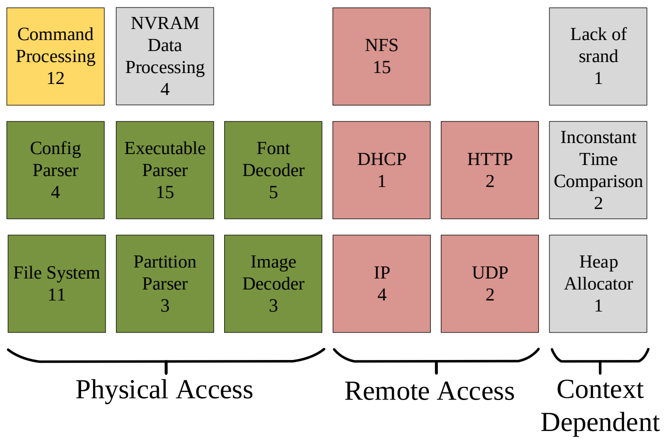

Fig. 2: Number and root causes of collected CVEs

TABLE I: Number and distribution of collected 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

the file system and file parser are linked to storage, while IP packet issues are associated with the network. As shown in Table I, of the 85 CVEs from four 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.

B. 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 [27], Intel TDX [28], and ARM TrustZone [2] address this vulnerability. Finding vulnerabilities in TEE software [10] [68] [60] is beyond the scope of this paper.

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 [51] can manipulate NVRAM variables from the OS, we exclude them from our threat model. However, if the bootloader implements an NVRAM [62] 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 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 paper.

C. Target Selection

In this paper, we analyze nine bootloaders selected from a list of available bootloaders [63]. 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 [22] has several versions, but we specifically choose GRUB2 as our analysis target).

Based on these criteria, we collected nine bootloaders as shown in Table II. These include widely used bootloaders

TABLE II: 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 paper writing, developers might add more features after that.

	Version	# of Source Files	Image Size	Supported Targets	Firmware	CI	Last Update
GRUB [22]	v2.02-beta2	5411	297KB	Linux, GNU/Hurd, macOS, BSD Solaris/illumos (x86 port), Windows	BIOS,UEFI	✓	2024.05
Limine [32]	v7.x	442	105KB	Linux	BIOS	-	2024.05
Das U-Boot [16]	v2024.04-rc3	11048	1.0MB	Linux, NetBSD, VxWorks, QNX RTEMS, INTEGRITY	BIOS	✓	2024.05
barebox [5]	v2024.01	4878	681KB	RTOSes	UEFI	✓	2024.05
CloverBootloader [11]	v2-5158	9048	1.6MB	macOS	UEFI	-	2024.05
Easyboot [9]	v1.0.0	47	69KB	Linux, Windows, OpenBSD, FreeBSD, FreeDOS ReactOS, MenuetOS, KolibriOS, SerenityOS, Haiku	UEFI	-	2024.04
rEFInd [47]	v0.14.3	173	306KB	Linux, Windows, macOS, TrueOS	UEFI	-	2024.04
systemd-boot [53]	v256	92	213KB	Linux	UEFI	-	2024.04
shim [44]	v15.8	546	936KB	GRUB	UEFI	-	2024.05

such as GRUB, Das U-Boot, and systemd-boot. They 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.

D. 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 VI and focus on the main three attack surfaces.

1) Storage: As shown in Table III, 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 [64]. 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:

a) 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 III, common image types supported by bootloaders include PNG, JPEG, and BMP.

b) 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.

TABLE III: Bootloader attack surface analysis for storage device input

	GRUB	Limine	Das U-Boot	barebox	CloverBootloader	Easyboot	rEFInd	systemd-boot	shim
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	base64, svg, png icns, bmp, png	config	png, jpeg, bmp icns	bcd, config	csv
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	hfs, iso9660, ext2fs ext4fs, reiserfs, fatfs	afs, befs, exfatfs ext234fs, minix3fs ntfs, ufs, xfs fatfs, fszfs	btrfs, ext2fs, ext4fs hfs, iso9660, reiserfs ntfs	-	-
Partition	Linux/ADFS, amiga disklabel64 macintosh, GPT MS-DOS, SUN SUN PC, BSDlabel	GPT, MS-DOS	amiga, GPT MS-DOS, macintosh	GPT, MS-DOS	-	-	-	-	-

c) 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.

