Page-Oriented Programming: Subverting Control-Flow Integrity of Commodity Operating System Kernels with Non-Writable Code Pages

Seunghun Han, The Affiliated Institute of ETRI, Chungnam National University; Seong-Joong Kim, Wook Shin, and Byung Joon Kim, The Affiliated Institute of ETRI; Jae-Cheol Ryou, Chungnam National University

https://www.usenix.org/conference/usenixsecurity24/presentation/han-seunghun
Page-Oriented Programming: Subverting Control-Flow Integrity of Commodity Operating System Kernels with Non-Writable Code Pages

Seunghun Han1,2 Seong-Joong Kim1 Wook Shin1 Byung Joon Kim1 Jae-Cheol Ryou2
1The Affiliated Institute of ETRI 2Chungnam National University
{hanseunghun, sungjungk, wshin, bjkim}@nsr.re.kr jcryou@cnu.ac.kr

Abstract
This paper presents a novel attack technique called page-oriented programming, which reuses existing code gadgets by remapping physical pages to the virtual address space of a program at runtime. The page remapping vulnerabilities may lead to data breaches or may damage kernel integrity. Therefore, manufacturers have recently released products equipped with hardware-assisted guest kernel integrity enforcement. This paper extends the notion of the page remapping attack to another type of code-reuse attack, which can not only be used for altering or sniffing kernel data but also for building and executing malicious code at runtime. We demonstrate the effectiveness of this attack on state-of-the-art hardware and software, where control-flow integrity policies are enforced, thus highlighting its capability to render most legacy systems vulnerable.

1 Introduction
Code-reuse attack (CRA) is a category of modern exploit techniques where attackers hijack control flows of legitimate software and transfer the control to existing code snippets, known as gadgets, to utilize them for malicious purposes. Techniques such as return-into-libc [60], return-oriented programming (ROP) [8, 19], and jump-oriented programming (JOP) [3, 15] fall into this category.

A notable series of studies have been conducted on control-flow integrity (CFI) enforcement to prevent control-flow hijacking attacks. The possible execution flow of a program can be approximated using its source or binary code prior to execution. The result can be represented as a control-flow graph (CFG), where a block of code corresponds to a node, and a flow path between nodes is represented as an edge. At runtime, the CFG is used to constrain the flow of control along predetermined forward and backward edges throughout the nodes. By invalidating arbitrary changes in control transfers, CRAs that are enabled by corrupting indirect branch targets of indirect calls, jumps, and returns can be eliminated. A significant number of studies have followed the seminal work [1, 2] on CFI enforcement, and some of them have been adopted to the toolsets used in actual practice [17, 27, 28, 50, 71]. The downside of the CFI policy enforcement is that its performance overhead increases as the number of indirect branches increases.

On the one hand, certain studies [14, 63, 78, 79] have proposed relaxed CFI enforcement policies to improve performance or compatibility. However, these policies have faced vulnerabilities that bypass their lenient enforcement [10, 21, 30, 31]. On the other hand, another group of studies has aimed to improve the precision of CFI enforcement. This has been achieved through techniques such as combining CFG generation with a compilation toolchain [18, 29, 61, 72] or supplementing dynamic data obtained at runtime [22, 35, 43, 44, 62, 73]. Their efforts narrowed down the size of a possible target set of an indirect branch (referred to as an equivalence class) [1, 2].

The CFI enforcement techniques currently in use, such as Microsoft Control Flow Guard (CFG) [50], PaX Reuse Attack Protector (RAP) [71], GNU Compiler Collection (GCC) CFI [28], Clang/LLVM CFI [17], and fine-grain CFI enforcement with indirect branch tracking (FineIBT) [27], employ static analysis during the compilation process to produce CFI policy-enforced binaries. These techniques incorporate the aforementioned studies with an emphasis on practicality and deployability. Some of them have even adopted hardware-based control-flow enforcement mechanisms [4, 38].

A critical requirement shared by most CFI enforcement research is non-writable code, which implies that code memory should be immutable at runtime. When CFI is enforced at the user level, the immutability can be maintained as long as a program itself does not modify access rights to its code memory, and the kernel sets the page table attributes to read-only. Unfortunately, this implies that CFI at the kernel level is neutralized with page table modification if a vulnerability in the kernel allows an attacker to read and write arbitrary data to memory [23, 45]. To ensure code immutability, hypervisor-level authority can be utilized [33, 66, 68], and the practicality of this approach has been showcased by Microsoft
In this paper, we present a novel form of data-only attack (DOA) called page-oriented programming (POP). POP enables an attacker to create an arbitrary control flow by executing a page-level CRA. It leverages direct branches in non-writable code memory and remaps desired gadgets in pages to the branch targets by modifying page tables. The state-of-the-art CFI enforcement focuses solely on ensuring the safe transfer of control through indirect branches. Meanwhile, in cases where hypervisor-based write protection is in place, an additional address translation layer is introduced by the hypervisor. This layer maps the OS’s physical addresses (i.e., guest physical addresses) to their corresponding physical addresses (i.e., host physical addresses) with read-write-execute permissions. However, this mechanism does not delve into mappings of logical addresses to guest physical addresses within the OS. POP exploits these blind spots by remapping the virtual address of the direct branch target to the physical address of the gadget within the kernel. Consequently, POP circumvents current CFI implementations in a novel way.

