Isolate and Detect the Untrusted Driver with a Virtual Box

YongGang Li
liyg@cumt.edu.cn
School of Computer Science and Technology, China University of Mining and Technology;
Mine Digitization Engineering Research Center of the Ministry of Education;
Xuzhou, Jiangsu, China

ShunRong Jiang
jsywow@163.com
School of Computer Science and Technology, China University of Mining and Technology;
Mine Digitization Engineering Research Center of the Ministry of Education;
Xuzhou, Jiangsu, China

Yu Bao
baoy@cumt.edu.cn
School of Computer Science and Technology, China University of Mining and Technology;
Mine Digitization Engineering Research Center of the Ministry of Education;
Xuzhou, Jiangsu, China

PengPeng Chen
chenp@cumt.edu.cn
School of Computer Science and Technology, China University of Mining and Technology;
Mine Digitization Engineering Research Center of the Ministry of Education;
Xuzhou, Jiangsu, China

Yong Zhou
yzhou@cumt.edu.cn
School of Computer Science and Technology, China University of Mining and Technology;
Mine Digitization Engineering Research Center of the Ministry of Education;
Xuzhou, Jiangsu, China

Yeh-Ching Chung
ychung@cuhk.edu.cn
Chinese University of HongKong, ShenZhen;
ShenZhen, GuangDong, China

ABSTRACT

In kernel, the driver code is much more than the core code, thus having a larger attack surface. Especially for the untrusted drivers without source code, they may come from the hot-plug hardware or the user without security knowledge. Traditional isolation methods require analyzing source code to set checkpoints in the driver for control flow protection, which are not available for closed-source drivers. Even worse, the existing isolation methods can only prevent the hijacked control flows entering/existing drivers, while they cannot discover the illegal control flows inside drivers. Although the kernel address space location randomization (KASLR) can defend against control flow hijacking, it can be bypassed by code probes. In response to these issues, this paper proposes a novel method Dbox to isolate and detect the untrusted drivers whose source code is unavailable. Dbox creates a light hypervisor to monitor and analyze the untrusted driver's behavior without relying on source code. It isolates the untrusted driver in a private space and dynamically changes its virtual space through a sliding space mechanism. Under the protection of Dbox, all control flows jumping to/from untrusted drivers can be detected. Experiments and analysis show that Dbox has good protection against code probes, kernel rootkits and code reuse attacks, and the overhead introduced to the operating system is less than 3.6% in general scenarios.

CCS CONCEPTS

• **Security and privacy → Virtualization and security**.

KEYWORDS

Memory Isolation, Driver Security, Code Reuse Attacks, Rootkits

ACM Reference Format:
YongGang Li, ShunRong Jiang, Yu Bao, PengPeng Chen, Yong Zhou, and Yeh-Ching Chung. 2024. Isolate and Detect the Untrusted Driver with a Virtual Box. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3658644.3670269

1 INTRODUCTION

Linux drivers are loaded into the kernel in the form of loadable kernel modules (LKMs). Drivers typically run with high privilege in the kernel. In a driver, any malicious behavior can lead to catastrophic consequences, such as kernel crash. The control flow hijacking is one of the key threats faced by drivers. Kernel rootkits [19, 39] and code reuse attacks (CRAs) [2, 5, 35, 37] are the main threats to driver control flows. Kernel rootkits can hijack the control flow that should flow to the core kernel into malicious drivers. In contrast, CRAs can exploit vulnerabilities to hijack control flows to any code locations.

To prevent drivers from triggering illegal control flows, they need to be audited before being loaded into the kernel. The drivers that have not been audited are untrusted. In practice, users may directly load non-audited drivers into the kernel without any audit. Such drivers pose serious threats to the operating system (OS). Even worse, not all drivers are open-source. For example, Nvidia did not disclose its driver source code until 2022. For the drivers whose source code is unavailable (called untrusted drivers in this paper), all methods relying on source code or compiler cannot prevent their

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3670269

intentional or unintentional illegal activities. For security reasons, all control flows jumping into untrusted drivers (entry_flow) and all control flows jumping out of untrusted drivers (exit_flow) need to be detected.

From the perspective of attack principles, the drivers with malicious attributes can launch attacks during the loading and running stages. For example, kernel rootkits [19] can tamper with kernel data structures through their initialization functions at loading stage. Afterwards, the illegal control flow can be triggered under specific conditions, such as a system call. The most direct way to defend against such attacks is to use a standardized control flow graph (CFG) to detect the control flows. Although we can build a CFG for the closed-source driver by analyzing its binary code, the CFG will rapidly expand as the driver size increases. An expanded CFG contains some paths that do not exist in real execution scenarios, which will lead to misjudgment. In theory, ensuring the integrity of all the data that can affect control flows can prevent control flow hijacking launched by malicious drivers. However, such data is distributed throughout the kernel space and is dynamically created. Tracking all the data introduces significant overhead.

From the perspective of running modes, the drivers without malicious attributes can also be used maliciously. For example, CRAs can launch attacks through memory vulnerabilities in the kernel, and use the instruction *ret* contained in drivers to chain attack payloads (gadgets) [10, 15, 18]. For drivers, all entry_flows and exit_flows should be transferred through specific interfaces. Entry_flows are transferred to the driver by system calls, interrupts, and operation functions of device files. In contrast, exit_flows will be transferred to the locations outside the driver through the functions in core kernel and the exported functions in other LKMs. In theory, forcing all control flows to enter/exit drivers from specific interfaces can defend against control flow hijacking [11]. However, for the untrusted drivers without source code, the existing methods cannot accurately locate all specific interfaces by analyzing binary code.

Moreover, the traditional isolation methods can only restrict the paths that control flows jump to/from drivers, and cannot prevent illegal control flows inside drivers. In real attack scenarios, except for entry_flows and exit_flows, the control flows inside the untrusted driver can also be hijacked. For example, attackers can exploit memory vulnerabilities in the driver to launch CRAs and only use the driver code to build a complete gadget chain. Authough the existing KASLR methods [6, 12, 28] can hide code addresses from attackers to defend against illegal control flows, they need to pre-handle the source code of drivers if they want to re-randomize the running binary code in physical pages, which makes them invalid for the closed-source drivers. In contrast, some other KASLR methods can change the base address of the driver during the loading or linking phase without analyzing source code, they cannot re-randomize the driver code when the diriver is running. Even worse, existing research indicates that code probes [3, 8, 24, 29, 33] can bypass address space location randomization (ASLR).

To solve the above problems, we propose a method Dbox to isolate and detect the untrusted drivers. Dbox detects the control flows and changes the space locations of untrusted drivers at their running stage. Both changing space locations and detecting control flows are based on the execution logic of the untrusted drivers. However, for the untrusted drivers without source code, their execution logic is unknown. Fortunately, the execution logic can be obtained from the context of the running binary code of the driver. Therefore, we can obtain drivers' execution logic by capturing and analyzing their context. For example, for the indirect control transfer (ICT) instruction *call *pointer*, we can read the *pointer* to obtain its accurate jump target when the instruction is being executed. Based on real-time execution logic, we can check the legitimacy of the control flow based on code logic and data logic. And we can also ensure the driver code whose space locations have been changed maintains the original calling relationship. In summary, the contributions of this paper are as follows:

1) Create a lightweight hypervisor. Combining hardware-assisted virtualization, we establish a lightweight hypervisor. Unlike existing VMMs, our self-developed hypervisor does not provide complex virtual machine management mechanisms, but tracks and monitors running targets. It can monitor and control the data access and code execution of closed-source drivers, thereby extracting the context during their runtime. By analyzing the running context, we can obtain accurate execution logic without building a complex CFG.

2) Propose a sliding space mechanism. This mechanism isolates the untrusted driver in a private space. All entry_flows and exit_flows can be captured by adjusting the private space. Moreover, the driver space can be dynamically slid, which can defend against code probes.

3) Propose a detection method for entry_flows and exit_flows. Due to the sliding space, all control flows transferred to the untrusted drivers can be captured. Meanwhile, the control flows inside the driver space triggered by ICT instructions will be transformed into exit_flows and they can be captured. All entry_flows and exit_flows will be tracked and detected. This method can defend against kernel rootkits and CRAs.

2 RELATED WORK

To prevent drivers from breaking the control flow integrity (CFI) in kernel, researchers have proposed various methods. These methods can be divided into control data protection, memory isolation, and KASLR. In this section, we introduce them separately.

Control data protection. Malicious drivers can hijack control flows by tampering with control data. In theory, protecting control data can prevent illegal control flows of malicious drivers. Based on this principle, VTW [19] defend against kernel rootkits by checking control data. However, the control data it can detect is fixed, which results in poor detection on rootkit variants. To cover all legitimate control flow paths, existing methods, such as Ge [7] and FINE-CFI [16], establish accurate CFGs, which can be used to detect the tampered control data. However, it's impossible to establish a high-precision CFG for the untrusted driver without source code before it runs. The hardware technology ARM Pointer Authentication can be used to protect the integrity of function pointers when the binary code is running. It has been adopted by existing security methods, such as PATTER[41] and PAL[42]. These methods require analyzing the source code to locate all pointers when deployed, and they are ineffective for closed-source drivers.

