

A Survey on the Evolution of Bootkits Attack and Defense Techniques

Yilin Zhou^{1,2}, Guojun Peng^{1,2,*}, Zichuan Li^{1,2}, Side Liu^{1,2}

¹ Key Laboratory of Aerospace Information Security and Trust Computing, Ministry of Education, Wuhan 430072, China

² School of Cyber Science and Engineering, Wuhan University, Wuhan 430072, China

* The corresponding author, email: guojpeng@whu.edu.cn

Cite as: Y. Zhou, G. Peng, *et al.*, “A survey on the evolution of bootkits attack and defense techniques,” *China Communications*, 2024, vol. 21, no. 1, pp. 102-130. **DOI:** 10.23919/JCC.ja.2022-0409

Abstract: According to the boot process of modern computer systems, whoever boots first will gain control first. Taking advantage of this feature, a malicious code called bootkit can hijack the control before the OS bootloader and bypass security mechanisms in boot process. That makes bootkits difficult to detect or clean up thoroughly. With the improvement of security mechanisms and the emergence of UEFI, the attack and defense techniques for bootkits have constantly been evolving. We first introduce two boot modes of modern computer systems and present an attack model of bootkits by some sophisticated samples. Then we discuss some classic attack techniques used by bootkits from their initial appearance to the present on two axes, including boot mode axis and attack phase axis. Next, we evaluate the race to the bottom of the system and the evolution process between bootkits and security mechanisms. At last, we present the possible future direction for bootkits in the context of continuous improvement of OS and firmware security mechanisms.

Keywords: bootkit; hook; legacy BIOS; security mechanisms; UEFI

I. INTRODUCTION

Recently, some novel bootkits have caught the attention of security researchers by showing huge attack surfaces of ESP (i.e., EFI system partition) and EFI bootloader [1–3]. But bootkit is not new technology. On the contrary, they appeared very early. Moreover, their techniques have also been evolving with the evolution of the modes computers boot and corresponding security mechanisms. Before discussing bootkits techniques in detail, we first introduce similar but better-known malicious code called “rootkit.” Their names are similar, and their techniques also have something in common. Rootkits have plagued many computer systems [4] and IoT devices for many years. A rootkit is a kind of malicious code designed to provide an attacker with privileged access to a device without victim awareness [5–7]. The original design intention of rootkits and bootkits is basically the same. They were all designed to allow the attacker to stay in the victim system concealed to remain in control of the victim system for a long time. TDL3 is a typical rootkit distributed through a PPI (i.e., Pay-Per-Install) business model via the affiliates Dogma Millions and GangstaBucks [8]. It has a hook module as an infector and an independently developed hidden file system. The way it achieves long-term persistence is to select a boot-start driver randomly, inject malicious code into its resource section and modify the undocumented structure `_KLDR_DATA_TABLE_ENTRY` in the Dri-

Received: Jun. 27, 2022

Revised: Nov. 18, 2022

Editor: Guoru Ding

verSection field. The EntryPoint field then points to the code of the injected resource section. Every time the OS (i.e., Operating System) boots, all boot-start drivers will be loaded into the kernel space, and the malicious code of TDL3 will be executed. Therefore, TDL3 has achieved the goal of automatically executing every time the system boots.

This situation changed after Microsoft Windows released the kernel-mode code signing policy [9]. Kernel-Mode Code Signing Policy verifies the driver's signature in kernel mode and prevents drivers without legal signatures from being loaded into the kernel. Therefore, the self-starting rootkits similar to TDL3 could not work in the version after Windows XP by simply and directly modifying the boot-start drivers.

The first modern bootkit was considered BootRoot [10]. This concept was first mentioned by eEye Digital in their "BootRoot" project in 2005. The most significant difference between bootkits and rootkits lies in how they realize automatic startup. Rootkits' self-starting techniques do not destroy the integrity of the control flow of OS booting. Bootkits usually execute before the OS and then disable the verification of the driver signature by patching the OS bootloader. This method disrupts the system boot sequence and modifies the OS component simultaneously. So bootkits can break through the restriction that the kernel-mode code signing policy puts on rootkits.

With the emergence of UEFI (i.e., Unified Extensible Firmware Interface) [11], bootkits attack techniques have also changed. In 2018, ESET released an analysis report on LoJax [12], which made more security researchers pay attention to the bootkits techniques in UEFI boot mode. In fact, open-source UEFI bootkits such as DreamBoot [13], ThunderStrike [14], SMMbackdoor [15], etc., have appeared for research purposes since 2013. These open-source bootkits did not attract widespread attention at the time. However, on the other hand, it has promoted the development of malicious bootkit techniques in the wild [16]. Many malicious bootkits captured in the wild have been analyzed and confirmed to be based on the open-source bootkit and improved after partial module reuse [17, 3]. Compared with legacy BIOS, UEFI did not leave bootstrap code like MBR, VBR (i.e., Volume Boot Record), or IPL (i.e., Initial Program Loader) on the disk but stored the UEFI firmware code responsible for booting and initializing the platform hardware

in the SPI (i.e., Serial Peripheral Interface) flash memory [18]. Modifying the firmware was far more difficult than modifying the disk. Cleaning the bootkits in the firmware was also far more challenging. So bootkits needed to develop a new technique to counter the OEM/ODM manufacturers' security mechanism for firmware. Intel has introduced security mechanisms such as Secure Boot and Boot Guard to ensure that the firmware would not be tampered with by attackers. Bootkits have also begun to utilize SMM vulnerabilities [15, 19, 20], S3 scripts vulnerabilities [21–24], BMC vulnerabilities [25], Intel AMT vulnerabilities [26, 27] and Intel ME vulnerabilities [28–31] to attack UEFI firmware so as to achieve the initial mission of permanent residence and maintain the control.

This work mainly focuses on the evolution of bootkit's attack techniques and corresponding security mechanisms. However, we will mainly focus on the attack techniques of bootkits, supplemented by the security mechanisms for the related attacks. As illustrated in Figure 1, we summarize 12 classic bootkits attack techniques (i.e., A_1 – A_{12}) and six representative defense techniques (i.e., D_1 – D_6), which are the most discussed in various security conferences and papers, and also the most characteristic of bootkit attacks. For every attack technique, we annotate a few typical instances. They are not comprehensive, and some bootkits involve multiple attack techniques at the same time, but each of them is representative of its category. We divide all the above techniques into four parts by our proposed two axes. On attack phase axis, we divide techniques into infection phase and boot phase. Techniques in infection phase focus on how to attack or protect the storage media bootkits need to implant, while techniques in boot phase focus on how to promote or prevent bootkits from hijacking the control earlier than the OS bootloader or bypassing the integrity check. The closer it is to the arrow, the later the attack or defense technique works during the boot process. If an attack technique is below a defense technique in Figure 1, it often means the attack technique can bypass or undermine the defense technique, and vice versa (e.g., D_6 can protect RoT from A_7 , but may be undermined by A_9). On boot mode axis, we divide techniques into legacy BIOS and UEFI. The boot mode axis is sort of like a timeline, but not quite. It divides techniques into legacy BIOS and UEFI. Some techniques (e.g., A_8 , D_2 and D_3) works in both boot

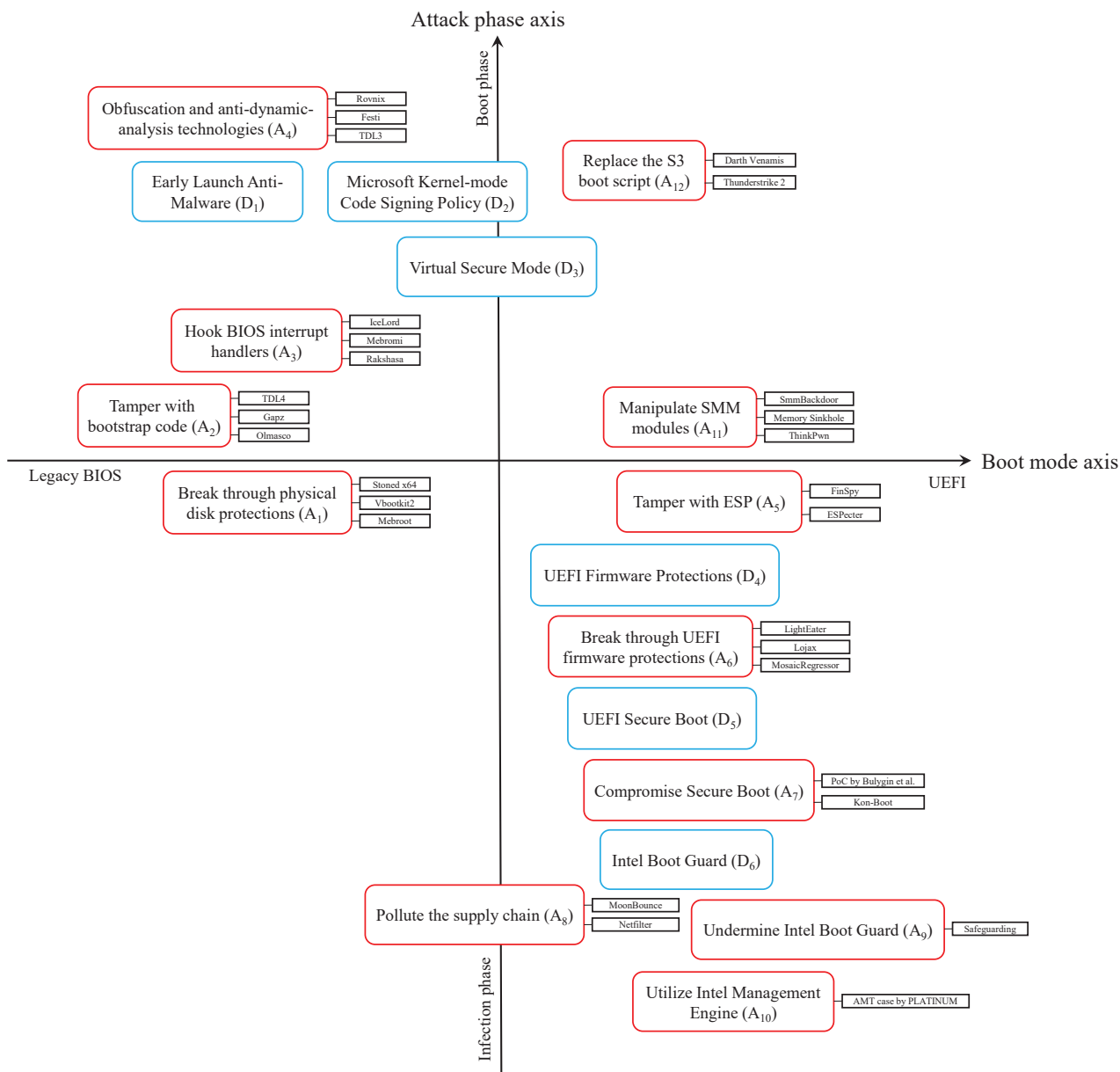


Figure 1. Bootkit attack techniques classified by two axes. The red box represents the attack techniques used by bootkits, and the blue box represents the defense techniques associated with bootkit.

modes, so we put them across the axis in Figure 1. As can be seen from the distribution of attack and defense techniques in Figure 1, techniques in legacy BIOS are mainly distributed in boot phase. In contrast, techniques in UEFI are mainly distributed in infection phase. This is primarily because legacy BIOS relies a lot on the OS boot security mechanisms against bootkits, while UEFI does a lot of work to verify the integrity of the firmware storage media. Each of these attack or defense techniques will be described in detail in the following sections.

The rest of this paper is organized as follows. Section II is organized along the boot mode axis. It presents two mainstream boot modes of the computer system named legacy BIOS and UEFI, as well as the points where bootkits attacks often occur. We believe it will be more helpful to understand the malicious behavior of bootkits in two different boot modes. Section III is organized along the attack phase axis. We present an attack model of a bootkit and its vital functional modules in infection phase and boot phase. It is based on our summary and generalization of several

typical bootkits analysis reports. Section IV mainly focuses on bootkits in legacy BIOS boot mode, and Section V focuses on bootkits in UEFI boot mode. In each section, we introduce the attack techniques of bootkits from the perspective of infection phase and boot phase, as well as a summary of corresponding security mechanisms. For bootkits in legacy BIOS, we introduce four approaches they used to hijack the control before OS booting, the hook method they utilized to transfer the control, and OS vendors' countermeasures against these attacks. For bootkits in UEFI, we introduce the process that they used various state-of-the-art attack methods to tamper with RoT (i.e., Root of Trust) and executed malicious code under the strict firmware security mechanisms by utilizing vulnerabilities in multiple components in firmware, as well as the process that firmware manufacturers moved RoT from the DXE stage to the hardware [32–34] in order to mitigate such security issues. In Section VI, we discuss the process of co-evolution between the bootkit attack techniques and the security mechanisms. We also analyze the possible attack surface and direction of development for bootkits in the future firmware system dominated by UEFI. Finally, we make our conclusion in Section VII.

In general, our main contributions can be summarized as follows:

- We comprehensively review the construction of bootkits and summarize the first bootkit attack model based on many typical bootkits cases. Compared to previous work [16, 6, 7, 35], the model we propose is the first comprehensive, structured attack model for bootkits.
- We provide a new perspective by evaluating the techniques used by bootkits along two axes. On boot mode axis, we divide bootkits into legacy BIOS and UEFI. On attack phase axis, we discuss bootkits techniques from infection phase and boot phase, respectively. Currently, the research on bootkits is mainly scattered in the analysis report of each sample, lacking a systematic and comprehensive review, especially the review of bootkits techniques in UEFI. And our research fills this gap.
- We analyze the co-evolution of bootkits attack and defense techniques in the past few years. Based on these, we propose the future attack

surface and possible development direction for bootkits. We believe bootkit attacks will have to rely on the latest underlying software vulnerabilities or OEM manufacturers' flaws in the implementation of security mechanisms, and the attack targets will change from PCs to IoT devices. Simultaneously, for the sake of economic interests, the authors of bootkits will make legalized use of their techniques to produce legalized commercial software (e.g., Computrace [36], and Kon-Boot [37]) and sell them publicly.

II. TWO MAINSTREAM BOOT MODES OF COMPUTER SYSTEM

The boot mode of the computer system is critical for bootkits, and the first axis we propose to characterize bootkits is boot mode. Both bootkit developers and security mechanisms developers hope to gain control at an earlier stage of the boot process. Nowadays, there are two mainstream boot modes (i.e., legacy BIOS and UEFI). Legacy BIOS was born in the CP/M computer in 1975 [17]. Nevertheless, it was an advanced technique and an indispensable part of the system at that time. With the development of computer software and hardware, the defects of legacy BIOS have also been exposed. Problems such as low efficiency, poor scalability, and poor security have made legacy BIOS gradually challenging to meet the needs of modern computers. The predecessor of UEFI was the Intel Boot Initiative developed by Intel in 1998 and was later renamed UEFI (i.e., Unified Extensible Firmware Interface). UEFI has gradually replaced the dominant position of legacy BIOS. Modern computers basically use the UEFI boot mode, but there are still lots of older computers that utilize the legacy BIOS boot mode. Bootkits have also gone through the process from legacy BIOS to UEFI. Next, we will discuss the processes and characteristics of these two boot modes. Simultaneously, we will also briefly outline the possible attack points for bootkits in their boot processes. We mainly take Windows operating system as an example. However, the boot process of Linux and other IoT devices is basically the same before their operating system officially runs.