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 IV, 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) 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 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.

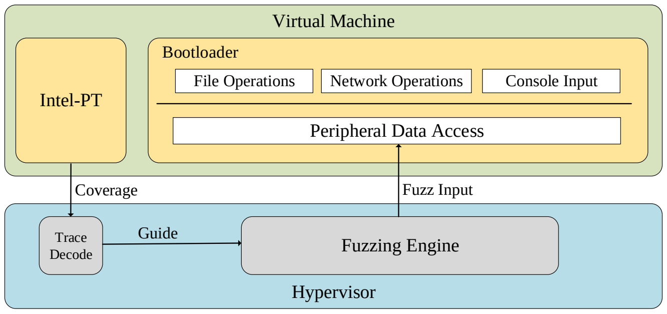

Fig. 3: Bootloader fuzzing overview

IV. Bootloader Fuzzing 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 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 guide 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.

A. Harness

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

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.

TABLE IV: Bootloader attack surface analysis for network device input. 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	-	-	-	-	HTTP wrapper
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	CloverBootloader	Easyboot	rEFInd	systemd-boot	shim

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.

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.

B. Fuzzing Engine

We implement our fuzzing framework based on kAFL [50, 48], 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.

C. *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 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.

V. Evaluation

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

TABLE V: 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	✓

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?

A. *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.

B. ***RQ1*** *Reproducibility of Known Vulnerabilities*

As shown in Table V, we collected seven recently available bootloader vulnerabilities that can be compiled for the Intel x86 architecture. These vulnerabilities span the three main attack surfaces. The vulnerabilities detail can be found in APPENDIX A. Our fuzzing framework was able to successfully reproduce them within several hours. 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.

C. ***RQ2*** *Finding New Vulnerabilities*

During our evaluation, we found 39 vulnerabilities, of which 38 were previously unknown. Table VI 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.

TABLE VI: 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

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.

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.

D. ***RQ3*** *Comparison with Other Works*

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* [4] and *Clang Static Analyzer* [33]. 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 VII shows the vulnerability detection result of the nine bootloaders.

TABLE VII: 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

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.

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) 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).

TABLE VIII: Number of modified or added lines of code

	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

E. ***RQ4*** *Manual Effort*

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 VIII 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 are able to implement the harness in an efficient way. 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.

VI. 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 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 paper, 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 [64] 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 environments, such as SHiFT [36], 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 [14, 17]. 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 [12, 45]. 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:

a) 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.

b) 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 paper, in the development of fuzzing strategies to guide and improve them.

Comparison with Existing Works The two fuzzing tools introduced by Axtens [13] and Starke [39] 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.

VII. Related Work

Compared to normal applications, low-level software is more closely integrated with the hardware. Due to its intrinsic nature of directly manipulating hardware, significant security issues can arise, potentially compromising the entire system. In recent years, the security of low-level software has garnered increasing attention.

Firmware is usually the first component executed during the system boot process. Modern UEFI firmware provides the OS with runtime services to manage hardware. While these richer features bring convenience, they also introduce potential vulnerabilities. Spender [67] applies static analysis to the System Management Mode (SMM) service, which operates at ring -2, a higher privilege level than the OS. Spender constructs a comprehensive inter-driver call graph based on the protocol producer and consumer implementations. By tracking the value flow within the call graph, SPENDER identifies privilege escalation vulnerabilities in the UEFI firmware. Yang et al. [65] 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 [66]. 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. [6] targeted SMM interrupt handler variables using symbolic execution, but faced common challenges such as path explosion. Surve et al. [52] 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 [25] identified several command line parser vulnerabilities in commercial Android devices. BootStomp [46] performs taint analysis on bootloader binaries of mobile device. 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.