Memory isolation. Memory isolation constructs an isolation layer between the target driver and other objects, through which only authorized access can pass. Gateway[34] hardens the kernel API to drivers by isolating driver code in a different memory address space. UnderBridge[9] moves the OS components of a microkernel between user space and kernel space at runtime while enforcing consistent isolation. LXFI[22] isolates kernel modules from the core kernel so that vulnerabilities in kernel modules cannot lead to a privilege escalation attack. MemCat[27] separates data based on attacker control to mitigate the exploitation of memory corruption vulnerabilities such as use-after-free and use-after-return. TDI [23] isolates memory objects of different colors in separate memory arenas and uses efficient compiler instrumentation to constrain loads to the arena of the intended color by construction. TZ-RKP[1] provides a complete and feasible isolation layer for kernel using TrustZone. $\text{kR}\hat{\text{X}}$ [30] is based on execute-only memory and code diversification, and it can benefit from hardware support to optimize performance. The above methods can detect the control flows jumping to/from the target drivers. However, they have a same limitation. That is they cannot detect the control flows inside the drivers. In addition, compiler-based methods, such as LXFI [22] and TDI [23], cannot construct an effective isolation layer during binary code generation if their protection targets are closed-source drivers.

KASLR. Traditional ASLR methods, such as fASLR [31], PointerScope [40], and Mardu [14], rely on source code analysis, which makes them unable to be applied to closed-source objects. Renanz [38] and RuntimeASLR [21] can randomize closed-source objects, but they introduce significant overhead. KASLR can hide kernel addresses, thereby preventing CRAs from chaining gadgets [18]. Adelie [28] enables efficient continuous KASLR for LKMs by using the PIC (position-independent code) model to make it hard to inject ROP gadgets through modules regardless of gadget's origin. KASLR-MT[36] is also a kernel randomization method for multi-tenant cloud systems, which maximizes memory savings rate. However, existing KASLR methods can be bypassed by code probes [17]. Moreover, KASLR needs to be completed between the occurrence of code probes and CRAs. Otherwise, it is invalid and can only incur overhead. How to choose an appropriate randomization time point is a challenge faced by all KASLR methods.

Compared with existing methods, our method Dbox is a method specifically designed to detect illegal control flows caused by closed-source drivers. Dbox captures the execution context in real-time by monitoring the data access and code execution of running drivers. The call relationships between different code blocks can be obtained by analyzing the captured context. Based on this design, we can avoid building a high-precision CFG for the closed-source drivers. Meanwhile, Dbox does not randomize physical code like existing KASLR methods, but dynamically changes the virtual space of the running drivers. Therefore, we don't need to maintain complex calling relationships for the randomized code.

3 ASSUMPTIONS AND ATTACK VECTORS

First, we assume attackers can inject untrusted drivers into the OS. In practice, untrusted drivers come from the hot-plug hardware or the closed-source objects installed by users. An open OS does not prevent root users from installing untrusted drivers. Second, we assume attackers can exploit memory vulnerabilities to launch control flow attacks. Overflow vulnerabilities are widely distributed in C++/C programs, which can be used to tamper with control data, such as return addresses or function pointers. Third, we assume attackers can use the techniques [8, 29, 33] to probe driver code. For example, attackers can directly read driver code and filter out the code snippets that match specific gadget forms. The threat model used in this paper includes the following 5 attack vectors:

Vector 1: Kernel rootkits. kernel rootkits are loaded into the kernel as LKMs, which can modify kernel control data, such as system call tables, and hijack kernel control flows.

Vector 2: The CRAs whose gadgets are located in different objects. The gadgets used by such CRAs are partly from untrusted drivers, and partly from the core kernel and other LKMs.

Vector 3: The CRAs whose gadgets are located in a same driver. That is, attackers can build a complete gadget chain using the code snippets in a single driver.

Vector 4: Code reading probes. Attackers can bypass KASLR by directly reading the driver code.

Vector 5: Code pointer probes. Attackers can bypass KASLR by reading the return address stored on the stack and the pointers stored in data structures.

4 MOTIVATION

The purpose of Dbox is to isolate the untrusted driver and discover the illegal control flows related to it without analyzing or compiling its source code. The defense targets include kernel rootkits, code probes, and CRAs. In summary, the motivations of this paper are as follows:

1) Get rid of dependence on source code. The execution logic of a closed-source driver is unknown. The unknown execution logic can hinder KASLR methods from maintaining the original calling relationship after re-randomizing driver code. Moreover, the unknown logic prevents traditional CFI methods from pre-collecting instruction boundaries to detect illegal control flows. For the untrusted drivers without source code, we can obtain accurate execution logic by capturing and analyzing their execution context.

2) Isolate the untrusted driver dynamically. The untrusted driver will be isolated in a private space, which can capture code probes. If there is a code probe, the driver space will slide dynamically. Unlike the existing KASLR methods, we change the untrusted driver's virtual space instead of its physical space, which can avoid frequent memory operations.

3) Filter out illegal control flows related to untrusted drivers. We prevent attackers from redirecting illegal control flows to untrusted drivers and hijacking control flows in untrusted drivers. Compared with existing methods, it does not use fixed instruction boundaries to constrain control flows, but detects control flows based on runtime context.

5 METHODOLOGY

The overall design of Dbox is shown in Figure 1. Dbox isolates the untrusted driver in a private space. This space is not fixed. It will dynamically slide along with the driver execution, which can defend against code probes. Moreover, all incoming/outgoing

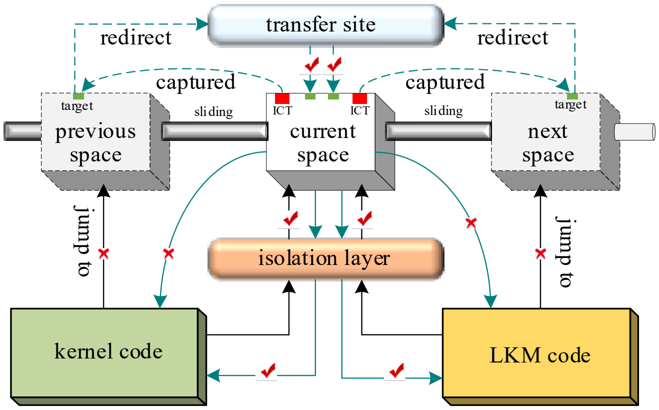

Figure 1: The overall design of Dbox

control flows need to pass through an isolation layer, and only legal control flows can jump to the target locations. Meanwhile, to ensure the control flows inside the untrusted driver are not hijacked, all internal control flows triggered by ICT instructions need to go through a transfer site to jump to the right locations. Under the protection of Dbox, code probe attacks cannot locate the executable code, and the illegal control flows can also be detected.

Our detection targets are the untrusted drivers without source code. We cannot obtain their behavior characteristics through static analysis, nor can we preset checkpoints in appropriate locations. Faced with these challenges, we build a lightweight hypervisor [20], as shown in Figure 2. Similar to KVM, it is loaded into the kernel as an LKM. Unlike the KVM, its main function is not to manage virtual machines, but to utilize EPT (Extended Page Tables), VMX (Virtual Machine Extension), and VT-d (Virtualization Technology for Directed I/O) to monitor and control drivers. Moreover, the hypervisor utilizes VMX root and VMX non-root to divide the running modes of the original OS into two types: *host* and *guest*. Under normal scenarios, the OS runs in the *guest* mode. When a specific event occurs, the running mode switches from *guest* to *host*, which is called a system trap. Combining EPT and VMX, multiple trap events can be set, including memory permission violation, page directory switching, breakpoint setting, execution of specific instructions (such as *int3*, *vmcall*), interrupts, single step debugging, and exception protection, etc. In addition, the CPU context can also be rewritten by modifying the fields in VMCS (Virtual Machine Control Structure) to achieve execution control. For example, by modifying the *guest rip* in VMCS, control flow redirection can be achieved. Based on the above designs, Dbox can monitor and control the behaviors of untrusted drivers without relying on source code.

To filter out the illegal control flows, we must capture the activities of control flow transfers. Control flow hijacking often occurs at the moment when the control flow enters or exits an untrusted driver. The differences between illegal control flows and legal control flows are reflected in the execution context at this moment. For example, the VFS (virtual file system) function pointer used by the kernel rootkit *adore-ng* no longer points to the original kernel code but rather to rootkit code. As a result, we can detect illegal control flows by analyzing the execution context. However, after this moment, the tampered control data (such as return addresses) may be recycled, which results the malicious action cannot be prevented in a timely manner. Therefore, we need to capture the entry_flows and exit_flows and extracts their context in real time.

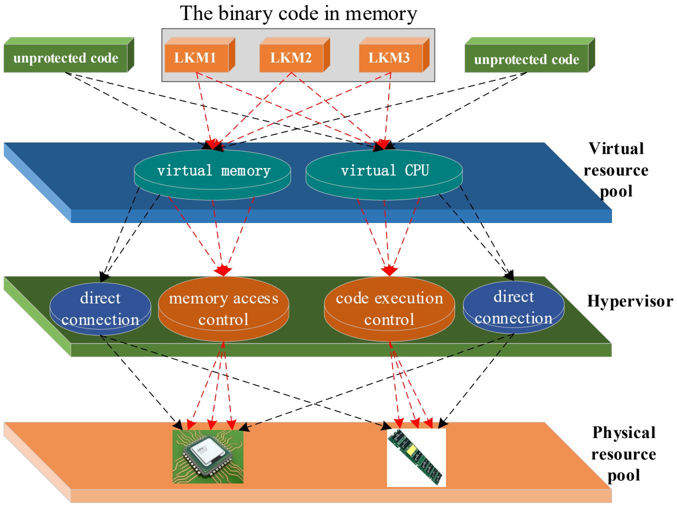

Figure 2: The overall design of the customized hypervisor

To capture entry_flows and exit_flows, we propose a sliding space mechanism, which dynamically adjusts the space and permissions of the driver. Dbox allocates redundant virtual spaces for untrusted drivers. The real address space of the untrusted driver can randomly slide among redundant spaces. At any time, there is only one space (called current space) that will be mapped to real code pages through EPT. Other redundant spaces and the original space of the untrusted driver are mapped to a trap page that is non-readable, non-writable, and non-executable. Therefore, all control flows that jump to the original space can be captured due to triggering system traps. In the same way, by remapping the core kernel and other LKMs and adjusting their code permissions, all control flows that jump out of untrusted drivers can also be captured.