2.1 Legacy BIOS Boot Mode

System booting in legacy BIOS mode can be briefly summarized as the following steps:

1. After powering on, the ROM BIOS code performs POST (i.e., power-on self-test) and hardware initialization operations.
2. Read the data in the first sector of the hard disk (i.e., MBR) into the physical memory address 0x7C00.
3. Execute the first 446 bytes of MBR as code, analyze the MBR partition table to find the active partition, and transfer the control to the first sector of the active partition (i.e., VBR).
4. Execute the code in the VBR and the 15 sectors immediately after the VBR to complete the analysis of the active partition file system.
5. Look for the OS bootloader in the active partition, such as bootmgr in Windows. Then run the bootloader, switch the CPU mode from 16-bit to 32/64-bit, and load the OS kernel until OS runs normally.

Bootmgr is a part of OS. So bootkit must hijack the control before bootmgr execution. Most bootkits in legacy BIOS attack the bootstrap code, which means the code is executed before the OS component. Figure 2 shows the structure of MBR, VBR, and the control flow of bootstrap code.

MBR mainly consists of MBR code and partition table. Only one of the partition table entries is marked active. For example, we assume that the third entry in Figure 2 is active. MBR will read and calculate the beginning address of the active partition. Then the control is transferred to the first sector of the active partition called VBR. Both MBR and VBR are 512 bytes. The first three bytes of VBR are a “JMP” instruction, which skips the BPB (i.e., BIOS Parameter Block) structure and executes the VBR code. The VBR code reads the BPB and locates the IPL (i.e., Initial Program Loader) according to the offset in the structure of the BPB. IPL’s mission is to access the file system of the current partition and transfer the control to the OS bootloader, such as bootmgr. The red crosses in Figure 2 represent possible bootkit attack points, which will be discussed in detail in Section 4.2.1.

The process described above is the control flow of the bootstrap code independent of the type of OS.

Bootstrap code will transfer the control to the OS bootloader. The original OS bootloader in Windows was called NTLDR. After Windows Vista, the legacy NTLDR was split into two modules: bootmgr and winload.exe (or winresume.exe if the OS is loaded from hibernation). Bootmgr also consists of two modules: the code in 16-bit and a compressed PE image. The 16-bit code switches the CPU from the real mode to the protected mode, and decompresses the PE image, then executes it. After that, the bootmgr will read the BCD (i.e., Boot Configuration Data), checks the integrity of itself, and calls winload.exe to load the OS kernel.

The boot process in legacy BIOS mode is critical for bootkits and security developers. Attack of bootkits often happens in the bootstrap code and bootloader because it’s too late for bootkits to launch an attack after the OS kernel has been loaded.

2.2 UEFI Boot Mode

System booting in UEFI mode is very different from it in legacy BIOS mode. UEFI specification has stipulated that the code responsible for hardware self-check and initialization were all stored in the firmware located in the SPI flash memory chip. So there is no bootstrap code on the hard disk for bootkits to tamper with.

As shown in Figure 3 [38], the boot process in UEFI mode consists of five stages, which are SEC (i.e., Security), PEI (i.e., Pre EFI Initialization), DXE (i.e., Driver Execution Environment), BDS (i.e., Boot Device Select), and TSL (i.e., Transient System Load). The RT (i.e., Run Time) provides some runtime services for OS running. The AL (i.e., After Life) is for the situation where the system is catastrophically damaged and whose specific actions have not been specified in UEFI specifications. Both RT and AL have nothing to do with the boot process, so we will not discuss them in detail here.

The five stages of the UEFI boot mode have their division of labor:

1. **SEC:** Verify the RoT and utilize Cache-as-RAM before the main RAM is activated.
2. **PEI:** Prepare the environment and HOB (i.e., Hand-Off Block) list for DXE. The PEI dispatcher will schedule PEIM (i.e., PEI Module) to init the CPU, chipset, and board.

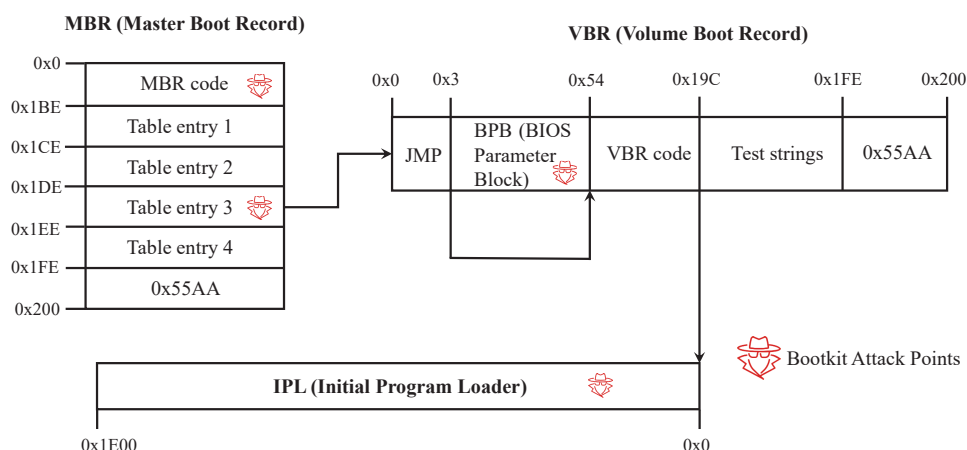


Figure 2. Control flow of bootstrap code.

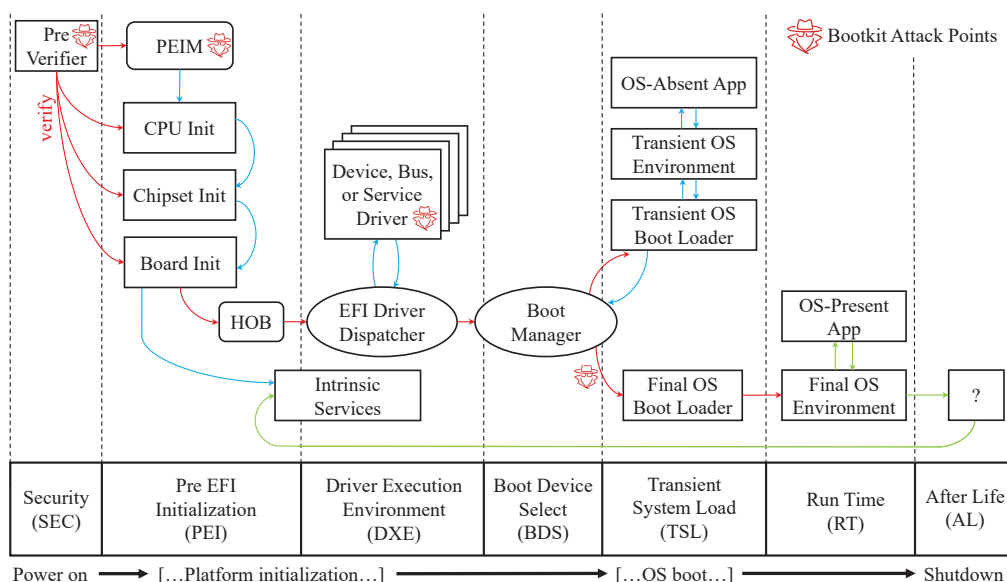


Figure 3. The architecture execution flow of UEFI. The red arrows represent the integrity verification processes, the blue arrows represent the control flow transfers during UEFI system initialization, and the green arrows represent the control flow transfers after the system boot.

- 3. DXE:** The main module to execute the UEFI drivers and modules. The DXE dispatcher executes each driver according to the list named mDiscoveredList, which records the drivers that were important to initialize the essential devices.
- 4. BDS:** Execute boot items according to a particular policy by reading UEFI variables in NVRAM (i.e., Non-Volatile RAM), which is equivalent to selecting the OS to boot in most instances.
- 5. TSL:** The first stage that the OS bootloader executes as a UEFI application. If any severe error happens or the user intervenes, it would enter the

UEFI shell. After the OS loader calls the Exit-BootServices function, the TSL stage ends, and the system enters the RT stage. All the system resource begins to transfer from UEFI core to the OS kernel.

From the functional division of these stages, it can be inferred that there are some stages that are critical to the security of the UEFI boot process. The SEC, PEI, and DXE stages are all vital points where the bootkit will try to attack and tamper. The attacks to Boot Guard and ME (i.e., A_9 and A_{10}) even happen before the SEC stage. PEI and DXE are the main

stages of the UEFI boot process where UEFI drivers and applications are executed, so many bootkits will inject the payload into these two stages. Also, replacing the OS bootloader between BDS and TSL stages is also a common attack vector for bootkits. The details and other bootkit attack techniques in UEFI will be discussed in Section V.

Compared with legacy BIOS, UEFI boot mode is faster and safer. Their main difference is shown in Table 1. From the perspective of bootkit developers, the differences between legacy BIOS and UEFI mode systems are as follows:

- UEFI only loads the drivers necessary for initializing the essential hardware in the boot process rather than initializing all the hardware in legacy BIOS. Bootkits need to select the appropriate hook driver in a targeted manner.
- Utilizing the larger physical memory, most UEFI code runs in 32/64-bit mode rather than the 16-bit mode in legacy BIOS. Larger address space means increased attack difficulty.
- In legacy BIOS, the disk I/O and bootloader interaction are implemented by BIOS interrupts. In UEFI, these are all implemented by UEFI services. Bootkits must adopt a different control hijacking strategy.

From the security development point of view, legacy BIOS does not have a dedicated safety mechanism to protect the boot process from bootkit attacks because its architecture was designed a long time ago and was never changed. Bootmgr can verify itself, but it is executed after the bootstrap code, which can be modified by bootkits without verification or warning. Secure Boot in UEFI guarantees that every stage in the boot process will verify the code integrity of the next stage, such as SEC will verify the integrity of PEI before it transfers the control to it. Therefore, Secure Boot can theoretically ensure the integrity of the entire UEFI boot process under the condition of RoT credibility.

III. ATTACK MODEL OF BOOTKITS

After summarizing the analysis reports of some sophisticated bootkit samples [39, 40, 14, 13, 12, 41–44], we found that they had something in common in the structure that applied to most of the bootkits in legacy BIOS and some of the bootkits in UEFI.

The second axis we propose to characterize bootkits is attack phase. In this section, we summarize and present a bootkit attack model from the perspective of attack phase. On attack phase axis, bootkit attacking can be divided into infection phase and boot phase. Infection phase occurs when a bootkit infects the target system for the first time. And it will not exhibit malicious behavior until the next reboot. Boot phase occurs every time the system reboots after the infection phase. In boot phase, bootkits hijack the control and tamper with the OS loader to perform malicious functions. According to the characteristics of these two phases, we divide a bootkit into four components (i.e., an infector, a hidden file system, a malicious bootloader, and malicious kernel drivers). As illustrated in Figure 4, the infector and the hidden file system are corresponding to the infection phase, while the malicious bootloader and malicious kernel drivers are corresponding to the boot phase. Both phases have their own goals to achieve. In infection phase, bootkits must break through the restriction of modifying the data on the hard disk or SPI flash and hide attack payloads in the victim system. In boot phase, bootkits have to hijack the control earlier than the OS bootloader, bypass the integrity check, and load the malicious code into memory.

3.1 Infection Phase

Infection is the first step for bootkits to control the target system persistently. The infector and the hidden file system will come into play during this phase. The infector will utilize vulnerabilities to write malicious data into the hard disk or SPI flash, while the hidden file system will protect the malicious payload from detection by OS or security software.

3.1.1 Infector

The infector, also called dropper in some articles [8, 42, 9], is an essential component of a bootkit. The infector modifies the data on the hard disk or SPI flash to drop the main function modules.

In legacy BIOS, modifying the bootstrap code is relatively more straightforward. The infector needs the privilege to read and write the hard disk. If the infector does not have the privilege, it needs to utilize the privilege escalation vulnerabilities to do this. The bootkit Olmasco's [44] infector utilized MS10-092,

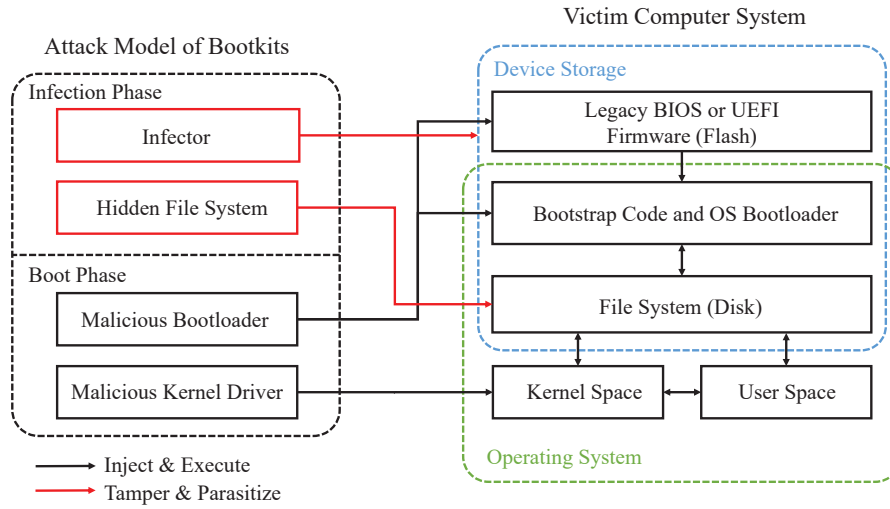


Figure 4. Attack model of bootkits.

Table 1. The main difference between legacy BIOS and UEFI.

	legacy BIOS	UEFI
Development process	Unspecified firmware development process, all BIOS vendors independently support their own codebase	Unified specification for firmware development and Intel reference code (e.g., EDK/EDKII)
Compatibility	Phoenix, AMI, etc. are not compatible with each other	Most are compatible with each other
Implementation	Mostly assembly language	C/C++
Memory model	16-bit real mode	32/64-bit protected mode
Bootstrap code	MBR, VBR, and IPL	None
Attack surface	Mostly on the disk	Mostly in the SPI flash
Partition scheme	MBR partition table	Guid Partition Table (i.e., GPT)
Security mechanism	None	Secure Boot and Boot Guard
Disk I/O	BIOS Interrupts	UEFI services
Bootloaders	Bootmgr and winload.exe	bootmgfe.efi and winload.efi
POST scheme	Test every device every boot	Only test the essential devices for booting
OS interaction	BIOS Interrupts	UEFI services
Boot configuration	CMOS memory	UEFI NVRAM variable

which was the vulnerability utilized by Stuxnet [45] to escalate privilege, and then wrote malicious boot-loader, dbg64, ldr64, and drv64 to the disk. These files were executed sequentially during the boot process.