Before the OS gains control of the system, another optional component, the hypervisor, may come into play. The hypervisor can introspect the OS, and if compromised by an attacker, it can lead to virtual machine escape or even full system control. Fuzzing techniques have been applied to the hypervisor to identify vulnerabilities. Hyper-Cube [49] designs a custom mini OS running inside the virtual machine that can feed fuzz inputs to the hypervisor. V-shuttle [42] addresses the DMA access problem in hypervisor fuzzing, while MundoFuzz [38] filters out noise from coverage feedback and infers input structures. Morphuzz [8] infers the I/O behaviors of real-world virtual devices and discovered new vulnerabilities in both QEMU and bhyve. In another threat model, the hypervisor itself may not be trusted: since it has higher privileges than the OS, a hardware mechanism such as a Trusted Execution Environment (TEE) is used to protect the OS data. Despite efforts to design and implement secure TEEs such as VirTEE [59], Penglai [19], and Keystone [31], memory corruption vulnerabilities persist. SEnFuzzer [68], SGXFuzz [10], and SyzTrust [60] apply fuzzing to both open-source and commercial TEEs. Since TEE memory cannot be directly accessed, these tools propose various methods to extract coverage feedback to guide the fuzzing process.

From various low-level software security analyses, we have learned that fuzzing has been widely adopted and has proven effective in detecting new vulnerabilities. General fuzzing tools such as AFL [37] and LibAFL [21] are continuously integrating more techniques, including cmplog [3], coverage accounting [61], weighted scheduling [20], and power scheduling [7]. These techniques and methods apply for both specific domain and general application fuzzing. They focus on solving the common problems such as hard-to-bypass control flow obstacles and seed selection. Generating harnesses, such as Winnie [30] and FAST [24], are also beneficial in fuzzing new targets.

VIII. Conclusion

In this paper, we have presented a comprehensive memory safety analysis of bootloaders. Our study identified the three main attack surfaces (storage, network, and console) in bootloaders based on a survey of 85 CVEs. We also introduced a fuzzing framework, which we used to analyze nine bootloaders in detail. To facilitate this process, we intercepted peripheral access at the source code level, enabling us to feed fuzz inputs and manipulate peripheral operations to effectively trigger the fuzzing process. Our experiments uncovered 39 vulnerabilities, with 29 of these being confirmed or patched by the 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.

Acknowledgement

This work was funded by the European Research Council (ERC) under the consolidator grant RS^3 (101045669).

References

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

[2] 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.

[3] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with Input-to-State Correspondence. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[4] Pavel Avgustinov, Oege De Moor, Michael Peyton Jones, and Max Schäfer. QL: Object-oriented queries on relational data. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, 2016.

[5] barebox developers. Barebox. https://www.barebox.org/.

[6] Oleksandr Bazhaniuk, John Loucaides, Lee Rosenbaum, Mark R Tuttle, and Vincent Zimmer. Symbolic Execution for BIOS Security. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2015.

[7] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[8] Alexander Bulekov, Bandan Das, Stefan Hajnoczi, and Manuel Egele. Morphuzz: Bending (input) space to fuzz virtual devices. In *USENIX Security Symposium*, 2022.

[9] bzt. easyboot. https://gitlab.com/bztsrc/easyboot/.

[10] Tobias Cloosters, Johannes Willbold, Thorsten Holz, and Lucas Davi. SGXFuzz: Efficiently synthesizing nested structures for SGX enclave fuzzing. In *USENIX Security Symposium*, 2022.

[11] CloverBootloader developers. CloverBootloader. https://github.com/CloverHackyColor/CloverBootloader/.

[12] Coverity. Coverity Scan: barebox. https://scan.coverity.com/projects/barebox.

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

[14] Daniel Axtens. Fuzzing grub, part 2: going faster. https://sthbrx.github.io/blog/2021/06/14/fuzzing-grub-part-2-going-faster/.

[15] 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.

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

[17] Das U-Boot Developers. Das U-Boot Sanitizer Compilation Kconfig. https://github.com/u-boot/u-boot/blob/master/Kconfig#L157.

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

[19] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Scalable Memory Protection in the PENGLAI Enclave. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.

[20] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining incremental steps of fuzzing research. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

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

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

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

[24] Jiong Gong, Yun Wang, Haihao Shen, Xu Deng, Wei Wang, and Xiangning Ma. FAST: Formal specification driven test harness generation. In *Tenth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMCODE2012)*, 2012.

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

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

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

[28] 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.