In practice, the control flow enters untrusted drivers through specific points (such as *syscall*). Dbox creates an isolation layer. Based on the layer, all entry_flows must pass through specific points to enter untrusted drivers. Otherwise a system trap will be triggered and the control flow will be checked according to security strategies.

In theory, legal control flows can exit the driver through direct transfer instructions, indirect transfer instructions, and return instructions. In the real world, drivers only use *call address*, *call *xx*, and *ret* to transfer control flows to other LKMs or core kernel, as shown in Table 1. In real attack scenarios (such as ROP and JOP), CRAs transfer illegal control flows to the next gadget through ICT instructions or return instructions[32]. The code snippets containing *call *pointer* or *ret* in untrusted drivers are key components to build gadget chains. To defend against CRAs, all exit_flows driven by ICT instructions and *ret* should be detected and analyzed. For legal control flows, Dbox uses a transfer site to transfer them to right locations.

Both kernel rootkits and CRAs can tamper with the control data to make it point to an absolute location. Kernel rootkits hijack the control flow that should be transferred to the kernel to an untrusted driver, while CRAs redirect the control flow to the gadgets that

Table 1: Number of control flow transfer instructions in common drivers.

type	driver	size(MB)	ret	call *pointer	call *register	call address	jmp *pointer	jmp *register	jmp address
network driver	e1000.ko	2.9	242	13	0	1514	0	0	2764
	e1000e.ko	5.8	532	7	0	5156	0	0	4743
	igb.ko	5.1	553	17	0	3136	0	0	4460
	fm10k.ko	5.4	364	18	0	1282	0	0	2280
	igc.ko	2.9	223	11	0	1170	0	0	1593
file driver	xfs.ko	29	2409	288	0	11642	0	0	16349
	ext4.ko	13	1076	171	0	6850	0	0	11445
	fat.ko	1.8	160	16	0	742	0	0	1276
	nfs.ko	10	616	129	0	2905	0	0	4212
	gfs2.ko	8.5	764	202	0	4808	0	0	6414
GPU driver	nvidia.ko	30	11301	42	0	66614	0	0	87915
	ttm.ko	2.9	235	50	0	1030	0	0	1586
	drm.ko	15	1151	70	0	4543	0	0	6360
	i915.ko	82	4554	305	0	26129	0	0	31896
	amdgpu.ko	122	7298	159	0	37410	0	0	49575

have no calling relationship with the current code snippets. Illegal control flows inevitably violate code logic or data logic. For example, a tampered function pointer no longer points to the function header, but rather points to the interior of another function, which violates data logic; the instruction *ret*, whose machine code is *c3*, is not an opcode in binary code but an operand, which violates the code logic. Based on the above designs, Dbox can detect illegal control flows and discover tampered control data by analyzing the code logic and data logic. Meanwhile, the private space can detect code probes, which will trigger the sliding space mechanism.

6 IMPLEMENTATION

6.1 Build dynamic spaces

The dynamic spaces can be used to defend against code probes. The space sliding mechanism proposed in this paper is essentially a runtime randomization method, as shown in Figure 3. The real address space of the untrusted driver can slide within multiple areas in the 64-bit kernel space. In addition, the spaces of core kernel and LKMs depended by the untrusted driver can also dynamically slide.

For the untrusted driver being loaded, Dbox maps it to a new address space after it is loaded into kernel (module->state is MODULE_STATE_LIVE) and before it is called. The kernel function *do_mmap* is used to create redundant spaces for the untrusted driver. One of the redundant spaces will be selected as an executable area. The handling steps are as follows.

First, we read all addressing items related to the untrusted driver and filter out the last item that can address the entire driver space (called *last item*). In the 64-bit space, each item in four level page tables can address 512GB, 1GB, 2MB, and 4KB memory, respectively. For example, if the driver size is 1MB, the *last item* is located in PMD (Page Middle Directory). The page table pointed to by *last item* is called *last table*. Second, we allocate private page tables for untrusted drivers. In private page tables, the *last items* corresponding to the original space, current space, and redundant space point to different *last tables*. The *last table* of the current space can address all code and data of untrusted drivers. In contrast, all items in the *last table* of redundant spaces point to the trap page. Finally, taking the virtual address's bits corresponding to the *last item* as the lower boundary, the high-order address of the redundant space can slide freely in the kernel, thereby changing the driver space.

However, the sliding spaces face two issues. One is the community Linux OSes restrict their KASLR range to 32 bits due to architectural constraints, which makes them vulnerable to even unsophisticated brute force ROP attachments due to low entropy [28]. Another issue is that the sliding spaces cannot hide return addresses on the stack, and the attacker (Vector 5) can bypass the sliding space by reading them. To address these issues, we propose a control flow redirection method, as shown in Figure 4.

To overcome the limitation of 32-bit address width in the driver code and hide return addresses, we rewrite all relative addresses related to the instruction *call* in binary code. Before the untrusted driver is loaded, the executable file will be analyzed to extract all *call xx*. After driver loading, the operand *address* of the *call address* in memory will be rewritten with a value pointing to the sliding window. The code in the sliding window rewrites the return address on the stack to make it point to an ICT instruction (*jmp *target_2*). Afterwards, another ICT instruction (*jmp *target_1*) transfers the control flow to the target function (*func1*) of the original instruction (*call addr*), which ensures the original control flow can be transferred to the right location. When the target function returns, the instruction *ret* uses the modified return address (*addr_6*) on the stack to transfer the control flow to the pre-designed ICT instruction (*jmp *target_2*). Finally, the ICT instruction transfers the control flow to the original return address (*addr2*). Unlike *call address*, the *pointer* in the *call *pointer* is not fixed. The function address (*func_2*) pointed to by the *pointer* will be modified to point to the address of the sliding window. The subsequent operation is the same as the processing method of handling *call address*. It should be noted that all the operands used by ICT instructions mentioned in the above point to a function pointer table (*target*), which stores the absolute addresses of all target functions of the instruction *call*.

Through the above operations, all real return addresses have been rewritten. Therefore, attackers cannot calculate the sliding space based on the return address. At the same time, the sliding window is unreadable, which can hide the *target* that stores real function pointers. In addition, the *target* is non-writable, which ensures function pointers cannot be tampered with. The sliding

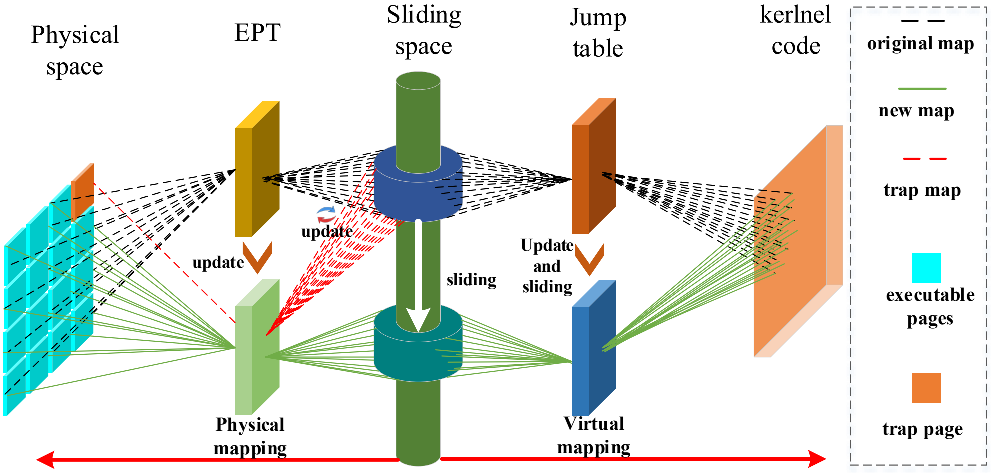

Figure 3: Dynamic space sliding mechanism

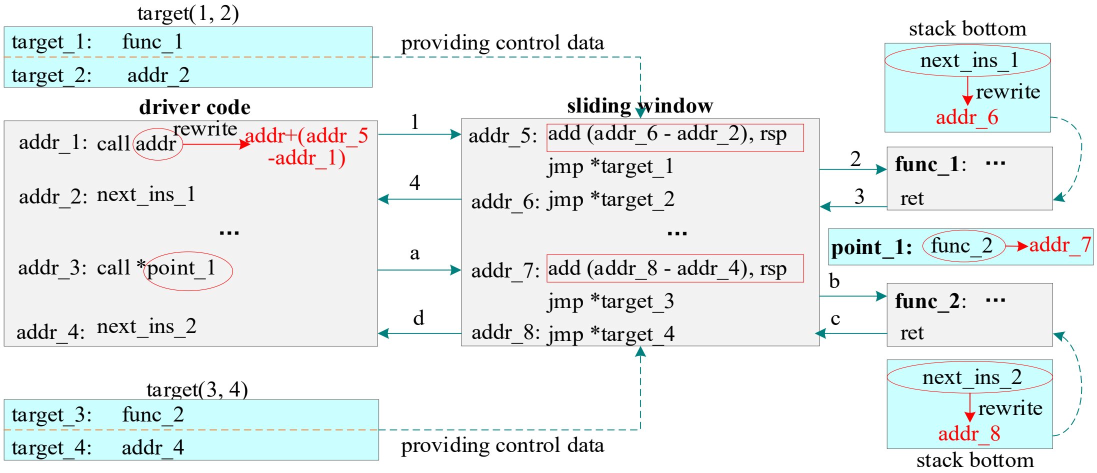

Figure 4: The redirection method of control flows

window and *target* will slide along with the driver space. Therefore, after the driver space is changed, we only need to update the function pointers in the table *target*, which ensures all control flows can be transferred to right locations.