In UEFI, the infector's goals are more challenging to achieve because reading and writing operations in SPI flash can not be performed directly. It must go

through three steps to modify the firmware. It first needs to dump the local firmware. The infector usually utilizes a particular driver to read and analyze the firmware. LoJax is a typical UEFI bootkit exposed in 2018 whose infector utilized the RwDrv.sys, which was the kernel driver component of a benign software named RWEverything [12]. RwDrv.sys supported

some IOCTL code to perform reading and writing operations to memory-mapped I/O space or a given PCI configuration register. Second, it needs to patch the dumped firmware. LoJax's approach was to replace Ip4Dxe and NtfsDxe with a malicious compressed section that contained a PE image. Third, it has to break through the BIOS write protections. With the signed normal kernel driver RwDrv.sys and a race condition vulnerability proposed by Corey Kallenberg and Rafal Wojtczuk [46], LoJax's infector eventually wrote the patched firmware back to the SPI flash and completed the infection process.

3.1.2 Hidden File System

The hidden file system is another feature of bootkits. Due to the fact that scanning the file system has been a primary function of some security software, storing the malicious driver or PE image in the native OS file system is a dangerous way for bootkits. Many advanced bootkits have their own file systems, which are independent of the native file system. Simultaneously, the bootkit's file system is not visible to OS by means of sophisticated hooks.

TDL3 is the first one to store its configuration files and payloads in a hidden file system. Its successor TDL4 and other bootkits, such as Rovnix, Gapz, etc., also utilized this technique. Figure 5 is the architecture of the storage device driver stack in Microsoft Windows. There are three levels of drivers in the storage device driver stack. The file system drivers are on the top of the stack. The processing of an IRP (i.e., I/O request packet) structure that is addressed to some object located on a storage device. This storage device begins at the file system driver's level. The specific device of the storage object (i.e., the disk partition, which is a contiguous storage area initially reserved for a file system) is determined by the corresponding file system driver, which issues another IRP to a storage class driver's device object that is the middle level of the stack. The storage class driver transfers the I/O request into a corresponding miniport device object. The lowest level of the driver stack is the storage port driver, which provides an interface between a hardware-independent class driver and an HBA-specific (i.e., host-based architecture) miniport driver, such as the SCSI miniport.

In the era when legacy BIOS was the mainstream,

the security software merely cared about the low-level drivers. The kernel-mode hooks were set up at the lowest hardware-independent level in the storage device driver stack by TDL3 sophisticatedly, which made it bypass any monitoring tools or security software at the upper level of the file system or storage class driver. Firstly, TDL3 got a pointer to the miniport driver object of the corresponding device object. Then it created a new malicious driver object to overwrite the DriverObject field in the miniport driver object with the pointer to a newly created field. The malicious driver object had an array named MajorFunction in the DRIVER_OBJECT structure. The IRP major handler pointers in the MajorFunction array can intercept IRP_MJ_INTERNAL_CONTROL and IRP_MJ_DEVICE_CONTROL for the following IOCTL (i.e., Input/Output Control) code in order to monitor and modify read/write requests to the hard drive. So when a read operation is encountered, TDL3 zeros out the return buffer on completion of the I/O operation, and it skips the whole read operation in the event of a write data request. This hooking technique does not touch any of the frequently protected and monitored areas. Many classic hooking techniques, including hooking the SSDT (i.e., System Service Descriptor Table) [47] and the IDT (i.e., Interrupt Descriptor Table) [48] are much more eye-catching by the security software.

The physical space of the hard drive occupied by the hidden file system is designed to grow from the end forward. And all the data is encrypted by XOR with 0x54. More advanced Rovnix [49] uses RC6 to encrypt the hidden file system. A very important point of the design concept of the hidden file system is that the hidden file system can be accessed by Windows APIs like CreatFile, ReadFile, WriteFile, etc. Bootkits reused the upper-level logic of Windows APIs and performed hooking operations at the bottom driver level. This approach could minimize the implementation cost of bootkit developers and maximize the stability of the system due to the fact that the more changes to the OS, the more unstable the system will be.

Table 2 [17] shows the hidden file systems characteristics of four classic bootkits. Except that Rovnix's file system is modified based on FAT16, the other three have their own customized file systems. They all encrypt the contents with different algorithms, and some

Table 2. Hidden file system implementations of four classic bootkits.

Functionality/malware	TDL4	Rovnix	Olmasco	Gapz
File system type	Custom	FAT16 modification	Custom	Custom
Encryption	XOR/RC4	Custom (i.e., XOR+ROL)	RC6 modification	RC4
Compression	✗	✓	✗	✓

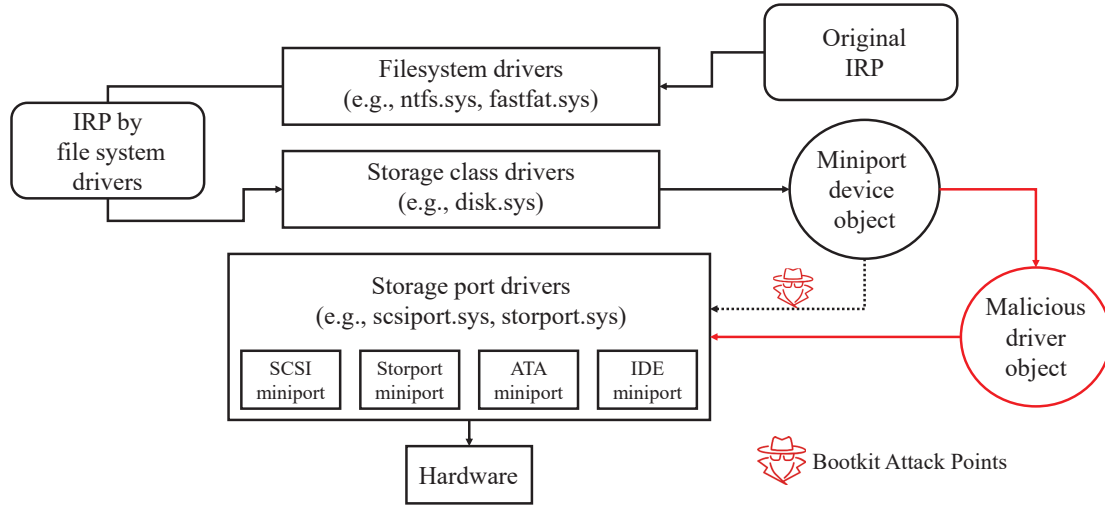


Figure 5. Microsoft Windows storage device driver stack.

are even compressed after encryption. Therefore, it is not easy to analyze the attack load inside the hidden file systems.

However, Rodionov et al. [16] developed a tool for analyzing the hidden file systems named 'HiddenFsReader' to recover the contents of bootkits' hidden storage. It consists of two modules which are a kernel-mode driver and a user-mode application. The kernel-mode driver is responsible for disabling rootkit/bootkit self-defense mechanisms, and the user-mode application provides the user with an interface to access the hard drive in a low level of the driver stack. But it could not apply to most cases. In general, the existence of a hidden file system dramatically increases the difficulty of bootkit detection.

3.2 Boot Phase

Boot phase is the crucial manifestation of bootkits different from other malicious code. In boot phase, the malicious bootloader and the malicious kernel drivers come into play. The malicious bootloader will destroy the integrity of the control flow of OS booting, while

malicious kernel drivers will perform corresponding malicious behavior according to the purpose of bootkit developers.

3.2.1 Malicious Bootloader

After successfully infecting the system, the bootkit will function next boot. The malicious bootloader is the first component executed during the boot process of a bootkit. It has mainly three duties:

- **Hijack the control before the OS bootloader.** Modifying the OS kernel to perform malicious behavior after the OS completely boot is unrealistic due to the fact that the kernel-mode code signing policy will verify the signature of every driver loaded into the kernel space. So the infector must execute earlier to patch the bootloader to disable the corresponding security mechanisms [8, 50].
- **Transfer the control to a certain point to regain the control.** Considering that a malicious bootloader will transfer control to the original bootloader to boot the OS, the bootkit must be able to regain the control after the original bootloader

gets the control. Bootkits usually used the hook method to hook the INT 13h handler to regain control every time when the disk services were used [39, 51, 52].

- **Bypass the security mechanisms and load the malicious module into memory.** This is the critical step for bootkits. Due to the security mechanisms like Kernel-Mode Code Signing Policy, rootkits were not able to function. Booting before the original bootloader gives the bootkit the advantage of disabling security mechanisms in time. The following sections introduce the ways to disable such security mechanisms by their logical loophole. Once the kernel-mode code signing policy is disabled, the malicious bootloader loads malicious kernel drivers into memory, and the duties of the malicious bootloader are accomplished [42–44].

3.2.2 Malicious Kernel Driver

Malicious kernel drivers contain the major functions that the bootkit developers want to implement. The functions of the malicious kernel drivers depend on the goals of the bootkits. They were loaded by either replacing the normal kernel driver or directly loading into kernel space after disabling the integrity check transitorily. We classify the functions of the malicious kernel drivers into the following categories. A bootkit malicious kernel driver may have one or more of these functions.

- **Implement the kernel driver hooks.** Hide the file system of the bootkit, intercept all the read/write requests and protect critical areas from being read or overwritten. The bootkits with the hidden file systems (e.g., TDL4, Gapz, Olmasco, Rovnix, etc.) often have such modules in their kernel drivers.
- **Inject the payloads into processes.** Bootkits often have some payloads that are stored in the hidden file system or downloaded from the C&C server. Festi's way is downloading the malicious function modules from the C&C server to the memory and executing them [53], while Rovnix's way is looking for the signature "JFA" in its hidden file system and injecting the modules which contain the signature into the normal processes [50].

- **Implement the network communication modules.** Some bootkits are designed to create a botnet for attackers. Such bootkits often need to communicate with the C&C servers to download the malicious payloads or get commands to execute [54]. Festi [53] and Gapz [42] implemented the customized TCP/IP stack protocols in kernel mode based on the miniport adapter driver, which made them bypass firewalls and network monitoring tools running on the infected machine. Gapz even encrypted the messages exchanged between the botnet and the C&C server and verified the authenticity of the source of the messages to prevent them from being analyzed [16].

- **Implement the self-defense mechanisms against the security software and analysts.** Firstly, bootkits have to take action to protect their core data structure from being modified by security software. Festi protected its registry keys by hooking the SSDT function ZwEnumerateKey to prevent them from being enumerated by OS [53]. Secondly, since bootkits had been utilized as commercial tools, advanced bootkits developers have considered the confrontation with analysis and debugging tools. The next section will introduce the anti-debugging and anti-emulating techniques of bootkits.

The four components of the bootkit attack model make a bootkit just like a small OS parasitic on the native OS. The "parasitic" OS has its malicious bootloader corresponding to the native bootloader, the hidden file system corresponding to the native file system, and the malicious kernel driver corresponding to the native OS kernel. The red arrows in Figure 4 show that infection phase of a bootkit can be regarded as the process that the infector installs a small OS on the native OS. And boot phase of a bootkit can be regarded as the process that an OS normally boots. As for IoT devices that utilize the cropped Linux kernel or other simpler systems, the bootkit threats they face are more superficial, which means the bootkit attacks for IoT devices are more straightforward.

In the next two sections, we will discuss bootkit attack techniques on the proposed two axes. On boot mode axis, Section IV and Section V respectively introduce the bootkit attack techniques in legacy BIOS and UEFI. On attack phase axis, we discuss the at-

tack techniques in infection phase and boot phase in each section. Techniques in infection phase focus on the methods to gain read/write access to the medium that stores the code that bootkits need to attack. Techniques in boot phase focus on the methods to attack the system boot mechanisms and perform malicious behavior. However, we also discuss the security mechanisms in each boot mode and the co-evolution of bootkits and security mechanisms.

IV. BOOTKIT TECHNIQUES IN LEGACY BIOS

In this section, we will illustrate some attack techniques in our attack model (i.e., A_1 - A_4), as well as the security mechanisms (i.e., D_1 - D_3) applied in the boot process and their implementation logic flaws in legacy BIOS.

4.1 Infection Phase in Legacy BIOS

Infection phase for bootkits in legacy BIOS is much easier than it is in UEFI. All the targets the infector needs to attack are on the disk. So bootkits infectors just need to break through physical disk protections and gain the read/write access to a specific physical location on the disk. Then they can easily tamper with the bootstrap code and write malicious function modules into the disk.

4.1.1 Break Through Physical Disk Protections (A_1)

In most systems, reading or writing in insensitive partitions in the file system does not need high privilege (e.g., administrator privilege in Windows, root privilege in Android, and Linux). But reading or writing to a specific address on the hard disk always requests higher privileges. For bootkits in legacy BIOS, infection is based on the ability to modify the disk by physical address. Therefore, LPE (i.e., local privilege escalation) has become a necessary part of bootkits in infection phase.

Many vulnerabilities have been utilized to achieve LPE in the past few years. Olmasco, TDL4, and Stuxnet have been confirmed to utilize MS10-092 to escalate privileges [44, 8, 45], while Gapz leveraged CVE-2010-4398 [55] and CVE-2011-3402 [56] for LPE. Huang et al. showed that the essence of LPE

in Windows is actually the abnormal interactions between high-privilege processes and user-controllable files [57], and the situations in other operating systems (e.g., Linux and Android) are also similar [58, 59]. Kujanpää et al. summarized this interactions model and utilized reinforcement learning to automatically escalate privilege in an emulated Windows 7 environment [60]. However, Scott et al. conducted a survey about techniques to prevent LPE, but they concluded that no single technique was found to effectively mitigate all known and potential attack vectors with reasonable performance cost overhead. In general, it is feasible to achieve LPE under the condition of exploitable vulnerabilities and interactions model between high-privilege processes and user-controllable files. So we think the infection phase is an indispensable but not the most valuable part of bootkit attacks in legacy BIOS. The most valuable part of bootkits in legacy BIOS is the techniques in boot phase.

4.2 Boot Phase in Legacy BIOS

In legacy BIOS, the primary duties of bootkits for boot phase were to gain the control before OS bootloader and maintain the control until malicious function modules were executed. Next, we will discuss the techniques bootkits used to achieve this goal.

4.2.1 Tamper with Bootstrap Code (A_2)

One of the malicious bootloader's duties is hijacking the control before the OS bootloader. In legacy BIOS, it is primarily achieved by tampering with bootstrap code. In general, the attacks to bootstrap code can be classified into the following four categories. We will illustrate their strengths and weaknesses in Figure 6.