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

[30] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. Winnie: Fuzzing windows applications with harness synthesis and fast cloning. In *Symposium on Network and Distributed System Security (NDSS)*, 2021.

[31] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *European Conference on Computer Systems (EuroSys)*, 2020.

[32] Limine developers. Limine. https://limine-bootloader.org/.

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

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

[35] Budhaditya Majumdar. Boot sector programming. 2007.

[36] Alejandro Mera, Changming Liu, Ruimin Sun, Engin Kirda, and Long Lu. SHiFT: Semi-hosted Fuzz Testing for Embedded Applications. In *USENIX Security Symposium*, 2024.

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

[38] Cheolwoo Myung, Gwangmu Lee, and Byoungyoung Lee. MundoFuzz: Hypervisor fuzzing with statistical coverage testing and grammar inference. In *USENIX Security Symposium*, 2022.

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

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

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

[42] Gaoning Pan, Xingwei Lin, Xuhong Zhang, Yongkang Jia, Shouling Ji, Chunming Wu, Xinlei Ying, Jiashui Wang, and Yanjun Wu. V-shuttle: Scalable and semantics-aware hypervisor virtual device fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2021.

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

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

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

[46] Nilo Redini, Aravind Machiry, Dipanjan Das, Yanick Fratantonio, Antonio Bianchi, Eric Gustafson, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. BootStomp: On the security of bootloaders in mobile devices. In *USENIX Security Symposium*, 2017.

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

[48] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wör-ner, and Thorsten Holz. Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types. In *USENIX Security Symposium*, 2021.

[49] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. HYPER-CUBE: High-Dimensional Hypervisor Fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2020.

[50] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL:Hardware-Assisted feedback fuzzing for OS kernels. In *USENIX Security Symposium*, 2017.

[51] Sentinel-One. efi_fuzz. https://github.com/Sentinel-One/efi_fuzz.

[52] Priyanka Prakash Surve, Oleg Brodt, Mark Yampolskiy, Yuval Elovici, and Asaf Shabtai. SoK: Security Below the OS–A Security Analysis of UEFI. *arXiv preprint arXiv:2311.03809*, 2023.

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

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

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

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

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

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

[59] Jianqiang Wang, Pouya Mahmoody, Ferdinand Brasser, Patrick Jauernig, Ahmad-Reza Sadeghi, Donghui Yu, Dahan Pan, and Yuanyuan Zhang. VirTEE: A full backward-compatible TEE with native live migration and secure I/O. In *Design Automation Conference (DAC)*, 2022.

[60] Qinying Wang, Boyu Chang, Shouling Ji, Yuan Tian, Xuhong Zhang, Binbin Zhao, Gaoning Pan, Chenyang Lyu, Mathias Payer, Wenhai Wang, et al. SyzTrust: State-aware Fuzzing on Trusted OS Designed for IoT Devices. In *IEEE Symposium on Security and Privacy (S&P)*, 2023.

[61] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization. In *Symposium on Network and Distributed System Security (NDSS)*, 2020.

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

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

[64] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing file systems via two-dimensional input space exploration. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[65] Zhenkun Yang, Yuriy Viktorov, Jin Yang, Jiewen Yao, and Vincent Zimmer. Uefi firmware fuzzing with simics virtual platform. In *Design Automation Conference (DAC)*, 2020.

[66] Jiawei Yin, Menghao Li, Yuekang Li, Yong Yu, Boru Lin, Yanyan Zou, Yang Liu, Wei Huo, and Jingling Xue. RSFuzzer: Discovering Deep SMI Handler Vulnerabilities in UEFI Firmware with Hybrid Fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*, 2023.

[67] Jiawei Yin, Menghao Li, Wei Wu, Dandan Sun, Jianhua Zhou, Wei Huo, and Jingling Xue. Finding smm privilege-escalation vulnerabilities in uefi firmware with protocol-centric static analysis. In *IEEE Symposium on Security and Privacy (S&P)*, 2022.

[68] Donghui Yu, Jianqiang Wang, Haoran Fang, Ya Fang, and Yuanyuan Zhang. SEnFuzzer: Detecting SGX Memory Corruption via Information Feedback and Tailored Interface Analysis. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2023.

Appendix

A. CVE details

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.