The probe technology based on reading code (Vector 4) is a basic way to obtain code snippets that conform to gadget forms. To address this issue, Dbox uses EPT to set the code pages of untrusted drivers to be unreadable. For mixed pages, we migrate the data to a new readable page and redirect data accesses to the page. Moreover, we use VT-d to hide driver code pages, thereby preventing attackers from obtaining driver code through DMA (direct memory access). However, unreadable code can only prevent attackers from obtaining code forms, while cannot prevent attackers from probing code addresses. Because the exception thrown by reading code indicates that the pages being read are code pages. In response to this issue, we slide the driver space when capturing code reading, which can make the code address obtained by the attacker unusable.

6.2 Detect control flows

To filter out illegal entry_flows and exit_flows, we isolate the untrusted driver in a private space, as shown in Figure 5. We allocate private EPTs (D-EPT) and private page tables (D-CR3) for untrusted drivers. In the original EPT (O-EPT), all redundant spaces including the real space of the untrusted driver are mapped to a non-executable trap page, while other spaces keep their original mapping relationships. It should be noted that although the original space of the untrusted driver is mapped to real code, it is non-executable with O-EPT. So, when the entry_flow is transferred to an untrusted driver, a system trap will be triggered. In D-EPT, the original space is mapped to a non-executable trap page, while the kernel sliding space and current driver space are mapped to executable code pages. In addition, when an untrusted driver uses *call *register/pointer* to transfer control flows to other functions of the driver, it will also trigger system traps. The reason is the pointer in *register/pointer* still points to the original driver space, which is mapped to a trap page.

Based on the above designs, when ICT instructions and *ret* in the driver attempt to transfer exit_flows to the kernel, LKMs, or original

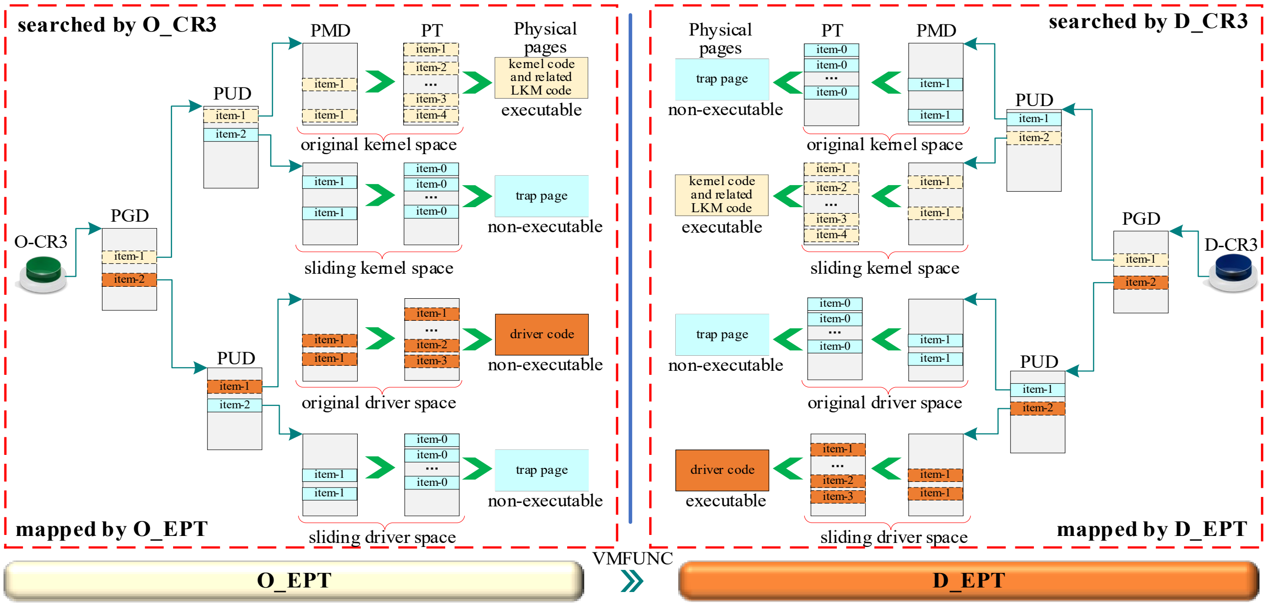

Figure 5: The Isolation Mechanism

driver space, the control flows can be captured due to system traps. In contrast, the relative instruction *call/jmp address* that cannot be used as gadgets can transfer control flows to right locations through sliding spaces without triggering any system traps. After the *call address* is executed and before the control flow returns, if the kernel/LKM uses *call *xx* to transfer control flows to the original space, a system trap will also be triggered. In this scenario, we redirect the control flow of *call address* to a specific module to switch the current page table to O-EPT and O-CR3 (see section 6.3 for the method) by fixing *address*. Meanwhile, the return address of *call address* will be replaced with a pointer pointing to a module to switch back to D-EPT and O-CR3. This design can reduce the frequency of system traps.

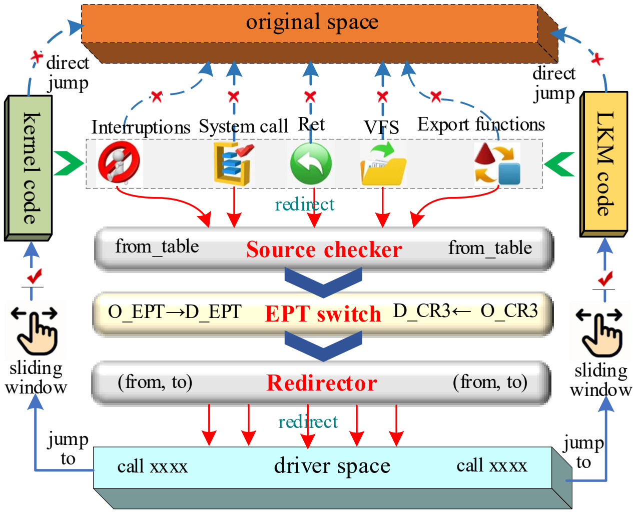

Figure 6: The detection mechanism for entry_flows

A. Detect entry_flows

Kernel rootkits (Vector 1) can hijack control flows by tampering with control data, such as system call tables and VFS pointers, and redirect them to untrusted drivers. CRAs can exploit memory vulnerabilities to redirect illegal control flows to gadgets (Vector 2) in untrusted drivers. In response to these threats, we propose a detection mechanism for entry_flows, as shown in Figure 6. Entry_flows can only enter the real driver code through *source checker*, *EPT switch*, and *redirector*. For entry_flows, the legal paths entering driver code include system calls, interrupts, exported functions, device files, and return instructions, as shown in Table 2. Entry_flows can only enter the untrusted driver through these paths, otherwise they are illegal.

When the entry_flow enters an untrusted driver for the first time, it will be transferred to the original space that has been mapped to a trap page. Therefore, a system trap will be triggered and the entry_flow can be captured. Then, the callee instruction, callee address, caller instruction, caller address, and control data will be recorded one by one, which are the basis for checking the legitimacy of control flows. To detect entry_flows, we backup the system call table (sys_call_table), interrupt vector table (IDT), interrupt descriptor (struct irq_desc), and registered driver functions (irq_desc->action->handler) during OS startup. After that, we update and backup these contents again before the untrusted driver loading. The detection method for entry_flows is shown in Algorithm 1, which adopts the following security strategies.

(1) If the caller instruction is in the kernel function *__handle_irq_event_percpu*, it indicates that the control flow is caused by an interrupt. If the original IDT has been tampered with, the control flow is illegal. If there is a function registered by a driver that no longer points to the original driver, but points to an untrusted driver, it indicates that the original function pointer has been tampered with by the untrusted driver. At this point, the control flow is also

Table 2: The information of entry_flow

events	caller instruction	caller function	control data	data type
interruption	call *pointer	__handle_irq_event_percpu	*pointer	struct irqaction
syscall	call *pointer	do_syscall_64	*pointer	sys_call_table
ret	ret	__switch_to	rsp	stack
export function	call xx	LKM functions	xx	struct kernel_symbol
VFS	call *pointer	kernel functions	*pointer	sstruct file_operations

Algorithm1: The processing method of enter_flow.

```
Input: the instruction jumping to the driver—Ins
Output: null
if Ins ∈ __handle_irq_event_percpu then
    if (check_idt(idt)==1) then
        if ∃ old_irq_desc[i]->action->handler ! =
                new_irq_desc[i]->action->handler then
            goto Error
        else goto Trans
    else goto Error
else if Ins ∈ do_syscall_64 then
    if j<syscall_num, ∃ old_sys_table[j] != new_ sys_table[j] then
        goto Error
    else goto Trans
else if Ins==ret then
    if Ins∈ __switch_to then
        goto Trans
    else goto Error
else if Ins∈ LKM functions then
    if Ins∈ module->kernel_symbol then
        goto Trans
    else goto Error
else if Ins∈ other kernel functions then
    if ∃ init_func ⊆ {cdev_add, __device_add_disk,
        _register_netdev } || (check_ops(Ins)==0) then
        goto Error
    else goto Trans
end if
Error:
    Insert_GP();
Trans:
    Transfer(Ins);
```

illegal. In contrast, legal control flows will be transferred to the untrusted driver through the control data in a new *irq_desc*.