- **Replace the whole MBR [8].** MBR is the first code executed on the hard disk. TDL4 backed up the original MBR and replaced the first sector with its own code. The bootkit code that occupied the first sector of the disk would jump to the bootkit partition, which has been created during the infection process. After finishing the job of patching the OS bootloader, the original MBR would be recovered, and the normal boot process would carry on. As illustrated in Figure 6a, this approach is the simplest and most direct. However, replacing the whole MBR is too obvious for security software to ignore. The bootkit code in

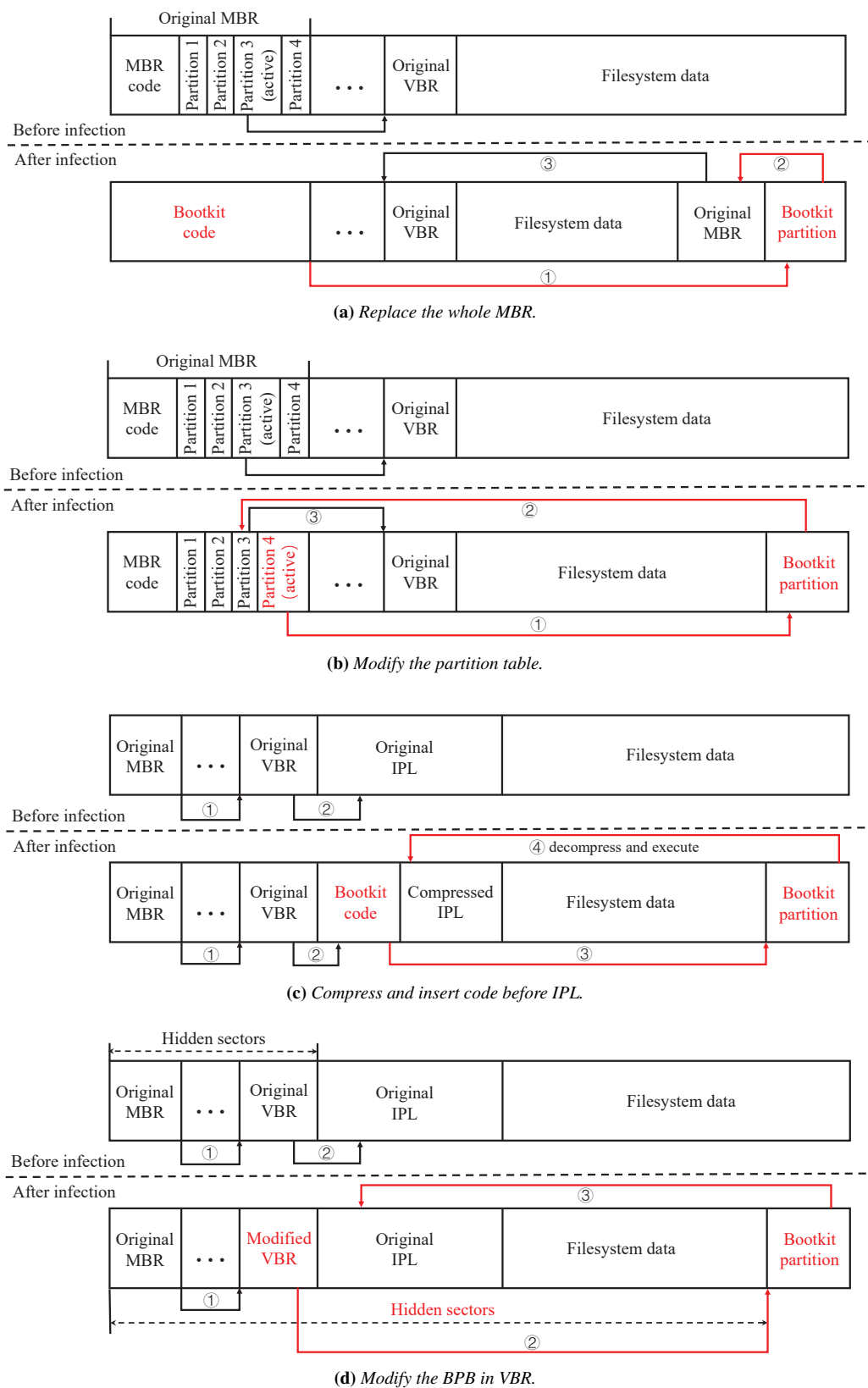


Figure 6. Four categories of methods to hijack the control before the bootloader. The black arrows represent normal control flow transfers, while the red arrows represent control flow transfers that were hijacked after bootkit attacks.

the first sector does not have the MBR code or partition table entry. If security software notice the MBR is obviously abnormal and recovers the MBR, the bootkit will fail to execute.

- **Modify the partition table [44].** Under normal conditions, MBR has only one partition marked as active. We assume the active partition is partition 3 in Figure 6b. Rather than replacing the whole MBR, Olmasco only modified the partition table. It selected an unoccupied partition and marked it active instead (i.e., partition 4). The selected partition in the partition table pointed to the bootkit partition, which made the bootkit hijack the control. After that, the bootkit would mark the original active partition (i.e., partition 3) as active and make the boot process carry on. This approach is more concealed than the previous one due to the fact that the partition table of each computer is different. So the security software can not determine whether the MBR has been attacked directly based on the value of the partition table. However, the weakness is also obvious. MBR partition table only has four entries. If the entries have been used up, this approach will fail.
- **Compress and insert code before IPL [50, 43].** Since the MBR has been focused on by security software, modifying any part of MBR was obvious for bootkits. Thus IPL became the attack target. As illustrated in Figure 6c, Rovnix modified the IPL by compressing it with the aPlib compression library [61] and putting the compressed IPL at the end of the place that the IPL occupied originally. The bootkit code was placed in front of the compressed IPL, and all 15 sectors would be encrypted (the original IPL occupied 15 sectors). Thus VBR would transfer the control to the bootkit code, and the bootkit would decompress the IPL and execute it to boot normally at last. This approach intercepts the control without touching MBR, but the decompression of the IPL must be accomplished in the 16-bit real mode, which is time-consuming and needs more code to implement.
- **Modify the BPB in VBR [42].** The ideal approach to hijack the control is to modify the data in the bootstrap code as little as possible. Gapz modified the variable “hidden sectors” in BPB,

which is a structure in VBR. The value of hidden sectors indicates the number of sectors before IPL on the disk. VBR code locates the IPL and transfers the control according to this value. As illustrated in Figure 6d, Gapz changed the hidden sectors to a relatively large value which was calculated to just offset to the bootkit partition. After that, the bootkit transfers the control back to the original IPL to boot normally. This approach is relatively hard to detect by scanning the disk. However, with the strengthening of security software’s awareness of bootstrap code backup and scanning, it has become increasingly difficult to modify the data on the disk to hijack the control flow.

4.2.2 Hook BIOS Interrupt Handlers (A_3)

In legacy BIOS, before OS has totally booted, the BIOS interrupts implemented the interaction with the system resource (e.g., disk I/O, debug registers). Nevertheless, the interrupt handlers are executed in memory and could be hooked. The most common hook approaches for bootkits are hooking the INT 13h, INT 19h, and INT 1h.

INT 13h is the direct disk service supported by legacy BIOS. Before the OS disk driver is loaded into the memory, any access to the disk must rely on INT 13h. The INT 13h functions from the bootstrap code to the OS bootloader. Even bootmgr and winload.exe still use the INT 13h to read and write the disk. There were many bootkits in legacy BIOS whose analysis reports [39, 43, 44, 42, 35, 8] have shown that they utilized the hook at INT 13h handler to regain the control every time the disk service is called. At the same time, hooking INT 13h can make the bootkit monitor every read and write operation to the disk. The hook process often happens when the bootkits firstly hijack the control in a malicious bootloader that locates the IVT (i.e., Interrupt Vector Table) entry of the INT 13h handler and inserts a “JMP” instruction ahead of the handler code [10].

INT 19h is the Bootstrap Loader Service supported by legacy BIOS. This interrupt attempts to load the sector at head 0, track 0, and sector one on the first diskette into memory at 0:7C00h. Bootkits can hook INT 19h handler and recall INT 19h to hijack the control before the original MBR [7]. However, INT 19h

would only be called once in the boot process, so this method is often used with hooking INT 13h.

The INT 1h handler is called when debugging event occurs. The dr0 through dr7 registers are used to trace and set hardware breakpoints. Rovnix utilized a sophisticated way to implement hooks by these debugging registers [50]. Rovnix only needs to hook the INT 1h handler, write the address of the hooking target to the dr0 through dr4 registers, and set the bitmask in dr7 to enable the hardware breakpoints. When the target address receives the control, it will be stuck in debugging event, and the hooked INT 1h handler will get the control. This approach achieves hooking any address with only a few code changes.

4.2.3 Obfuscation and Anti-Dynamic-Analysis Techniques (A_4)

As a category of advanced malware, some bootkits were even sold on a private underground forum [17]. Thus the developers have adopted some techniques to prevent security researchers from analyzing the bootkits statically and dynamically. Like context-aware malware, some bootkits have utilized obfuscation, anti-VM, and anti-debugging techniques. Branco et al. [62] summarized the techniques of anti-dynamic analysis that the malware often used. However, some bootkits utilized other clever ways to make analysts harder to analyze them on this basis.

As shown in Section 4.2.1, Rovnix needs to decrypt the whole IPL to execute the malicious code. The decryptor Rovnix adopted is polymorphic [50]. Rovnix developers split the decryptor into many basic blocks randomly, upset their order, and connected them with “JMP” instruction. This obfuscation approach makes the decrypting processes different in each instance of Rovnix.

WMI (i.e., Windows Management Instrumentation) could be used to detect the virtual machine environment [63]. Olmasco utilized the WMI IWBemServices to get the manufacturer names of the BIOS, disk, controller, etc., to determine if it was in a virtual machine. On the other hand, Olmasco could identify the company of the infected machine by the system name and domain name and deploy a customized payload that explicitly targets that company [44].

Anti-debugging is also a feature of some bootkits. A common and straightforward approach to detect the

debugging environment is checking the value of Kd-DebuggerEnabled by some Windows APIs [64]. Festi utilized a more violent method [53]: clearing the debug registers dr0 through dr4 periodically, which made the hardware breakpoints invalid.

4.3 Evaluation for Related Security Mechanisms in Legacy BIOS

4.3.1 Early Launch Anti-Malware (D_1)

The ELAM (i.e., Early Launch Anti-Malware) module was a kernel driver designed to allow the third-party security software’s drivers to execute before any other third-party drivers. It has an ELAM database of known benign and known malicious driver signatures [65]. ELAM verifies the hash and certificate of each driver image to be loaded. So it could prevent attackers from loading a malicious kernel driver after the OS kernel was initialized. However, the implementation of the ELAM was not perfect. The default policy of ELAM is PNP_INITIALIZE_BAD_CRITICAL_DRIVERS, which means loading known benign, unknown, and known bad but critical drivers. It is a relatively loose policy because ELAM must ensure the system can boot successfully firstly. Also, the design of ELAM determines that it could not prevent the bootkits from infecting the system due to the time that ELAM works being too late. A bootkit attack often happens before the OS bootloader, but ELAM works after the OS kernel is initialized as a kernel driver. ELAM could work against rootkits relatively effectively, but it could not stop bootkits, which is the most significant difference between bootkits and rootkits.

4.3.2 Microsoft Kernel-Mode Code Signing Policy (D_2)

Microsoft kernel-mode code signing policy requires that each driver be verified before it is loaded into the kernel. In the old version of Windows in 32-bit mode, it is optional. However, in the modern 64-bit Windows version, it is mandatory. The reason why bootkits need to be executed before OS is to bypass or disable the security mechanisms like this. If bootkits execute after OS booted, the kernel-mode code signing policy will block this malicious execution. Moreover, indeed, the kernel-mode code signing policy has some weaknesses for bootkits to bypass [17].

- In Microsoft Windows 7 and Vista, whether the integrity checks were enforced depended on a variable in kernel space named `nt!g_CiEnabled`. Turla utilized the vulnerability of `VBoxDrv.sys` to accurately clear the value of the `nt!g_CiEnabled` in the kernel space [66], thus the malicious kernel drivers could be loaded into memory later.
- If OS boots into WinPE (i.e., Windows preinstallation) mode, the `nt!g_CiEnabled` would be set to false, and the integrity checks would be disabled briefly. TDL4 utilized this characteristic to load its malicious kernel driver in WinPE mode [8]. TDL4 searched the value 16000020 and modified it to 26000022, which meant `BcdOSLoaderBoolean.WinPEMode`. When bootmgr read the BCD, the OS booted into WinPE mode and loaded the TDL4 malicious driver.
- If OS did not boot into WinPE mode, there was another and last chance to disable the integrity checks: the TESTSIGNING option, which was intended to be used for debugging the developing benign drivers. If it is set to TRUE, a driver could be loaded into the kernel with an invalid signature. Necurs utilized the TESTSIGNING to load the malicious kernel driver with a custom certificate [67].

Since Windows 8, the integrity checks did not rely on a single variable anymore but utilized more fields for additional callback functions to verify the signatures of the PE images. However, for bootkit developers, they can still achieve the same goal in a similar way, but with more reverse engineering effort.

4.3.3 Virtual Secure Mode (D_3)

VSM (i.e., Virtual Secure Mode) is a relatively new technique based on Microsoft's Hyper-V in Windows 10 [68]. VSM makes the OS and critical system modules execute in many isolated containers protected by Hyper-V. Access to these isolated regions is controlled and granted solely through the hypervisor. The containers in VSM are kind of similar to the enclave in Intel SGX [69]. When the VSM is in place, the containers will still be secure under the condition of the kernel having been compromised since virtual containers are separated from each other. To the best of our knowledge, no bootkit has succeeded in compromising the Windows VSM that has been spotted in the

wild ever. However, the performance overhead and resource consumption of VSM is large, which causes it to be unused in most cases.

V. BOOTKIT TECHNIQUES IN UEFI

In this section, we also introduce the attack (i.e., A_5 - A_{12}) and defense techniques (i.e., D_4 - D_6) in UEFI from infection phase and boot phase. Different from the focus of techniques in legacy BIOS, the most valuable part for bootkits in UEFI is the techniques in infection phase.

5.1 Infection Phase in UEFI

Section 4.1 has shown that the key for bootkits to infect in legacy BIOS was high-privileged access to hard disk physical addresses. Bootkits resided in areas on the disk that were not aware by the OS, which made them survive after reboots or even system reinstall. Formatting the hard disk can completely remove such bootkits. But if the UEFI firmware stored in the SPI flash is infected with bootkits, it is much harder to remove them. Infection phase in UEFI mainly refers to the phase that bootkits try to modify binaries in firmware or tamper with the execution process of firmware in memory. We will discuss some techniques used by bootkits to achieve this goal.

5.1.1 Tamper with ESP (A_5)

ESP (i.e., EFI System Partition) is a segment of storage space on the disk. Usually, an ESP contains the bootloaders (e.g., `bootmgfw.efi` and `bootx64.efi`) or kernel images (e.g., `kdstub.dll`) for all installed operating systems and configuration or log files (e.g., BCD and BCD.log). At the end of the UEFI firmware code execution (i.e., the BDS stage), the control is transferred to the bootloader of the chosen OS located in ESP.

As the connection between UEFI and OS on the disk, ESP has become a critical attack vector for bootkits. FinSpy [2] replaces the `bootmgfw.efi` with a malicious bootloader in ESP and encrypts its malicious kernel drivers with RC4. The decryption key is the unique GUID of the EFI partition, so it is different in every infected machine. When the UEFI code transfers the control to `bootmgfw.efi`, FinSpy hijacks the execution and hooks the kernel functions

to drop the attack payloads. Similarly, ESPECTER [1] adds new PE sections and modifies the entrypoint of bootmgfw.efi and bootx64.efi to refer to the malicious code. The modification of both FinSpy and ESPECTER will patch the OS bootloader (i.e., winload.efi). The patched OS bootloader will further patch the OS kernel (i.e., ntoskrnl.exe) to perform malicious behaviors. In UEFI, attacking ESP is a relatively easier way to tamper with the boot process than attacking firmware in SPI flash. But this method only works when Secure Boot is turned off. The modified bootloader in ESP can not pass the integrity check of the previous stage.

5.1.2 Break Through UEFI Firmware Protections (A_6)

The goal of breaking through UEFI firmware protections for bootkits is similar to the goal of breaking through physical disk protections in Section 4.1.1. Most manufacturers have implemented some protection schemes [70–72] to protect the firmware from being tampered with. Nevertheless, some of them had weaknesses in logic. Attacks to firmware protections can be classified into two categories, which aim at the BIOS control bits protection and the update process of the firmware.

Attacking the BIOS Control Bit Protection. The BIOS.CNTL register is designed to protect the UEFI BIOS from unauthorized modification. But its implementation had logically flawed. The BIOS.CNTL register has three critical bits (i.e., BIOSWE, BLE, and SMM_BWP) involved with writing operations to the firmware. The firmware is allowed to be written when BIOSWE is set to 1. BLE is used to ensure BIOSWE is 0, which prevents BIOSWE from an accidental flip. Kallenberg et al. found that the implementation of BIOS.CNTL had a race condition vulnerability [46]. When the SMM tries to set BIOSWE to 1, the BIOSWE is set to 1 shortly indeed under the condition of proper configuration of BLE. And BLE will soonly make BIOSWE set to 0 during a periodic check, which causes a short time for attackers to utilize. In this way, with enough attempts, an attacker can write arbitrary malicious code into the firmware and stay there for a long time. LoJax used this vulnerability to create two threads; one repeatedly tried to set BIOSWE to 1, and the other tried to write mali-

cious code into the SPI flash [12].

Attacking the Update Process of the Firmware.

The first attack to the BIOS update process was presented by Wojtczuk et al. [73]. After that, the update process of the firmware has been a critical attack surface for attackers. According to the UEFI specification, only signed UEFI capsule-based updates are allowed to be written to the firmware. Bashun et al. summarized the attack vectors of the update of the UEFI variables and signature database in firmware, and they found many update processes did not verify the signatures of the binaries [74]. Matrosov et al. found that the DerStarke could use the PeiLoader (i.e., PL.efi) to hook the firmware update process on the fly and inject the implant named DxeInjector (i.e., DI.efi) to the modified firmware binaries [75]. In Summary, the main reason for the update process being attacked is that the user-mode update program did not perform signature verification on the binaries to be updated, or the signature verification mechanism itself has vulnerabilities.

5.1.3 Compromise UEFI Secure Boot (A_7)

Secure Boot is a critical security characteristic of UEFI. It establishes a chain of trust from the RoT during the boot process. Similar to ELAM, there are two critical databases in Secure Boot architecture named db and dbx. The db records a list of trusted public keys certificates authorized to authenticate signatures, and the dbx records a list of certificates of public keys and hashes of UEFI executables that are prohibited from executing at boot time. Matrosov et al. proposed two approaches to attack Secure Boot based on the condition that protections of SPI flash have already been broken [17].

- **Patch PI firmware to disable Secure Boot [76].**

Before the Verified Boot and Measured Boot [77] were adopted, Secure Boot relied on PI firmware (i.e., SEC, PEI, and DXE dispatcher) and PK (i.e., Public Key) stored in SPI flash as RoT. Since EDK II has been widely used among firmware vendors, it was not a secret that the source code of the routine DxeImageVerificationHandler for verifying the signatures was located in SecurityPkg/Library/DxeImageVerificationLib folder in EDK II. Attackers could patch that routine to re-

turn `EFI_SUCCESS` every time. Thus they could load and execute the malicious code as they want.

- **Modify the UEFI variables to bypass security checks [78, 79].** Most implementations of Secure Boot are stored in UEFI NVRAM variables. Attackers can add the hash of the malicious modules to db or remove it from the dbx. They can even replace the PK with their own certificates and control the RoT permanently.

However, the two attack approaches were ineffective since Intel introduced the Verified Boot and Measured Boot. But there were still many machines vulnerable to these attacks due to whose manufacturers did not configure or implement the security mechanisms properly [21, 75, 80, 81].

5.1.4 Pollute the Supply Chain (A_8)

SUNBURST in SolarWinds event [82] has shown the high destructiveness and stealth of supply chain attack. In fact, supply chain attacks have appeared in legacy BIOS [83, 84]. But from a bootkit developer's perspective, supply chain attacks make more sense for breaking through security mechanisms in UEFI. For bootkits in UEFI, the purpose of infection phase is to break through the protections of firmware and modify or insert their malicious function modules into the victim systems. Polluting the supply chain of firmware vendors is a relatively convenient method for bootkit developers because it lightens the workload for utilizing other post-exploitation techniques to complete the infection phase.

Supply chain attack in firmware often happens during the firmware development process. Attackers can introduce misconfigurations to the security mechanisms (e.g., the policy of Secure Boot and the flag of manufacturing mode for FPF) implemented by OEM hardware vendors. It could be performed through physical contact or remotely. Attackers can get physical access to the personal computers of firmware or hardware developers when they hand over laptops for a security check or leave them in the hotel. On the other hand, Vu et al. have shown that attackers could also infiltrate into the internal network of developers and add malicious code to the source code repository or build server [85]. These scenarios could happen in APT (i.e., Advanced Persistent Threat) attacks [86] and would cause large-scale damage once successful.

Ibdah et al. demonstrated an attack that leveraged tampered firmware and system management cycles to covertly collect data from the application layer [87], which showed the huge threat of untrusted firmware to upper-layer system applications. Attacking the supply chain makes bootkits infection phase easier; in turn some bootkits have been confirmed to be used for supply chain attacks. MoonBounce was considered a bootkit used for supply chain attack [3]. The sample found was only three functions hooked away from the original firmware. It utilized open-source components, targeted the MSI E7846IMS.M30, and was released on the same day as the MSI firmware patch [88]. Therefore, we believe that supply chain attack will be an increasingly important technique for bootkit developers, especially APT groups.

5.1.5 Undermine Intel Boot Guard (A_9)

Section 5.2.1 shows that firmware binaries in the SPI flash could not be trusted due to the fact that many SMM attacks could compromise them. Intel introduced the Boot Guard to use signed ACM (i.e., Authenticated Code Module) to verify the UEFI BIOS binaries before executing them. The RoT was also moved from the DXE to the hardware inside the Intel microarchitecture. The hash of an OEM public key is locked within the FPF (i.e., Field Programmable Fuse), which can be programmed only one time by the end of the manufacturing process. The ACM code is parsed and verified by the CPU's microcode which will not be tampered with by attackers. Then verified ACM will verify the signature of UEFI BIOS binaries, which makes the boot process can be trusted even if the firmware protections have been broken.

But in implementation, Matrosov et al. found that some vendors have not written their hashes in the FPF, or they did so but did not subsequently disable the manufacturing mode that still allows a write operation for attackers to write FPF keys of their own and then lock the system, which makes them control the RoT forever [80]. At the same time, the FPF could also be modified by Intel ME as its memory regions when the ME is still in the manufacturing mode, and ME in that mode can be accessed from the OS for both reads and writes in turn [17]. On the other hand, some CPU debugging interfaces were not disabled before they left the factory (e.g., Intel DCI), which made at-

tackers destroy as they wished with physical access to the CPU[81].

5.1.6 Utilize Intel Management Engine (A_{10})

Intel ME (i.e., Management Engine) is a special and critical part for both firmware vendors and attackers. ME has a separated x86-based CPU and an embedded real-time operating system on a separate 32-bit microcontroller which is totally independent of the main CPU [89]. It is an ideal attack vector for bootkits developers to insert their malicious code to attack the RoT due to the power of ME to execute in SMM. ME code is executed on its own chip, but it needs to communicate with other components on the board, which would be an essential attack surface for bootkits. ME uses HECI (i.e., Host-Embedded Controller Interface) to communicate with the OS kernel. If ME does not verify the input from OS, it could be compromised when the attacker has taken over the OS kernel.

Goryachy et al. found vulnerabilities to execute arbitrary code in ME's OS [90–92]. And they showed that the Bootkit could attack the RoT to bypass or disable the security mechanisms (e.g., Intel Boot Guard) [29].

Intel AMT (i.e., Active Management Technology) platform is implemented as an application in ME's OS [27]. AMT can communicate with another system over the network in ME environment, even if the central system has not booted. It can also access the memory independently of the main CPU. Tereshkin et al. first focused on AMT security and proposed ways to inject code into AMT to make ME perform bootkit functions [30]. Kovah et al. found a channel named SOL (i.e., Serial-over-LAN) in AMT [26], and Microsoft found the bootkit instance, which utilized the SOL to bypass the OS-level firewall to communicate with the C&C servers [93]. Nowadays, chips on the board independently of the main CPU are common, such as Intel ME, EC (i.e., Embedded Controller), and BMC (i.e., Baseboard Management Controller), in which some attacks have been implemented [94, 25]. These chips are also the ideal attack objects for bootkits at the same time. Although they are relatively safe inside, the boundary interfaces of them are vulnerable to attackers.

5.2 Boot Phase in UEFI

Different from infection phase, boot phase in UEFI and boot phase in legacy BIOS have a lot in common. Because even though the boot process of the BIOS and the storage location of the firmware have changed, the process of loading the OS kernel has not changed much. The methods that malicious bootloaders used to bypass security checks and load the malicious kernel drivers in legacy BIOS were confirmed that they were still utilized by many recently appeared bootkits in UEFI [1, 3, 2, 95]. However, there are some new techniques worth noting that appeared in UEFI, and we will supplement these techniques below.

5.2.1 Manipulate SMM Modules (A_{11})

SMM (i.e., System Management Mode) is the highest privileged execution mode in x86 processors, and it implements platform-specific management functions independently of the OS. OS software can raise an SMI (i.e., System Management Interrupt) to enter the SMM, which is also called Ring -2. The high privilege means that attacks in this mode will cause great harm to the system. Yao et al. [72] classified the attacks to SMM into five categories: unlocking SMRAM [96], Cache Poisoning [97], SMRAM remap [98], branch outside of SMRAM [99] and SMM communication attack [71]. For bootkits, the main purpose of attacking SMM is that SMM is closely related to the update process of SPI flash. So we focus on the last two categories of attacks.

The communication between the SMRAM and other physical memory has to obey strict verification. However, the C language used to implement the UEFI does not help track the regions to which a pointer has ever pointed. For instance, the `ValidateBufferIsOutsideSmram` is used to judge whether a pointer points to a memory buffer outside the SMRAM range. But it can not accurately detect the situation that the pointer to be judged points to a structure that has a field as a pointer to another buffer outside SMRAM. A vulnerability numbered CVE-2021-26943 on ASUS laptops [100] showed that an attacker could repeatedly trigger an error code (i.e., 0x7) to construct a physical address 0x07070707 in SMRAM, and then put the malicious code in address 0x07070707. SMM would consume the data outside the SMRAM, thus causing arbitrary code execution in SMM. The privilege escalation from

root (i.e., ring 0) to SMM (i.e., ring -2) has become one of the most critical attack surfaces for bootkits in SMM. ThinkPwn [101] was a typical bootkit that manipulated the data consumed by SMI handlers to escalate privileges in a similar way. Yin et al. [102] proposed a protocol-centric static analysis method and found 36 SMM privilege escalation vulnerabilities. It means that vulnerabilities in SMM remain an extensive and severe issue for bootkit attacks.

5.2.2 Replace the S3 Boot Script (A_{12})

The S3 resume process has been a vital attack surface since Wojtczuk et al. showed the vulnerability of the S3 boot script (i.e., CVE-2014-8274) in 2014 [24]. S3 boot script is a series of opcodes interpreted by the boot script executor firmware module at the end of PEI to be used by BIOS to wake from sleep mode [103]. The platform can skip the DXE to reduce the time to wake from the S3 sleep state by the S3 boot script [22]. Since the S3 boot script is stored in memory and can be executed before many security mechanisms are activated, locating and replacing the memory of the S3 boot script is feasible for bootkits. One approach proposed by Wojtczuk et al. to execute arbitrary code in an S3 boot script is as follows [23]:

- Get the S3 boot script pointer from the UEFI variable `AcpiGlobalVariable` and copy the original UEFI boot script table to restore the original state after the attack.
- Use the modification dispatch code `EFI_BOOT_SCRIPT_DISPATCH_OPCODE` to add the record into the UEFI boot script table as the first boot script opcode command, which points to the malicious code to execute.
- Wait or trigger the S3 sleep mode to execute the malicious boot script by other malicious code in the OS.

The `MdeModulePkg/Library/PiDxeS3BootScriptLib/`, which is also open-source in EDK II shows the implementation of the S3 resume. That makes the mechanism clearer to bootkit developers. Intel has introduced a mechanism named LockBox to protect the S3 boot script from any modifications outside of SMM [103]. However, SMM itself also has many vulnerabilities for attackers to utilize, which makes the S3 boot script still vulnerable.

5.3 Evaluation for Related Security Mechanisms in UEFI

5.3.1 UEFI Firmware Protections (D_4)

The UEFI firmware protection is the basic security guarantee against UEFI bootkits for PCs. Unlike infection phase in legacy BIOS which has few specialized security mechanisms to protect physical disks from tampering, UEFI firmware is protected by special registers and security mechanisms. Moreover, breaking through UEFI firmware protections is the first and inevitable step in a successful bootkits attack. In Section 5.1.2, we introduced two ways to break through UEFI firmware protections. However, attack and defense techniques are always in a process of co-evolution. After the disclosure of Lojax, security mechanisms against this type of race condition vulnerability were also proposed. To prevent BIOS control bit protection attack, the firmware manufacturers only need to configure the `SMM_BWP` properly, which ensures that the BIOS region is writable only if all the cores are running in SMM (i.e., System Management Mode) and `BIOSWE` is set to 1. However, Matrosov et al. [75] have shown that many manufacturers did not configure the `BIOS_CNTL` correctly due to the consideration of convenience.