(2) If the caller instruction is in the kernel function *do_syscall_64*, it indicates that the control flow is caused by a system call. In this scenario, the control flow can only enter the driver through a newly added system call. Otherwise, the control flow is illegal.

(3) If the instruction *ret* that triggers a system trap is in the *__switch_to*, it indicates that the kernel thread of the untrusted driver is being scheduled, and the driver space has been slid. Afterwards, we redirect the control flow to the current driver space. It should be noted that, if the driver space remains unchanged, *ret* will not trigger any system traps. In addition, the *ret* in the kernel/LKM function called by the untrusted driver does not trigger a system trap. The reason is the return address on the stack has been replaced with a pointer pointing to the sliding window. As a result, if the *ret* is not in the *__switch_to*, the control flow is illegal.

(4) If the caller instruction that triggers a system trap is in another LKM, it indicates that the control flow is driven by an exported function (recorded by *module->kernel_symbol*). Only the LKMs with dependencies on the untrusted driver are allowed to transfer control flows to the the driver space through exported functions. Otherwise, the control flow is illegal.

(5) If the caller instruction is in other kernel functions, it indicates that the control flow is caused by a device file (i.e. VFS) access. Kernel functions can utilize VFS function pointers in the data structures *file_operations*, *block_device_operations* and *net_device_ops* to enter character device drivers, block device drivers, and network device drivers, respectively. Attackers can hijack control flows by tampering with their function pointers. To prevent such attacks, we should locate the registered VFS functions and check the legitimacy of function pointers after a system trap occurs. In the driver initialization phase, we set breakpoints at the functions *cdev_add*, *usb_register_dev*, *__device_add_disk* and *_register_netdev*. Then, we can locate the registered VFS functions based on the their parameters (the data structures *cdev*, *usb_class_driver*, *gendisk*, and *net_device*). After a system trap occurs, if the caller function is not the VFS function registered by the untrusted driver, it indicates that the control flow is illegal.

For illegal control flows, we inject a regular protection exception into the current execution object to prevent its control flows from continuing to flow. For legal control flows, we transfer them to the current space of the untrusted driver. Meanwhile, the callee address, caller address, control data, and data address will be recorded, which can be used to avoid the legal control flows being detected again when they enter the untrusted driver. We propose an algorithm to transfer legal control flows, as shown in Algorithm 2.

If the legal control flow enters the untrusted driver through the instruction *call *pointer*, we rewrite the transfer target in the *pointer* with the address of a detection module. If the caller instruction is *call address*, we modify the *address* with a pointer pointing to a detection module. The detection module can check the caller instruction based on the recorded addresses and data. If the address has been recorded and the control data has not been changed, it indicates that the control flow is legal. Afterwards, the return address on the stack will be rewritten to make it point a function that can switch EPTs and page tables. Next, the original EPT will be switched to the private EPT through *vmfunc*, and the page tables will also be switched to the private page tables through *mov to cr3*. Finally, the detection module searches for the right jump target based on

Algorithm2: The transfer method of legal control flow.

```
Input: the transfer instruction—Ins; the address of Ins--α; the target of Ins in memory--τ; the return address--γ; he offset between the original space and the current space--σ;
Output: null
if Ins ⊆ {__handle_irq_event_percpu, do_syscall_64, VFS functions, LKM functions } then
    rewrite_tar(τ, check_addr)
    rewrite_rip(Ins, τ+σ)
end if
if Ins ∈ __switch_to || Ins==ret then
    rewrite_rip(Ins, τ+σ)
end if
check_addr:
    if (check_source(α, τ)==1)
        rewrite_ret(γ, switch_addr)
        switch(o_ept, d_ept, o_cr3, d_cr3)
        jmp_table(α)
    else goto Error
switch_addr:
    switch (d_ept, o_ept, d_cr3, o_cr3)
Error:
    Insert_GP();
```

the instruction address (stored in the non-writable *jmp_table*), and uses the instruction *jmp *xx* to transfer the control flow to the real space of the driver. After the driver space is changed, we only need to update the *jmp_table* that can transfer control flows to right locations. When the control flow attempts to jump to an untrusted driver using the recorded instruction *call *pointer/address* again, it will be redirected to the detection module without triggering a system trap. When the control flow returns, the EPT will switch back to the original EPT through *vmfunc*, and the page tables will also switch back to the original page tables. The above designs can avoid legal control flows repeatedly triggering system traps, thereby reducing overhead.

B. Detect exit_flows

The above designs can prevent attackers from transferring illegal entry_flows to untrusted drivers, but it cannot prevent attackers from transferring illegal exit_flows to the core kernel, other LKMs, and the untrusted driver's own code. In real attack scenarios, attackers can use dispatcher-gadgets [4] in the untrusted driver to transfer illegal exit_flows, which can be Vector-1/2/3. The instructions that can be used as dispatcher gadgets include *ret*, *call *pointer*, *call *register*, *jmp *pointer*, and *jmp *register*. These instructions in the driver may be legal instructions or non-aligned operands (such as the data *c3* used as an instruction *ret*) caused by vulnerabilities. Therefore, each exit_flow caused by the ICT instruction or *ret* needs to be captured for detection.

The scenario where the control flow jumps out of the driver can be divided into three categories. The first is the control flow returns its caller function through the instruction *ret* in an untrusted driver when the task is completed. The second is the untrusted driver transfers control flows to kernel or other LKMs through the instruction *call* (almost without involving *jmp*). The third is the CPU of the current kernel thread is preempted by another thread.

Due to the sliding space, when ICT instructions and *ret* use the original control data or probed code addresses to transfer exit_flows, a system trap will be triggered. We use the following security strategies to check the legitimacy of exit_flows.

1) If the return address used by *ret* no longer points to the sliding window (shown as Figure 4), the exit_flow is illegal. Under the sliding mechanism, the modified instruction *call* will transfer legal control flows to the sliding window. And the original return address has been rewritten to make it point to the sliding window. In contrast, CRAs tamper with the return address with a pointer pointing to the original space or probed space, rather than the sliding window. As a result, the illegal control will trigger a system trap and be detected.

2) If the caller instruction is *jmp *xx*, the exit_flow is illegal. Under normal circumstances, the driver uses *call xx* to call kernel functions or exported functions. In contrast, *jmp *xx* can transfer exit_flows to any locations.

3) If the caller instruction is *call *xx*, the exit_flow must jump to the header of an LKM exported function or a kernel function. Otherwise, it is illegal. Moreover, *call *xx* must match the opcode and operand in the executable file (*.ko*), otherwise it is illegal.

4) From the OS perspective, it needs to locate the kernel function based on the symbols in the driver during the module loading. The function addresses will be filled to the operands of all instructions *call address* or the data area of the driver. In benign drivers, all transfer targets of *call *pointer* are in driver's data area. In contrast, the transfer target of *call *register* may be in the data area or may be from the return value of the function *kallsyms_lookup_name*. From the attacker's perspective, it can only tamper with the pointers stored in writable pages rather than registers. Nevertheless, the *call *register* can still be used as a dispatcher gadget. The reason is function pointers are not always stored in registers and may be temporarily stored in writable memory (such as the stack) during parameter transfer. When the untrusted driver calls *kallsyms_lookup_name*, we record the return value in *rax*, which can be achieved by replacing the caller's return address with a recording function address. In a word, all pointers will be rewritten to make them point to the sliding window. All rewritten pointers are stored in a table, which records the pointer addresses, original pointers, and current pointers. After the system trap triggered by *call *xx* is captured, we check whether the pointer address is recorded in the table. The unchanged pointer will point to the sliding window, which can avoid triggering any system traps. If the pointer address has been recorded but the exit_flow still triggers a system trap, it indicates that the pointer stored in the original memory has been changed. Then, we can determine the current control flow is illegal.

7 EVALUATION

We conduct all experiments on a dell server equipped with 2 Xeon silver CPUs@2.4Ghz, Intel E1000E 1GbE, and 64GB Samsung 970 NVMe. The performance test is performed on an Ubuntu-21.04 with kernel 5.2, and all results are averaged after 10 runs.

7.1 Security evaluation

Vector 1. To verify Dbox's defense effect on kernel rootkits, we deploy 7 kernel rootkits (Vector 1) in different Linux OSes (such as Ubuntu 12.04 and Centos-6.10), as shown in Table 3. The results show that kernel rootkits will violate the security strategies when they hijack control flows. Compared with VTW [27], Dbox can capture and detect all entry_flows without frequent system traps, resulting in less overhead on target drivers.