As for attacks to the process of firmware update, Intel proposed BIOS Guard, which is a BIOS flash update hardening technology [104]. BIOS Guard changes the situation that SMM controls the MSR (i.e., Model-Specific Register) and the flash updates. Instead, it creates a small trust boundary for the firmware update process in the flash. BIOS Guard will replace SMI handlers to control MSR for read and write operations on SPI flash by an Intel-signed and hardware-verified BIOS Guard Authenticated Code Module. Simultaneously, update authentication is also performed by BIOS Guard module. So Intel BIOS Guard greatly reduces the attack surface for firmware update process attacks and provides a much more defensible environment from which to perform flash operations. However, again, Matrosov et al. [80] have shown that many OEM vendors did not enable this technology due to performance or cost issues. It means that bootkits still have the potential to break through firmware protections on many older machines.

5.3.2 UEFI Secure Boot (D_5)

There is another difference in infection phase between legacy BIOS and UEFI, which is that even bootkit has successfully broken through the firmware protections and injected malicious code into UEFI firmware, it still could not be executed directly because of UEFI Secure Boot. In order to build a trusted Boot chain during UEFI boot process, Secure Boot first chooses a security anchor as RoT (i.e., PI firmware, in most cases), and then ensures that each UEFI stage is validated by the previous stage before it is executed [33]. So if bootkits want to succeed in gaining control at UEFI firmware level, they have to bypass Secure Boot integrity check besides successfully writing data to the SPI flash. As mentioned in Section 5.1.3, the security of UEFI Secure Boot largely depends on the integrity of its key data structures (i.e., db, dbx and RoT). Both attacking the key data structures of Secure Boot and patching routines are used to verify signatures can bypass Secure Boot. Therefore, relying on Secure Boot alone can not protect UEFI firmware from bootkit attacks. On the other hand, Sanwald et al. have shown that more than 50% of the OEMs specify not to enable Secure Boot, disable certain functionality, or cryptographic secrets, and around 30% of the OEMs do not explicitly define a countermeasure in case the authentication fails [34]. This phenomenon also facilitates bootkit attacks.

5.3.3 Intel Boot Guard (D_6)

Section 5.3.2 has shown the original RoT of Secure Boot is not secure enough. In most cases, hardware is more reliable than software. Intel proposed Boot Guard to shift the RoT to dedicated hardware to ensure the integrity of UEFI Secure Boot. Boot Guard utilizes the CPU microcode and signed ACM to perform Verified Boot and Measured Boot. The former checks if the PI firmware has been altered or modified. The latter computes cryptographic hashes of key data structures and stores them in TPM PCRs (i.e., Trusted Platform Module Platform Configuration Registers) [77]. Boot Guard creates a more secure chain of trust. The RoT in dedicated hardware ensures the integrity of Secure Boot. Meanwhile, Secure Boot ensures the integrity of UEFI firmware. In theory, this set of security mechanisms can effectively resist bootkit attacks if fully implemented. However, Section 5.1.5 has shown

that there are still many flaws in the OEM vendor's Boot Guard implementation, which leaves Boot Guard still vulnerable to bootkit attacks.

VI. SUMMARY AND FUTURE FOR BOOTKITS

6.1 Summary of the Co-Evolution of Bootkits and Security Mechanisms

Table 3 shows the classification and evolution of classic bootkits according to the time they were exposed. Some of them are not “standard” bootkits, but the techniques they utilized and the intent their behaviors showed are similar to typical bootkits. We also illustrate all the attack and defense techniques discussed in Section IV and Section V by two axes in Figure 1. From the perspective of bootkit attack techniques, the attack surface has been getting wider on boot mode axis, and the attack position has been getting deeper in the opposite direction of the attack phase axis. In legacy BIOS, techniques in infection phase were relatively single. Many sophisticated bootkits developers focused on the techniques in boot phase to bypass the OS-level security mechanisms. In UEFI, many techniques in boot phase in legacy BIOS were still adopted. However, changes in boot process and enhancements to firmware security have forced bootkits developers to concentrate on techniques in infection phase. In general, the transition from legacy BIOS to UEFI not only brings about an overall improvement of PC, including security, but also brings a larger attack surface to bootkits (e.g., ESP, Intel ME, Secure Boot, and Boot Guard).

As for the perspective of security mechanisms, from the ELAM against rootkits to the kernel-mode code signing policy against legacy BIOS bootkits, the security mechanisms are also evolving at the same time. With the prevalence of UEFI, the bootkits have turned their attention to UEFI firmware attacks since 2013. The firmware and hardware manufacturers also introduced some techniques to protect the boot process (e.g., Intel Secure Boot and Boot Guard). On the other hand, the attack surface is wider but attacking is harder since manufacturers have introduced more and more complicated security mechanisms to protect firmware. The RoT in Secure Boot was in DXE when it was first proposed by Intel. It was unsafe be-

Table 3. *Summary of bootkits and samples with bootkits techniques.*

Name	Time	Type	Attack vector	Discovered way
WinCIH [105]	1998	Legacy BIOS	Attack BIOS from OS	In the wild
eEye BootRoot [10]	2005	Legacy BIOS	Attack MBR	Proof of concept
APCI rootkit [106]	2006	Legacy BIOS	Attack ACPI firmware	Proof of concept
Mebroot [52]	2007	Legacy BIOS	Attack MBR	In the wild
Vbootkit [51]	2007	Legacy BIOS	Abuse Windows Vista	Proof of concept
PCI OptRom rootkit [75]	2007	Legacy BIOS	Attack Option ROM for PCI	Proof of concept
IceLord [107]	2007	Legacy BIOS	Attack BIOS from OS	Proof of concept
SMM rootkit [19]	2007	Legacy BIOS	Attack SMM	Proof of concept
Mebratix [108]	2008	Legacy BIOS	Attack MBR	In the wild
SMM rootkit 2 [20]	2008	Legacy BIOS	Control USB devices by SMM	Proof of concept
Vbootkit2 x64 [40]	2009	Legacy BIOS	Bypass the digital signature checks on Windows 7	Proof of concept
Stoned [39]	2009	Legacy BIOS	Attack MBR	Proof of concept
BIOS patching [109]	2009	Legacy BIOS	Attack BIOS from OS	Proof of concept
Computrace [36]	2009	Legacy BIOS	Modify BIOS for business purposes	Commercial software
TDL4 [8]	2010	Legacy BIOS	Attack MBR	In the wild
Stoned x64 [16]	2011	Legacy BIOS	Attack MBR	Proof of concept
Olmasco [44]	2011	Legacy BIOS	Modify MBR partition table	In the wild
Evil Core [110]	2011	Legacy BIOS	Use SMP to boot into protected mode	Proof of concept
Rovnix [50]	2011	Legacy BIOS	Compress and tamper with the IPL	In the wild
Mebromi [111]	2011	Legacy BIOS	Attack BIOS similarly to IceLord	In the wild
VGA [112]	2012	Legacy BIOS	Install itself in the VGA firmware	Proof of concept
Gapz [42]	2012	Legacy BIOS	Attack VBR	In the wild
Rakshasa [113]	2012	Legacy BIOS	Attack BIOS	Proof of concept
Kon-Boot [37]	2012	UEFI	Bypass Secure Boot and Windows password authentication	Commercial software
DreamBoot [13]	2013	UEFI	Hook bootloaders in UEFI	Proof of concept
BadBIOS [114]	2013	Legacy BIOS	Attack BIOS	In the wild
X86 Memory Bootkit [115]	2013	UEFI	Exploit all available memory above 4 GB by PAE	Proof of concept
PoC by Bulygin et al. [76]	2013	UEFI	Bypass Seure Boot on Windows 8	Proof of concept
PoC by Zaddach et al. [116]	2013	UEFI	Hook the firmware for hard-drive operation	Proof of concept
OldBoot [117]	2014	Android	Attack the update process	In the wild
Darth Venamis [24]	2014	UEFI	Attack S3 boot script	Proof of concept
Thunderstrike [14]	2014	UEFI	Attack the thunderbolt port with a malicious Option ROM	Proof of concept
LightEater [118]	2015	UEFI	Expose sensitive information from firmware memory	Proof of concept
Vector EDK [119]	2015	UEFI	Implant rkloader by inserting an USB device	Partially leaked source code
SmmBackdoor [15]	2015	UEFI	Attack SMM	Proof of concept
Thunderstrike 2 [120]	2015	UEFI	Attack S3 resume and thunderbolt port simultaneously	Proof of concept
Memory Sinkhole [121]	2015	UEFI	Utilize the APIC to attack SMM	Proof of concept
PoC by Gorobets et al. [21]	2015	UEFI	Expose memory regions protected by VMM	Proof of concept
Safeguarding [122]	2016	UEFI	Rewrite the PPF and control the RoT	Proof of concept
PeiBackdoor [123]	2016	UEFI	Execute malicious code in PEI	Proof of concept
ThinkPwn [101]	2016	UEFI	Utilize a privilege escalation vulnerability in SMM	Proof of concept
AMT case by PLATINUM [93]	2017	UEFI	Utilize Intel AMT SOL for communication	In the wild
LoJax [12]	2018	UEFI	Utilize the Speed Racer to insert a DXE module	In the wild
MosaicRegressor [95]	2020	UEFI	Utilize a DXE driver to download payload from C&C servers	In the wild
FinSpy [2]	2021	UEFI	Modify OS bootloader in ESP to load unsigned drivers	In the wild
Netfilter [124]	2021	UEFI	Communicate with C&C servers as a Microsoft signed driver	In the wild
ESPecter [1]	2021	UEFI	Replace bootmgfw.efi in ESP and decrypt malicious code dynamically	In the wild
MoonBounce [3]	2022	UEFI	Utilize DXE implant to hook three critical function	In the wild
CosmicStrand [125]	2022	UEFI	Patch bootgmr to load malicious kernel driver	In the wild
Black Lotus [126]	2022	UEFI	Disable HVCI and Windows Defender to load unsigned drivers	In the wild

cause DXE was a relatively late phase in the UEFI boot process. The Verified and Measured Boot that was later introduced made the RoT shift to the hardware, which made bootkits hard to tamper with [77]. Due to the long firmware supply chain, manufacturers sometimes have a disconnect (i.e., some OEM manufacturers simplify or misconfigure the parameters of security mechanisms for performance or convenience) in the implementation of security mechanisms [127]. Simultaneously, more advanced security mechanisms such as VSM [68], STM [128], etc., have not been widely used due to performance overhead or complex configuration. So the security of UEFI firmware still faces many challenges today. Before LoJax was discovered in 2018, no typical UEFI bootkit had been found in the wild. Most UEFI bootkits were presented by researchers as PoC for research, and some of them did not implement the whole function of a bootkit but just proved an attack approach was feasible. However, the emergence of LoJax and the later discovered UEFI bootkits meant that UEFI bootkits have been lurking in the wild for many years, which shows that bootkit is still a massive threat in the wild.

6.2 Future Directions for Bootkits

In recent years, as major manufacturers have gradually deepened their understanding of security, the security of the boot process has been gradually improved. As a result, it is more and more difficult to implant a bootkit on a standard PC directly. More often, it is necessary to rely on the latest underlying software vulnerabilities to complete the tampering of the firmware. Simultaneously, the development of technology like edge-assisted IoT [129] makes the embedded IoT device more and more valuable for attackers to invade. And some commercial software, such as Kon-Boot that utilize bootkit techniques has also been iterated with the update of the operating systems. So we classify the future direction of development for bootkits into three categories:

With the latest vulnerability. Unlike the time when the system was designed without safety in mind (e.g., attackers could tamper with the bootstrap code by directly programming without utilizing any vulnerability in the legacy BIOS stage), the firmware systems and operating systems have been much harder to undermine since the popularity of UEFI. As we men-

tioned above, the difficulty of implanting bootkits in a PC is increasing due to major security companies having paid more and more attention to the security of underlying software. However, according to Eclypsium's report [130], there has been a significant increase in firmware vulnerabilities from 2016 to 2019. Furthermore, it reported that 80% of enterprises had experienced at least one firmware exploits in the last two years. The number of vulnerabilities disclosed can partly reflect the popularity of a security research field. As shown in Figure 7, we collect the numbers of CVEs with keywords "firmware," "BIOS," "UEFI," "Windows kernel," and "Linux kernel" from 2005 to 2021. For comparison, we also collected the total number of CVEs disclosed each year. We chose 2005 to start because it was the point at which the first bootkit first appeared. It is obvious that since the popularity of UEFI in 2013, the number of underlying software (e.g., keywords "firmware," "BIOS," and "UEFI") related vulnerabilities began to rise sharply. From Table 3, we can see that it was just the time point when UEFI bootkits started popping up. It is very likely that some of these vulnerabilities have been used in bootkit attacks. Simultaneously, we cross-referenced the numbers of vulnerability disclosures in other mainstream security research areas (i.e., OS kernel security). Figure 7d and Figure 7e show the trend in the number of vulnerabilities disclosed for Windows kernel and Linux kernel over the past decade. In the context of a continued rise in the number of CVEs disclosures overall (i.e., every black line in Figure 7), from the perspective of the number of CVEs, research interest in OS kernels peaked in 2017 but has since declined. In contrast, the number of CVEs in the field of underlying software security has continued to rise in recent years, indicating that underlying software security is becoming a research hotspot. Besides, some OEM vendors might deliberately leave backdoors in the firmware for debugging or other non-malicious purposes. Martin found that ChgBootDxeHook and SecureBackDoor in Lenovo laptops firmware could be used to disable SPI flash protections or Secure Boot directly [131], which has been identified by Lenovo as vulnerabilities in its products. This is also a factor that may have led to a large number of bootkit attacks.