Vector 2. CRAs can transfer illegal entry_flows from kernel/LKMs to untrusted drivers. And they can also transfer illegal exit_flows from untrusted drivers to kernel/LKMs (Vector 2). Due to the isolation layer, the entry_flow/exit_flow must pass through specific entries to enter/exit an untrusted driver. Otherwise, a system trap will be triggered and the illegal entry_flow/exit_flow can be captured. For example, a ROP attack initiated in kernel space will be blocked due to violating security strategy (3) when redirecting the return control flow to an untrusted driver.

Vector 3. To verify Dbox's defense effect on Vector 3, we simulate a specific ROP in the driver e1000. During e1000 running, we periodically capture the running function in the driver and manually modify the return address to make it point to the original driver space. In this experiment, all illegal exit_flows can be captured by Dbox due to violating security strategy 1). For JOP, we select the instruction *call *pointer* in the function *e1000_down* as a dispatcher gadget. Then, we modify the pointer with a random address in the original driver space. The test result shows that the exit_flow violates security strategy 4).

Vector 4. To verify Dbox's defense effect on code probes, we trigger the sliding space mechanism by reading driver code. After code reading, the spaces mapped to real code pages are shown in Table 4. The results show that the executable code of the driver will be mapped to a different space after a code probe occurs. At the same time, the original space and the probed space remain non-executable. Therefore, neither the code in the original space nor the probed code can be used as gadgets.

Vector 5. Since the return addresses have been modified to make them point to the sliding window, attackers cannot calculate the addresses of available gadgets based on them. For function pointers in data area, they generally appear as member variables in specific data structures, such as *struct file_operations*. These data structures are initialized at driver loading stage and remain unchanged throughout the driver's lifecycle. Once such function pointers are changed, system traps will be triggered when they are dereferenced, which can be captured and analyzed by Dbox. Therefore, they are difficult to be used for control flow hijacking. In addition, after the sliding window is enabled, such pointers do not point to the original functions, but to a detection module. So, even if attackers combine the known binary code with the pointers, they cannot calculate any available gadgets.

Table 3: The detection effect on kernel rootkits. P: process, F: file, M: module, N: network port.

rootkit	attack target	affected objects	violated strategy	detect?
z-rootkit	VFS	P, F, M, N	(5)	yes
enyelkm	syscall, idt	P, F	(1)(2)	yes
ivyl	VFS	P, F, M	(5)	yes
kbeast	syscall, idt	P, F, N	(1)(2)	yes
adore-ng	VFS	P, F, M, N	(2)(5)	yes
f00lkit	VFS	P, F, M	(5)	yes
syslogk	VFS	P, F, M, N	(5)	yes

In summary, Dbox has good defense against code probes, kernel rootkits, and CRAs. However, Dbox still has the potential for false positive and false negative. First, when a root user performs online debugging on a running driver, he will read the driver code. Such a legal activity will be misjudged as a code probe. Second, if an attacker can build a complete gadget chain using relative jump instructions (such as *call/jmp address*), Dbox will misjudge such an attack as a benign activity. In practice, such attacks do exist. For example, in the code snippet "*if (val==0) call func-1; else call func-2*", if the non-control data *val* is stored in the memory that can be tampered with (such as the stack), an attacker can set the calling order of *fun-1* and *fun-2* arbitrarily by tampering with *val*. Fortunately, such attacks have significant limitations, making them difficult to achieve practical attack effects. Third, Dbox cannot prevent kernel rootkits from tampering with kernel data in real-time. It can only prevent them from hijacking control flows when the tampered data is dereferenced.

7.2 Performance evaluation

Impact on the OS When Dbox is working, it will preempt the CPU, which causes overhead to the OS. To observe the impact of Dbox on OS under different CPU loads, both stress-ng and SpecCPU2006 run simultaneously. In this period, we gradually increase CPU usage using stress-ng and observe the overhead caused by Dbox. The experiment results are shown in Figure 7. All results in the figure have been normalized using the original OS as the standard. The results show that the overhead introduced by Dbox gradually increases as the CPU load increases. When the CPU usage is less than 40%, the average overhead introduced by Dbox is less than 3.6%. When the CPU is close to full load (the limit of stress-ng is 99%, not 100%), the average overhead is about 10%.

In addition to CPU overhead, Dbox has an impact on system latency and bandwidth. We use Lmbench to measure the system latency and bandwidth attenuation caused by Dbox, as shown in Figure 8. During the test, we inject different numbers of untrusted drivers with a code size of 1MB into the OS. The results show that when the number of drivers monitored by Dbox is 0, both the average system latency and average bandwidth attenuation are less than 2%. As the number of monitored drivers increases, both system latency and bandwidth attenuation will increase. When the number of monitored drivers reaches 40, the both increase to 8.6% and 7.6%, respectively.

During running, Dbox can trigger system traps, which affects the OS. System traps can be divided into unconditional traps and conditional traps. The former are triggered by specific instructions and they are inevitable. In *guest* mode, the instructions *CPUID*, *gettsec*, *invd*, *xsetbv*, and all VMX instructions except *vmfunc* will cause unconditional traps. Therefore, even if Dbox does not monitor any drivers, it incurs some overhead. The conditional traps are triggered by specific events set by Dbox. When a control flow enters/exits the untrusted driver for the first time, a system trap will be triggered. Additionally, both the control flow transfer violating

Table 4: The executable space of the driver. Pn: The executable space of the driver after the n^{th} probes.

Driver	P0		P1		P2		Physical pages	
	start	end	start	end	start	end	start	end
iptable_nat	ffffffffa0425000	ffffffffa04282d3	ffff888010000000	ffff8880100032d3	ffffc00041110000	ffffc000411132d3	b0af4000	b0af72d3
e1000e	ffffffffa0200000	ffffffffa023bf9c	ffffaff0a3010000	ffffaff0a304bf9c	ffff9110b2010000	ffff9110b204bf9c	9ecb0000	9ecebf9c
bluetooth	ffffffffa052f000	ffffffffa058d875	ffffc121d40d0000	ffffc121d40e248b	ffffaa1140000000	ffffaa114001248b	aa130000	aa14248b
iptable_filter	ffffffffa0202000	ffffffffa020520a	ffffa00031215000	ffffa0003121820a	ffffb110a0805000	ffffb110a080820a	910f0000	9101320a
kvm_intel	ffffffffa0342000	ffffffffa0364e79	ffffb102d5110000	ffffb102d5132e79	ffff9a1954f10000	ffff9a1954f32e79	c1a10000	c1a32e79
lp	ffffffffa006d000	ffffffffa007155f	ffffc00400210000	ffffc0040021455f	ffff891011032000	ffff89101103655f	a0b20000	a0b2455f

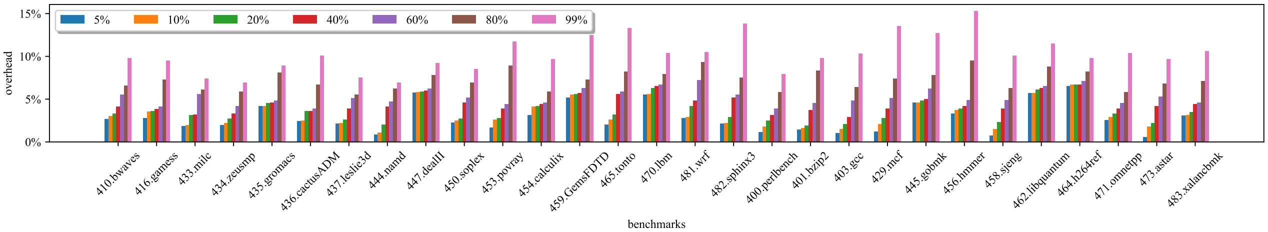

Figure 7: The test results of SpecCPU2006

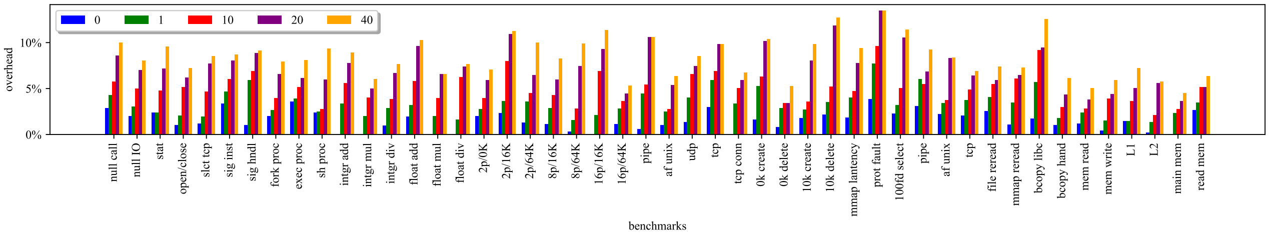

Figure 8: The test results of Lmbench

security strategies and the memory access violating EPT configuration can also trigger system traps. During handling system traps, Dbox suspends the running function to check the legitimacy of the current control flow, which affects the running speed of the target. Moreover, during the detection period, Dbox needs to occupy CPU time slices, which in turn affects the execution of other processes.

Dbox needs to allocate redundant spaces for untrusted drivers. At the same time, it also needs to add some code for control flow detection. Therefore, compared to the original driver, the untrusted driver needs to occupy more memory. To observe the memory overhead introduced by Dbox, we manually compiled and loaded some common drivers. Dbox takes them as untrusted drivers and allocates different numbers of redundant spaces for them. The memory overhead is shown in Figure 9.