With different platform. Secure Boot on PC has been widely adopted these years, while there is still a lack of protection on embedded systems due to

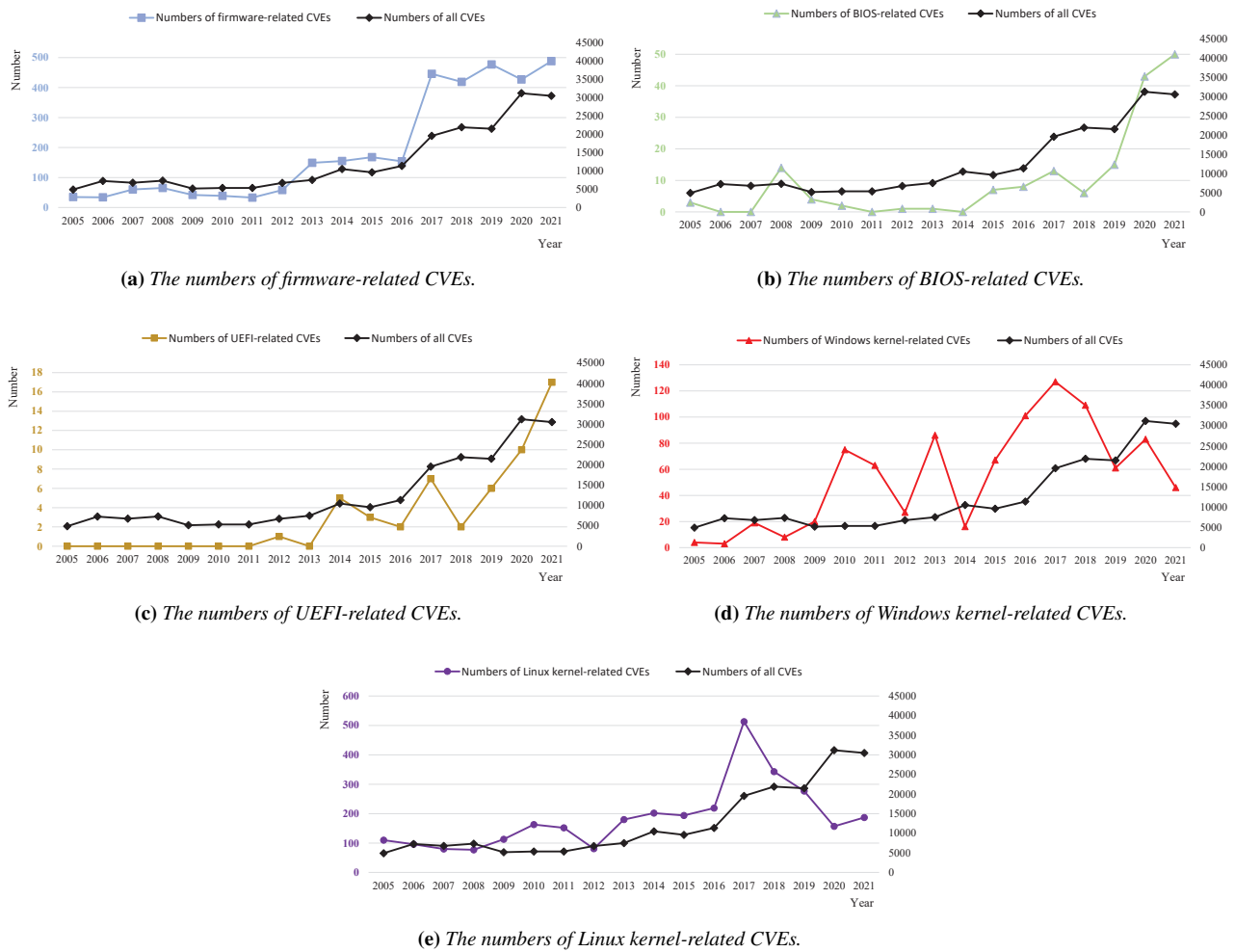


Figure 7. The numbers of CVEs with keywords “firmware”, “BIOS”, “UEFI”, “Windows kernel” and “Linux kernel”.

the limitation of hardware resources. On the other hand, the number of connected devices, from appliances to smart homes, has increased dramatically with the prevalence of IoT while also increasing the potential profit in IoT attacks. The cropped kernel, relatively simple security mechanism, and limited hardware resources make bootkit implantation on IoT devices relatively easy. There are already existing captured rootkits for IoT devices that are used in APT attacks [132]. It is possible that these mechanisms could involve bootkits, or maybe there are already existing uncaptured bootkits. Moreover, the benefit of undermining IoT devices with bootkits is excellent for attackers. Cisco predicted that the data generated by IoT devices would reach 507.5 zettabytes per year by 2019 [133], and the number is still growing up today. Once an IoT device is implanted by a bootkit, it will be con-

trolled by the attacker for a long time. The communication module of a bootkit enables the attacker to monitor or even manipulate the sensitive physical data of users [134]. Microsoft also introduced the Windows 10 IoT edition to support the IoT devices in UEFI [135], but it is a cropped edition and has fewer security features. Therefore, from a multi-faceted perspective, we believe that IoT will be the next main battlefield for bootkits.

With different purpose. Different from some bootkits that intend to perform malicious behaviors or even APT attacks in target systems, some intelligent software has utilized bootkit techniques to provide convenience to users. Computrace, also known as LoJack [36], was a proprietary laptop theft recovery software developed by Absolute Software. It was legitimately installed in the laptop’s firmware and uti-

lized the bootkit techniques in boot phase to run itself every time system booted. LoJax, found in 2018, was the malicious variant of LoJack developed by Sednit. But before LoJax was discovered, LoJack had been sold as commercial software for many years. Another typical instance is Kon-Boot. We conducted a reverse analysis of its implementation mechanism and found that it has no malicious behavior beyond its declaration. Kon-Boot is executed by choosing to boot from a USB flash disk in the UEFI shell. Then it sets hooks and directly loads kon64.bin into memory and executes it to bypass the password authentication, which is a typical bootkit technique in boot phase. Although Microsoft regarded Kon-Boot as a virus, Kon-Boot is still sold publicly and legally and even supports the latest Windows 11 [136]. We believe that it will be a trend to apply some bootkit techniques to legitimate commercial software for profit, and there will be more non-malicious bootkits like Kon-Boot in the future.

VII. CONCLUSION

This work comprehensively reviewed the techniques that bootkits used to hijack the control and execute malicious code before OS was in place. We proposed two axes to classify bootkit techniques systematically. To this end, we first introduced two mainstream boot processes. Further, we summarized an attack model based on many classic bootkits instances and discussed the functions of each component. We classified bootkits into two modes (i.e., legacy BIOS and UEFI) on the boot mode axis. In each category, we also divided the attack techniques into two phases (i.e., infection phase and boot phase) on the attack phase axis. For bootkits in legacy BIOS, we evaluated the LPE techniques, the four approaches bootkits utilized to hijack the control before OS, the hook methods, the self-defense techniques, and the security mechanisms to counter these attacks. As for bootkits in the UEFI stage, we introduced the security mechanisms as well as the corresponding attack techniques in the complicated firmware system. We analyzed the co-evolution of bootkits and security mechanisms and concluded that it was harder and harder for bootkits to compromise the boot process directly in UEFI. Finally, we summarized and proposed the future direction for bootkits based on the trend of firmware vulnerabilities in recent years, the potential profit in attack-

ing the embedded systems and IoT devices, and the legal as well as non-malicious usage of bootkit techniques. We hope that this survey could help the community focus on bootkits, which have been an enormous threat applied in firmware and OS security, and even APT attacks. Moreover, prioritize research efforts to address the emerging firmware security issues and bootkit threat issues.

ACKNOWLEDGEMENT

This work was supported by NSFC under Grant 62172308, Grant U1626107, Grant 61972297 and Grant 62172144.

References

- [1] "Uefi threats moving to the esp: Introducing espector bootkit," <https://www.welivesecurity.com/2021/10/05/uefi-threats-moving-esp-introducing-espector-bootkit/>, 2021.
- [2] "Finspy: Unseen findings," <https://securelist.com/finspy-unseen-findings/104322/>, 2021.
- [3] "Moonbounce: The dark side of uefi firmware," <https://securelist.com/moonbounce-the-dark-side-of-uefi-firmware/105468/>, 2022.
- [4] D. Zhao, C. Liu, *et al.*, "A security enhancement model based on switching edge strategy in interdependent heterogeneous cyber-physical systems," *China Communications*, 2022, vol. 19, no. 2, pp. 158–173.
- [5] I. Kuzminykh and M. Yevdokymenko, "Analysis of security of rootkit detection methods," in *2019 IEEE International Conference on Advanced Trends in Information Theory (ATIT)*. Kyiv, Ukraine: IEEE, 2019, pp. 196–199.
- [6] X. Li, Y. Wen, *et al.*, "An overview of bootkit attacking approaches," in *2011 Seventh International Conference on Mobile Ad-hoc and Sensor Networks*, 2011, pp. 428–431.
- [7] H. Gao, Q. Li, *et al.*, "Research on the working mechanism of bootkit," in *2012 8th International Conference on Information Science and Digital Content Technology (ICIDT2012)*. IEEE, 2012, vol. 3, pp. 476–479.
- [8] E. Rodionov and A. Matrosov, "The evolution of tdl: Conquering x64," *ESET*, 2011.
- [9] A. Matrosov and E. Rodionov, "Defeating x64: Modern trends of kernel-mode rootkits," 2011.
- [10] D. Soeder and R. Permeh, "Eeye bootroot," *Black Hat USA*, 2005.
- [11] M. Abd El-Aziz, A. Tarek, *et al.*, "Introduction to uefi technology," 2013.
- [12] "Lojax - first uefi rootkit found in the wild courtesy of the sednit group," <https://www.welivesecurity.com/wp-content/uploads/2018/09/ESET-LoJax.pdf>, 2018.
- [13] S. Kaczmarek, "Uefi and dreamboot," in *Hack in the Box Security Conference*, Kuala Lumpur, Malaysia, 2013.
- [14] T. Hudson and L. Rudolph, "Thunderstrike: Efi firmware bootkits for apple macbooks," in *Proceedings of the 8th*

- ACM International Systems and Storage Conference, *SYS-TOR*, D. Naor, G. Heiser, *et al.*, Eds. ACM, 2015, pp. 15:1–15:10.
- [15] D. Oleksiuk, “Building reliable smm backdoor for uefi based platforms,” URL: <http://blog.cr4.sh/2015/07/building-reliable-smm-backdoorfor-uefi.html>, 2015.
- [16] D. E. Rodionov, A. Matrosov, *et al.*, “Bootkits: Past, present and future,” in *VB Conference*. Citeseer, 2014.
- [17] A. Matrosov, E. Rodionov, *et al.*, *Rootkits and Bootkits: Reversing Modern Malware and Next Generation Threats*. No Starch Press, 2019.
- [18] N. Nikolov, “Research firmware update over the air from the cloud,” in *2018 IEEE XXVII International Scientific Conference Electronics-ET*, 2018, pp. 1–4.
- [19] C. BSDaemon, “D0nand0n. System management mode hack: Using smm for other purposes,” *Phrack Magazine*, 2008, vol. 12, p. 65.
- [20] S. Embleton, S. Sparks, *et al.*, “Smm rootkit: A new breed of os independent malware,” *Security and Communication Networks*, 2013, vol. 6, no. 12, pp. 1590–1605.
- [21] M. Gorobets, O. Bazhaniuk, *et al.*, “Attacking hypervisors via firmware and hardware,” *Black Hat USA*, 2015.
- [22] J. Yao and V. Zimmer, “S3 resume,” in *Building Secure Firmware*. Springer, 2020, pp. 313–333.
- [23] R. Wojtczuk and C. Kallenberg, “Attacking uefi boot script,” in *31st Chaos Communication Congress (31C3)*, 2014.
- [24] R. Wojtczuk and C. Kallenberg, “Attacks on uefi security, inspired by darth venamis’s misery and speed racer,” in *Proc. 15th Annu. CanSecWest Conf.(CanSecWest)*, 2015.
- [25] F. Périgaud, A. Gazet, *et al.*, “Subverting your server through its bmc: The hpe ilo4 case,” *Recon Brussels*, 2018.
- [26] X. Kovah and C. Kallenberg, “How many million bioses would you like to infect?” *Vancouver, BC, Canada. CanSecWest*, 2015.
- [27] D. Evdokimov, A. Ermolov, *et al.*, “Intel amt stealth breakthrough,” *Black Hat*, 2017.
- [28] I. Skochinsky, “Intel me secrets,” *Code Blue*, 2014.
- [29] M. Ermolov and M. Goryachy, “How to hack a turned-off computer, or running unsigned code in intel management engine,” *Black Hat Europe*, 2017.
- [30] A. Tereshkin and R. Wojtczuk, “Introducing ring-3 rootkits,” *Black Hat USA*, 2009.
- [31] A. Ogolyuk, A. Sheglov, *et al.*, “Uefi bios and intel management engine attack vectors and vulnerabilities.”
- [32] K. Dietrich and J. Winter, “Secure boot revisited,” in *2008 the 9th International Conference for Young Computer Scientists*. IEEE, 2008, pp. 2360–2365.
- [33] R. Wilkins and B. Richardson, “Uefi secure boot in modern computer security solutions,” in *UEFI Forum*, 2013.
- [34] S. Sanwald, M. Stöttinger, *et al.*, “Secure boot revisited: Challenges for secure implementations in the automotive domain,” *SAE Int. J. Transp. Cyber. & Privacy*, 2020.
- [35] B. Grill, A. Bacs, *et al.*, “‘nice boots!’ - a large-scale analysis of bootkits and new ways to stop them,” in *Detection of Intrusions and Malware, and Vulnerability Assessment - 12th International Conference, DIMVA 2015, Milan, Italy, July 9-10, 2015, Proceedings*, ser. Lecture Notes in Computer Science, M. Almgren, V. Gulisano, *et al.*, Eds. Springer, 2015, vol. 9148, pp. 25–45.
- [36] “Computrace by absolute software now supported in firmware of getac computers,” <https://www.absolute.com/company/press-releases/2009/computrace-by-absolute-software-now-supported-in-firmware-of-getac-computers/>, 2009.
- [37] “Forgot windows password? Bypass it with kon-boot!” <https://kon-boot.com/>, 2022.
- [38] “Edk ii build specification,” <https://edk2-docs.gitbook.io/edk-ii-build-specification>, 2021.
- [39] P. Kleissner, “Stoned bootkit,” *Black Hat USA*, 2009, pp. 5–7.
- [40] N. Kumar and V. Kumar, “Vbootkit 2.0-attacking windows 7 via boot sectors,” in *Proceedings of the Hack in the Box Conference (HITBSecConf)*, 2009.
- [41] B. Grill, “Bootkits revisited: Detecting, analysing and mitigating bootkit threats,” Ph.D. Dissertation, Wien, 2016.
- [42] E. Rodionov and A. Matrosov, “Mind the gapz: The most complex bootkit ever analyzed?” <https://www.welivesecurity.com/wp-content/uploads/2013/04/gapz-bootkit-whitepaper.pdf>, 2013.
- [43] R. Dolmans and W. Katz, “Rp1: Carberp malware analysis,” 2013.
- [44] A. Matrosov, “Olmasco bootkit: Next circle of tdl4 evolution (or not?).” <https://www.welivesecurity.com/2012/10/18/olmasco-bootkit-next-circle-of-tdl4-evolution-or-not-2/>, 2012.
- [45] M. Baezner and P. Robin, “Stuxnet,” *ETH Zurich, Tech. Rep.*, 2017.
- [46] C. Kallenberg and R. Wojtczuk, “Speed racer: Exploiting an intel flash protection race condition,” *Bromium Labs*, 2015.
- [47] M. Alzaidei, “The study of ssdt hook through comparative analysis between live response and memory image,” 2012.
- [48] Y. Wang, D. Gu, *et al.*, “Virus analysis on idt hooks of rootkits trojan,” in *2009 International Symposium on Information Engineering and Electronic Commerce*. IEEE, 2009, pp. 224–228.
- [49] E. Rodionov and A. Matrosov, “Defeating anti-forensics in contemporary complex threats,” in *ESET*, 2012.
- [50] A. Matrosov, “Eset-rovnix bootkit framework updated,” 2012.
- [51] N. Kumar and V. Kumar, “Vbootkit: Compromising windows vista security,” *Black Hat Europe*, 2007, vol. 2007.
- [52] E. Florio and K. Kasslin, “Your computer is now stoned (... again!),” *Virus Bulletin*, 2008.
- [53] A. Matrosov and E. Rodionov, “Festi botnet analysis and investigation,” in *Proc. 15th AVAR Conf*, 2011, pp. 12–14.
- [54] T. Wüchner, M. Ochoa, *et al.*, “Malflow: Identification of c&c servers through host-based data flow profiling,” in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, 2016, pp. 2087–2094.
- [55] “Cve-2010-4398,” <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-4398>, 2010.
- [56] “Cve-2011-3402,” <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-3402>, 2011.
- [57] C. Huang, X. Han, *et al.*, “Lpet – mining ms-windows software privilege escalation vulnerabilities by monitoring interactive behavior,” in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications*

- Security, 2020, pp. 2089–2091.
- [58] M. O’Leary, “Privilege escalation in linux,” in *Cyber Operations*. Springer, 2019, pp. 419–453.
- [59] S. Hutchinson and C. Varol, “A survey of privilege escalation detection in android,” in *2018 9th IEEE Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*, 2018, pp. 726–731.
- [60] K. Kujanpää, W. Victor, *et al.*, “Automating privilege escalation with deep reinforcement learning,” in *Proceedings of the 14th ACM Workshop on Artificial Intelligence and Security*, 2021, pp. 157–168.
- [61] Z. Paróczy, “Lz based compression benchmark on pe files,” *Acta Cybernetica*, 2019, vol. 24, no. 1, pp. 121–129.
- [62] R. R. Branco, G. N. Barbosa, *et al.*, “Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies,” *Black Hat*, 2012, vol. 1, pp. 1–27.
- [63] K. Maduranga, “Investigate windows management instrumentation (wmi) attacks in windows operating systems,” Ph.D. Dissertation, 2017.
- [64] B. Zhang, “Research summary of anti-debugging technology,” in *Journal of Physics: Conference Series*. IOP Publishing, 2021, vol. 1744, p. 042186.
- [65] A. Milenkoski, “Elam: The windows defender elam driver,” Ph.D. Dissertation, ERNW Enno Rey Netzwerke GmbH, 2019.
- [66] I. Korkin, “Hypervisor-based active data protection for integrity and confidentiality of dynamically allocated memory in windows kernel,” *arXiv preprint arXiv:1805.11847*, 2018.
- [67] L. Kessem, “The necurs botnet: A pandora’s box of malicious spam,” 2019.
- [68] A. Milenkoski and D. Phillips, “Virtual secure mode: Architecture overview,” Ph.D. Dissertation, ERNW Enno Rey Netzwerke GmbH, 2019.
- [69] V. Costan and S. Devadas, “Intel sgx explained,” *IACR Cryptol. ePrint Arch.*, 2016, vol. 2016, no. 86, pp. 1–118.
- [70] T. Fukai, S. Takekoshi, *et al.*, “Bmcarmor: A hardware protection scheme for bare-metal clouds,” in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2017, pp. 322–330.
- [71] J. Yao, V. J. Zimmer, *et al.*, “A tour beyond bios: Using iommu for dma protection in uefi firmware,” 2017.
- [72] J. Yao, “Smm protection in edk ii,” *UEFI Plugfest 2017*, 2017.
- [73] R. Wojtczuk and A. Tereshkin, “Attacking intel bios,” *BlackHat, Las Vegas, USA*, 2009.
- [74] V. Bashun, A. Sergeev, *et al.*, “Too young to be secure: Analysis of uefi threats and vulnerabilities,” in *14th Conference of Open Innovation Association FRUCT*. Espoo, Finland: IEEE, 2013, pp. 16–24.
- [75] A. Matrosov and E. Rodionov, “Uefi firmware rootkits: Myths and reality,” *Black Hat Asia*, 2017.
- [76] Y. Bulygin, A. Furtak, *et al.*, “A tale of one software bypass of windows 8 secure boot,” *Black Hat USA*, 2013.
- [77] J. Yao and V. Zimmer, “Trusted boot,” in *Building Secure Firmware*. Springer, 2020, pp. 197–255.
- [78] Y. Bulygin, J. Loucaides, *et al.*, “Summary of attacks against bios and secure boot,” *Proceedings of the DefCon*, 2014.
- [79] C. Kallenberg, S. Cornwell, *et al.*, “Setup for failure: Defeating secure boot,” in *the Symposium on Security for Asia Network (SyScan)(April 2014)*, 2014.
- [80] A. Matrosov, “Betraying the bios: Where the guardians of the bios are failing,” 2017.
- [81] M. Ermolov and M. Goryachy, “Intel visa: Through the rabbit hole,” *Black Hat Asia*, 2019.
- [82] E. D. Wolff, K. M. GroWIEy, *et al.*, “Navigating the solarwinds supply chain attack,” 2021.
- [83] D. Rohret and J. Willmann, “Supply chain attacks: Basic input output systems (bios), mux multiplexers and field programmable gate arrays (fpga),” in *International Conference on Cyber Warfare and Security*. Academic Conferences International Limited, 2012, p. 254.
- [84] W. J. Heinbockel, E. R. Laderman, *et al.*, “Supply chain attacks and resiliency mitigations,” *The MITRE Corporation*, 2017, pp. 1–30.
- [85] D. L. Vu, I. Pashchenko, *et al.*, “Towards using source code repositories to identify software supply chain attacks,” in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 2093–2095.
- [86] A. Yeboah-Ofori and S. Islam, “Cyber security threat modeling for supply chain organizational environments,” *Future Internet*, 2019, vol. 11, no. 3, p. 63.
- [87] D. Ibdah, N. Lachtar, *et al.*, “Dark firmware: A systematic approach to exploring application security risks in the presence of untrusted firmware,” in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, 2020, pp. 413–426.
- [88] “A deeper uefi dive into moonbounce,” https://www.binarly.io/posts/A-deeper-UEFI-dive_into_MoonBounce/index.html, 2022.
- [89] L. Zhou, F. Zhang, *et al.*, “A coprocessor-based introspection framework via intel management engine,” *IEEE Transactions on Dependable and Secure Computing*, 2021, no. 1, pp. 1920–1932.
- [90] “Cve-2017-5705,” <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5705>, 2017.
- [91] “Cve-2017-5706,” <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5706>, 2017.
- [92] “Cve-2017-5707,” <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5707>, 2017.
- [93] “Platinum continues to evolve, find ways to maintain invisibility,” <https://www.microsoft.com/security/blog/2017/06/07/platinum-continues-to-evolve-find-ways-to-maintain-invisibility/>, 2017.
- [94] A. Matrosov and A. Gazet, “Breaking through another side,” *Blackhat US 2019*.
- [95] “Mosaicregressor: Lurking in the shadows of uefi,” <https://securelist.com/mosaicregressor/98849/>, 2020.
- [96] L. Zhou, J. Xiao, *et al.*, “Nighthawk: Transparent system introspection from ring-3,” in *European Symposium on Research in Computer Security*. Springer, 2019, pp. 217–238.
- [97] D. Wang and W. Y. Dong, “Attacking intel uefi by using cache poisoning,” in *Journal of Physics: Conference Series*. IOP Publishing, 2019, vol. 1187, p. 042072.
- [98] L. Zhao and D. Lie, “Is hardware more secure than software?” *IEEE Security & Privacy*, 2020, vol. 18, no. 5, pp.

8–17.

- [99] A. Furtak, Y. Bulygin, *et al.*, “Bios and secure boot attacks uncovered,” in *the 10th Ekoparty Security Conference*, 2014.
- [100] “Cve-2021-26943,” <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-26943>, 2021.
- [101] D. Oleksiuk, “Exploring and exploiting lenovo firmware secrets,” 2016.
- [102] J. Yin, M. Li, *et al.*, “Finding smm privilege-escalation vulnerabilities in uefi firmware with protocol-centric static analysis,” in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2022, pp. 1570–1570.
- [103] “A tour beyond bios implementing s3 resume with edk ii,” <https://docplayer.net/32893451-A-tour-beyond-bios-implementing-s3-resume-with-edkii.html>, 2015.
- [104] Intel®, “Intel® hardware shield - below-the-os security,” <https://www.intel.com/content/dam/www/central-libraries/us/en/documents/below-the-os-security-white-paper.pdf>, 2021.
- [105] “Virus.win9x.cih,” <https://threats.kaspersky.com/en/threat/Virus.Win9x.CIH/>, 2002.
- [106] J. Heasman, “Implementing and detecting an acpi bios rootkit,” *Black Hat Federal*, 2006, vol. 368.
- [107] L. K. Yang, D. Chen, *et al.*, “Research on malicious behavior of firmware based on hardware resources access control,” in *Applied Mechanics and Materials*. Trans Tech Publ, 2013, vol. 340, pp. 737–743.
- [108] “Are mbr infections back in fashion? (infographic),” <https://www.kashifali.ca/2011/08/are-mbr-infections-back-in-fashion-infographic/>, 2011.
- [109] A. Ortega, A. Sacco, *et al.*, “Deactivate the rootkit: Attacks on bios anti-theft technologies,” 2009.
- [110] W. Ettlinger and S. Vieböck, “Evil core bootkit: Pwning multiprocessor systems,” http://downloads.ninjacon.net/downloads/proceedings/2011/Ettlinger_Viehoeck-Evil_Core_Bootkit.pdf, 2011.
- [111] M. Giuliani, “Mebromi: The first bios rootkit in the wild,” *Blog*. <http://blog.webroot.com/2011/09/13/mebromi-the-first-bios-rootkit-in-the-wild/>. Retrieved, 2011, pp. 09–19.
- [112] N. A. Economou and D. Juarez, “Vga persistent rootkit,” https://www.coresecurity.com/sites/default/files/private-files/publications/2016/05/corelabs-ekoparty-2012-VGA_Persistent.Rootkit.pdf, 2012.
- [113] S. Anthony, “Rakshasa: The hardware backdoor that china could embed in every computer,” <https://www.extremetech.com/computing/133773-rakshasa-the-hardware-backdoor-that-china-could-embed-in-every-computer>, 2012.
- [114] D. Goodin, “Meet badbios, the mysterious mac and pc malware that jumps airgaps,” *Ars Technica*, 2013, vol. 31, no. 10.
- [115] “X86 memory bootkit,” <https://github.com/AaL186/retraware/tree/master/MemoryBootkit>, 2013.
- [116] J. Zaddach, A. Kurmus, *et al.*, “Implementation and implications of a stealth hard-drive backdoor,” in *Proceedings of the 29th Annual Computer Security Applications Conference*, 2013, pp. 279–288.
- [117] Z. Xiao and Q. Dong, “Oldboot: The first bootkit on android,” <http://blogs.360.cn/360mobile/2014/01/17/oldboot-the-first-bootkit-on-android/>, 2014.
- [118] X. Kovah and C. Kallenberg, “Are you giving firmware attackers a free pass?” in *Proceedings of the RSA Conference, San Francisco, CA, USA*, 2015, pp. 20–24.
- [119] “Vector edk,” <https://github.com/hackedteam/vector-edk>, 2015.
- [120] T. Hudson, X. Kovah, *et al.*, “Thunderstrike 2: Sith strike,” *Black Hat USA Briefings*, 2015.
- [121] C. Domas, “The memory sinkhole,” *BlackHat USA*, 2015.
- [122] A. Ermolov, “Safeguarding rootkits: Intel boot guard,” *Zero nights 2016*.
- [123] A. Mytkäniemi, “Riskiarvio työpöytä- ja mobiililustojen käynnistyslaiteohjelmistojen kyberuhista,” 2021.
- [124] “Microsoft admits to signing rootkit malware in supply-chain fiasco,” <https://www.bleepingcomputer.com/news/security/microsoft-admits-to-signing-rootkit-malware-in-supply-chain-fiasco/>, 2021.
- [125] GReAT, “Cosmicstrand: The discovery of a sophisticated uefi firmware rootkit,” <https://securelist.com/cosmicstrand-uefi-firmware-rootkit/106973/>, 2022.
- [126] P. Paganini, “New uefi rootkit black lotus offered for sale at \$5,000,” <https://securityaffairs.co/wordpress/137252/malware/black-lotus-uefi-rootkit.html>, 2022.
- [127] “Repeatable firmware security failures: 16 high impact vulnerabilities discovered in hp devices,” https://www.binarly.io/posts/Repeatable_Firmware_Security_Failures_16_High_Impact_Vulnerabilities_Discovered_in_HP_Devices/index.html, 2022.
- [128] E. D. Myers, “Using the intel stm for protected execution,” 2018.
- [129] S. Shen, K. Zhang, *et al.*, “Security in edge-assisted internet of things: Challenges and solutions,” *Science China Information Sciences*, 2020, vol. 63, no. 12, pp. 1–14.
- [130] “Nist kicks the can: Why we need to address firmware security now,” <https://eclipsium.com/2021/07/09/nist-kicks-the-can-why-we-need-to-address-firmware-security-now/>, 2021.
- [131] M. Smolár, “When ‘secure’ isn’t secure at all: High-impact uefi vulnerabilities discovered in lenovo consumer laptops,” <https://www.welivesecurity.com/2022/04/19/when-secure-isnt-secure-uefi-vulnerabilities-lenovo-consumer-laptops/>, 2022.
- [132] “Drovorub linux rootkit iocs,” <https://threatravens.com/drovorub-linux-rootkit-iocs/>, 2020.
- [133] J. Ni, X. Lin, *et al.*, “Toward edge-assisted internet of things: From security and efficiency perspectives,” *IEEE Network*, 2019, vol. 33, no. 2, pp. 50–57.
- [134] J. Wang, X. Wang, *et al.*, “Physical layer security for uav communications: A comprehensive survey,” *China Communications*, 2022, vol. 19, no. 9, pp. 77–115.
- [135] “Windows 10 iot,” <https://developer.microsoft.com/en-us/windows/iot/>, 2021.
- [136] Microsoft, “Virtool:dos/konboot threat description,” <https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?name=VirTool:DOS/Konboot&threatid=2147624111>, 2017.

Biographies



Yilin Zhou received the B.S. degree from Wuhan University (WHU) in 2020. He is currently a PhD student in School of Cyber Science and Engineering, WHU. His current research interests include system security, cyber security, and firmware security.



Guojun Peng is currently a professor at the School of Cyber Science and Engineering, Wuhan University. He received the Ph.D. degree from Wuhan University in 2008. His research interests are system security and trust computing. He is a member of CCF and CSAC.



Zichuan Li is currently a second-year master student in School of Cyber Science and Engineering, WHU. His research interests include vulnerability detection, vulnerability exploitation and firmware security.



Side Liu received the B.S. degree from Wuhan University (WHU) in 2019. He is currently a PhD student in School of Cyber Science and Engineering, WHU. His research interests include malware analysis and system security.