The results show that the number of redundant spaces does not incur significant overhead on memory. When the number of redundant spaces is 10 and 100, Dbox increases the untrusted driver by an average of 5.6% and 5.8%, respectively. Due to the fact that redundant page tables are shared and they are mapped to a same trap page, they do not occupy too much physical pages. In contrast, EPT occupies more pages. To reduce unnecessary memory overhead, we only map the pages related to the current driver when creating private EPTs. The objects mapped by a private EPT include the untrusted driver, the core kernel, and the LKMs needed by the driver. In general, a private EPT occupies no more than 5MB memory.

Impact on the driver During the running of an untrusted driver, Dbox needs to track and detect all entry_flows and exit_flows, which can slow down the running speed of the driver. The sliding mechanism adopted by Dbox essentially is to a re-randomization method[28, 43]. We manually install NVMe and E1000e and see them as untrusted drivers. Afterwards, an LKM reads the driver

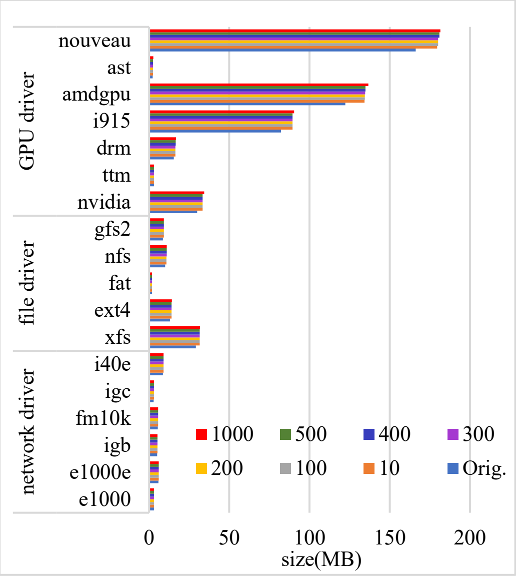

Figure 9: The memory overhead

code at regular intervals (1ms, 10ms, and 50ms) to trigger the sliding space mechanism.

When testing NVMe, we use the same method as Adelie [28] to open and read a file stored on the NVMe storage. The file is opened with O_DIRECT and O_SYNC flags, and a block size of 32 bytes is repeatedly read from the beginning of the file in a tight loop [28]. Afterwards, we set the block size to 64 bytes, 128 bytes, and 256 bytes, respectively. At the same time, we test the file reading throughput and CPU usage every 1 minute and compared them with Adelie's test results. The results are shown in Figure 10.

When testing E1000e, we run ApacheBench to test network performance. During the experiment, we test the network throughput and CPU usage at 0, 1, 5, and 10 minutes after the driver is loaded. The test results are shown in Figure 11.

In the above experiments, the I/O waiting time exceeded the CPU running time. Therefore, they cannot verify the running status of the driver under CPU constraints. Similar to Adelie [28], we also create a dummy device driver that implements a null *ioctl* operation. We repeatedly execute the syscall *ioctl* to call the driver code and measure the number of *ioctl* operations performed per second. The results are shown in Figure 12.

The above results show that Dbox incurs significant overhead in the early stage of driver running. As the driver runs, the overhead gradually decreases. The reason is that all ICT instructions will trigger system traps when they enter/exit the untrusted driver for the first time. Then, the instructions triggering system traps will be analyzed by Dbox. For the legal instructions, they will be recorded and rewritten, which can prevent them from triggering system traps when they are called again. As the driver runs, more and more legal ICT instructions are captured and recorded. Therefore, the number of system traps will decrease, which can reduce overhead gradually. However, the overhead incurred by Dbox is still more than Adelie, especially when the re-randomization interval is very small. The reason is our targets are untrusted drivers without source code instead of the open-source drivers protected by Adelie. As a result, Dbox cannot construct a control flow transfer table offline by analyzing the source code like Adelie. In addition, compared to Adelie, Dbox can detect entry_flows and exit_flows, which also increases more overhead.

Fortunately, Dbox is a event-triggered method. It does not randomize the untrusted driver periodically like Adelie. Although we periodically trigger the space sliding mechanism, the randomization frequency is not as high as that in the experiment in real execution scenarios. The current space of the untrusted driver will slide to an random location only when a code probe occurs. Moreover, it can also provide strong protection for the control flows of the running drivers, which is a capability not available in other KASLR methods. More importantly, Dbox can detect untrusted drivers without source code, which is impossible for the methods [13, 25, 26] relying on source code.

8 CONCLUSION

To isolate and detect the untrusted drivers without source code, this paper proposes a method Dbox. It isolates the untrusted drivers in a private space. Unlike existing KASLR methods, Dbox does not fix the untrusted driver in a driver space. It builds a sliding space mechanism for untrusted drivers and dynamically adjusts their code permissions. Based on our designs, all entry_flows and exit_flows can be captured and analyzed. The experiment results and analysis show that Dbox has good defense against code probes, kernel rootkits, and CRAs. In addition, as the untrusted driver runs, the impact of Dbox on the untrusted driver will gradually decrease. As a result, Dbox does not introduce too much overhead to the protected drivers.

However, Dbox still has some limitations. First, it is only effective for drivers that are loaded after the OS starts. For the drivers that start with the OS, Dbox cannot perform binary rewriting before they run. Fortunately, developers do not set untrusted drivers as the LKMs that start with the OS, let alone compile them into the kernel. Second, Dbox is only effective for the open-source Linux that is running on x86 processors equipped with VMX and EPT. Third, due to the limitation of the number of entries in the EPT list, Dbox can only detect up to 511 untrusted drivers at the same time. According to our observation, the total number of drivers does not exceed 511 in most scenarios, let alone untrusted drivers.

9 ACKNOWLEDGMENTS

This research was supported by the Fundamental Research Funds for the Central Universities (2023QN1078)

REFERENCES

[1] Ahmed M Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. 2014. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 90–102.

[2] Davi L et al Biondo A, Conti M. 2018. The Guard's Dilemma: Efficient {Code-Reuse} Attacks Against Intel {SGX}. In *Proceedings of the 27th USENIX Security Symposium*. 1213–1227.

[3] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. 2014. Hacking blind. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 227–242.

[4] Bramwell Brizendine and Austin Babcock. 2021. Pre-built JOP Chains with the JOP ROCKET: Bypassing DEP without ROP. *Black Hat Asia* (2021).

[5] Bigelow R et al Brown M D, Pruett M. 2021. Not so fast: understanding and mitigating negative impacts of compiler optimizations on code reuse gadget sets. In *Proceedings of the ACM on Programming Languages*. 1–30.

[6] Haubenwallner M et al. Canella C, Schwarz M. 2020. KASLR: Break it, fix it, repeat. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. 481–493.

[7] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. 2016. Fine-grained control-flow integrity for kernel software. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 179–194.

[8] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the Line: Practical Cache Attacks on the MMU.. In *NDSS*, Vol. 17. 26.

[9] Jinyu Gu, Xinyue Wu, Wentai Li, Nian Liu, Zeyu Mi, Yubin Xia, and Haibo Chen. 2020. Harmonizing performance and isolation in microkernels with efficient intra-kernel isolation and communication. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 401–417.

[10] Polychronakis M et al. Göktaş E, Athanasopoulos E. 2014. Size Does Matter: Why Using {Gadget-Chain} Length to Prevent {Code-Reuse} Attacks is Hard. In *Proceedings of the 23rd USENIX Security Symposium*. 417–432.

[11] C. Harini and C. Fancy. 2020. A study on the prevention mechanisms for kernel attacks. In *Artificial Intelligence Techniques for Advanced Computing Applications: Proceedings of ICACT 2020*. Springer, 11–17.

[12] Williams D Holmes B, Waterman J. 2022. KASLR in the age of MicroVMs. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 149–165.

[13] Detweiler D et al Huang Y, Narayanan V. 2022. KSplit: Automating Device Driver Isolation. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 613–631.

[14] Christopher Jelesnianski, Jinwoo Yom, Changwoo Min, and Yeongjin Jang. 2020. Mardu: Efficient and scalable code re-randomization. In *Proceedings of the 13th ACM International Systems and Storage Conference*. 49–60.

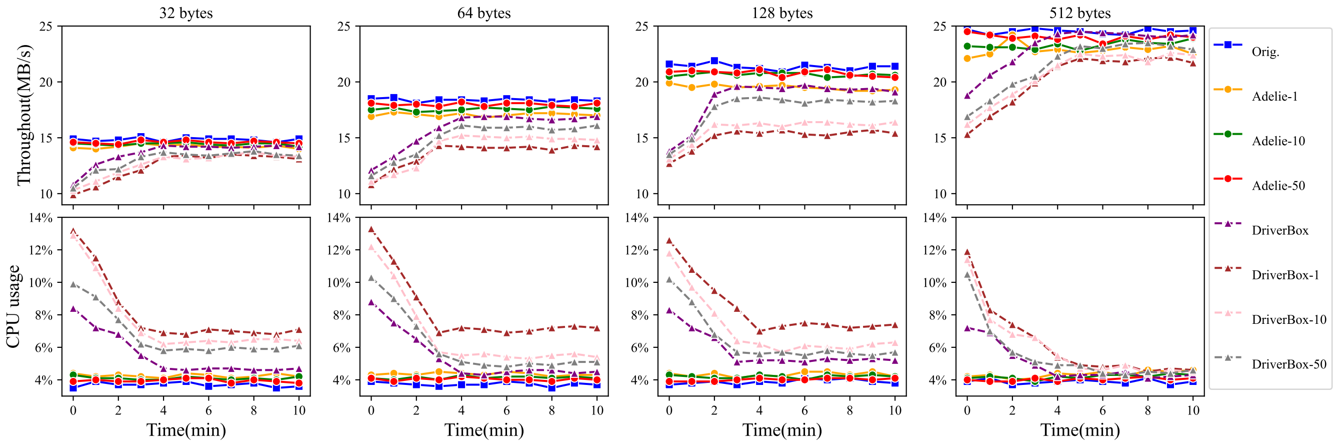

Figure 10: The throughput and CPU usage of NVMe

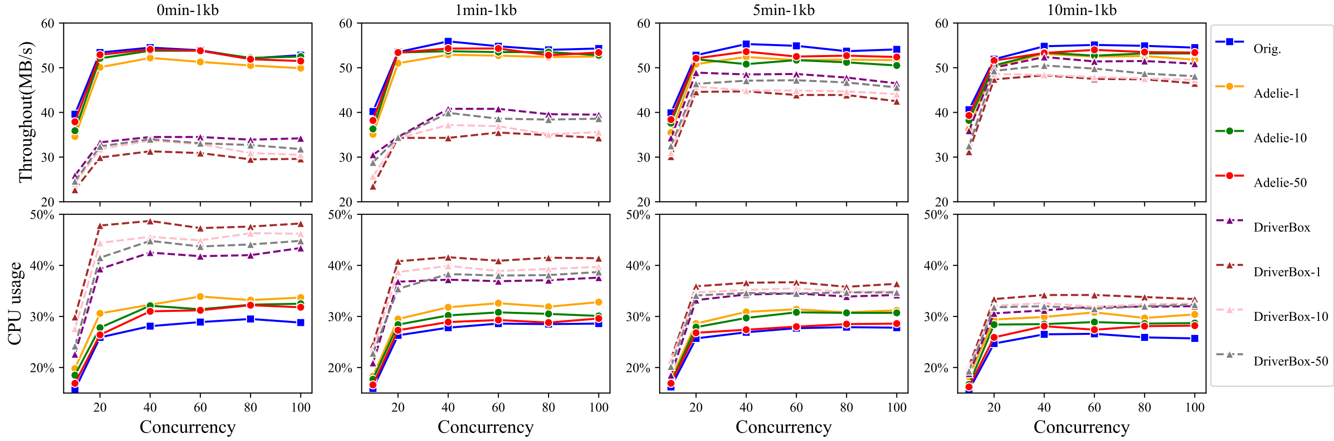

Figure 11: The throughput and CPU usage of E1000e

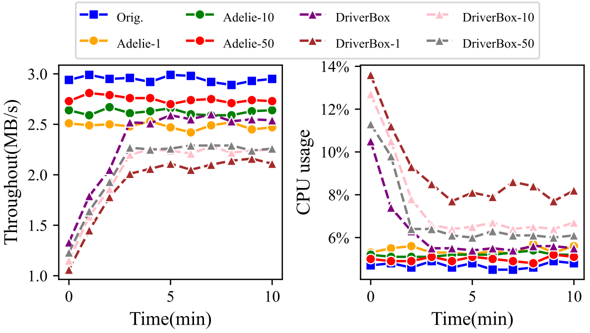

Figure 12: The throughput and CPU usage of ioctl

[15] Groß S et al. Lekies S, Kotowicz K. 2017. Code-reuse attacks for the web: Breaking cross-site scripting mitigations via script gadgets. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1709–1723.

[16] Jinku Li, Xiaomeng Tong, Fengwei Zhang, and Jianfeng Ma. 2018. Fine-cfi: fine-grained control-flow integrity for operating system kernels. *IEEE Transactions on Information Forensics and Security* 13, 6 (2018), 1535–1550.

[17] YongGang Li, Yeh-Ching Chung, Jinbiao Xing, Yu Bao, and Guoyuan Lin. 2022. MProbe: Make the code probing meaningless. In *Proceedings of the 38th Annual Computer Security Applications Conference*. 214–226.

[18] YongGang Li, GuoYuan Lin, Yeh-Ching Chung, YaoWen Ma, Yi Lu, and Yu Bao. 2023. MagBox: Keep the risk functions running safely in a magic box. *Future Generation Computer Systems* 140 (2023), 282–298.

[19] Yong-Gang Li, Yeh-Ching Chung, Kai Hwang, and Yue-Jin Li. 2020. Virtual wall: Filtering rootkit attacks to protect linux kernel functions. *IEEE Trans. Comput.* 70, 10 (2020), 1640–1653.

[20] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. 2015. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 1607–1619.

[21] Kangjie Lu, Wenke Lee, Stefan Nürnberger, and Michael Backes. 2016. How to Make ASLR Win the Clone Wars: Runtime Re-Randomization.. In *NDSS*.

[22] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M Frans Kaashoek. 2011. Software fault isolation with API integrity and multi-principal modules. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 115–128.

[23] Alyssa Milburn, Erik Van Der Kouwe, and Cristiano Giuffrida. 2022. Mitigating information leakage vulnerabilities with type-based data isolation. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1049–1065.

[24] Wei-Loon Mow, Shih-Kun Huang, and Hsu-Chun Hsiao. 2022. LAEG: Leak-based AEG using Dynamic Binary Analysis to Defeat ASLR. In *2022 IEEE Conference on Dependable and Secure Computing (DSC)*. IEEE, 1–8.

[25] Jacobsen C et al Narayanan V, Balasubramanian A. 2019. LXDs: Towards isolation of kernel subsystems. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 269–284.

[26] Tan G et al Narayanan V, Huang Y. 2020. Lightweight kernel isolation with virtualization and VM functions. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS international conference on virtual execution environments*. 157–171.

[27] Matthias Neugschwandtner, Alessandro Sorniotti, and Anil Kurmus. 2019. Memory categorization: Separating attacker-controlled data. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19–20, 2019, Proceedings 16*. Springer, 263–287.

[28] Ruslan Nikolaev, Hassan Nadeem, Cathlyn Stone, and Binoy Ravindran. 2022. Adelie: continuous address space layout re-randomization for Linux drivers. In

Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. 483–498.

[29] Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, and Cristiano Giuffrida. 2016. Poking holes in information hiding. In *25th USENIX Security Symposium (USENIX Security 16)*. 121–138.

[30] et al Pomonis, Marios. 2017. kRX̂: Comprehensive kernel protection against just-in-time code reuse. In *Proceedings of the Twelfth European Conference on Computer Systems*. 420–436.

[31] Xinhui Shao, Lan Luo, Zhen Ling, Huaiyu Yan, Yumeng Wei, and Xinwen Fu. 2022. fASLR: Function-based ASLR for resource-constrained IoT systems. In *European Symposium on Research in Computer Security*. Springer, 531–548.

[32] et al Shrivastava, Rajesh Kumar. 2022. Securing Internet of Things devices against code tampering attacks using Return Oriented Programming. *Computer Communications* 193 (2022), 38–46.

[33] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *2013 IEEE symposium on security and privacy*. IEEE, 574–588.

[34] Abhinav Srivastava and Jonathon T Giffin. 2011. Efficient Monitoring of Untrusted Kernel-Mode Execution.. In *NDSS*. Citeseer.

[35] Göktas E. et al Van Der Veen, V. 2016. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. 934–953.

[36] Fernando Vano-Garcia and Hector Marco-Gisbert. 2020. KASLR-MT: Kernel address space layout randomization for multi-tenant cloud systems. *J. Parallel and Distrib. Comput.* 137 (2020), 77–90.

[37] Wenhao Wang, Guangyu Hu, Xiaolin Xu, and Jiliang Zhang. 2021. CRAlert: Hardware-assisted code reuse attack detection. *IEEE Transactions on Circuits and Systems II: Express Briefs* 69, 3 (2021), 1607–1611.

[38] Zhe Wang, Chenggang Wu, Jianjun Li, Yuanming Lai, Xiangyu Zhang, Wei-Chung Hsu, and Yueqiang Cheng. 2017. Reranz: A light-weight virtual machine to mitigate memory disclosure attacks. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 143–156.

[39] Cui W et al Wang Z, Jiang X. 2009. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the 16th ACM conference on Computer and communications security*. 545–554.

[40] Mengfei Xie, Yan Lin, Chenke Luo, Guojun Peng, and Jianming Fu. 2022. PointerScope: Understanding Pointer Patching for Code Randomization. *IEEE Transactions on Dependable and Secure Computing* (2022).

[41] Yutian Yang, Songbo Zhu, Wenbo Shen, Yajin Zhou, Jiadong Sun, and Kui Ren. 2019. ARM pointer authentication based forward-edge and backward-edge control flow integrity for kernels. *arXiv preprint arXiv:1912.10666* (2019).

[42] Sungbae Yoo, Jinbum Park, Seolheui Kim, Yeji Kim, and Taesoo Kim. 2022. {In-Kernel}{Control-Flow} Integrity on Commodity {OSes} using {ARM} Pointer Authentication. In *31st USENIX Security Symposium (USENIX Security 22)*. 89–106.

[43] Changwei Zou, Xudong Wang, Yaoqing Gao, and Jingling Xue. 2022. Buddy stacks: Protecting return addresses with efficient thread-local storage and runtime re-randomization. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 2 (2022), 1–37.