Our POP involves three steps: page carving, page stitching, and page flushing. First, an attacker creates a list of gadgets and system calls that are carved out of the kernel binary. Second, the attacker creates a new control flow to a security-sensitive function by chaining these gadgets together. The physical pages containing the gadgets are then stitched together, along with the created control flow, after modifying the page tables. Finally, the attacker may need to flush stale information from the translation lookaside buffer (TLB) of the CPU after altering the page tables to ensure that the new mappings between the virtual and physical addresses are functional.

We explain the concept of POP and illustrate its three steps. We first transpose a real-world vulnerability into an up-to-date system that offers hypervisor-based kernel integrity protection, featuring both hardware and software-based CFI mechanisms. Subsequently, we present a proof-of-concept exploit that allows a user-level application to establish a new control flow leading to a security-sensitive function, ultimately achieving privilege escalation despite state-of-the-art CFI protections. This demonstration highlights the feasibility and effectiveness of the POP scheme.

2 Background

Numerous studies have been conducted on CFI thus far, resulting in a diverse range of research outcomes. These outcomes can be broadly categorized into academic achievements and practical applications.

2.1 Control-Flow Integrity

Academic Achievements. CFI studies aim to obtain more detailed information by implementing it within compiler toolchains and creating a more precise CFG. Modular CFI (MCFI) [61] utilizes a function label per unique function signature to restrict indirect branches and support module linking. Kernel CFI (kCFI) [29] enhances CFG precision by incorporating jump tables and restricted pointer indexing. PityPat [22] and µCFI [35] use dynamic instrumentation using Intel Processor Trace (PT) [38] to obtain more precise indirect branch targets. CFI with look back (CFI-LB) [44] and origin-sensitive CFI (OS-CFI) [43] add call-site information of indirect branches. Notably, OS-CFI utilizes additional data on function pointer origin, resulting in the reduced size of the equivalence class.

This series of trials exhibited two limitations. One limitation was the performance overhead. At runtime, CFI introduces the additional burden of testing whether each indirect branch is directed toward a predetermined set of nodes (Figure 1). While Abadi et al. [1, 2] excluded direct branches from consideration, assuming that the destinations of direct branches (the dotted arrow in Figure 1) are not easily replaceable, their work still resulted in an execution overhead of up to 45%. Recent research [43] has made progress in reducing the overhead, but it is still over 14% at the NGINX benchmark. The other limitation is the incompatibility with legacy systems. The aforementioned CFI studies require source code for precise CFG generation during the pre-analysis process. Some of them [22, 35] even rely on traces of dynamic instrumentation obtained from the CPU at runtime. However, legacy commercial-off-the-shelf (COTS) binaries and CPUs are unable to meet the requirements of these studies.

CFI for binary executables (Bin-CFI) [79] and compact CFI and randomization (CCFIR) [78] enforce CFI on COTS binaries by analyzing binary executables instead of relying on
source code. These approaches enforce relatively relaxed policies, resulting in lower overhead as compared to the CFI techniques based on the precise CFG. Conversely, the kBouncer [63] and ROPecker [14] techniques aimed to detect ROP patterns at runtime by leveraging a hardware-provided feature, that is the Last Branch Record (LBR) of Intel, which stores recently executed branch history. These approaches loosen the restrictions in policy enforcement to improve performance; however, this trade-off can result in attacks that bypass their CFI policies [9, 10, 21, 30, 31].

**Practical Applications.** Among the various CFI implementations, the most notable ones have already been adopted into the two major commodity OSes: Microsoft Windows and Linux. Table 1 summarizes the CFI implementations that are currently used.

In Microsoft Windows, a software-based forward-edge protection called CFG is developed, which can be optionally equipped with Windows 10 and 11 series. Moreover, a hardware-based backward-edge protection mechanism called shadow stack is employed. This mechanism is built on Control-flow Enforcement Technology (CET) that has been commercialized by Intel.

In Linux, several major compilers support CFI protections. First, RAP provides type-based indirect branch protection. Additional backward-edge protection can be achieved through return address duplication and encryption. Second, GCC supports the indirect branch tracking and shadow stack features of CET for CFI protection. Third, Clang/LLVM offers its own type-based forward-edge protection and software-based shadow call stack for backward-edge protection. Recently Clang/LLVM has also adopted CET. Starting from November 2022, the Linux kernel has enabled a hybrid CFI protection scheme, called FiniBT, which ensures the accuracy of the forward indirect branch by validating the hashes of functions and using CET. In the case of FiniBT, the hash validation is performed by callees, whereas in other schemes, it is performed by the callers. Notably, the hardware-based backward-edge protection, shadow stack, is only applicable to the userspace in Linux, while the other CFI implementations can be applied to both user applications and the kernel.

### 2.2 Code Immutability and Protection Mechanisms

Code region must be preserved as non-writable at runtime since CFI policies cannot be enforced on dynamically generated or self-modified code [1, 2]. The responsibility for ensuring the immutability of user- and kernel-level code memory has been given to the kernel.

**Data-Only Attack.** It is an attack technique that manipulates non-control data [34, 36, 39, 70] without violating CFI policies. Data-only attacks (DOAs) are effective when the kernel or drivers have memory access vulnerabilities [55, 56]. These vulnerabilities would allow an attacker to break the kernel address space layout randomization (KASLR) and gain escalated privileges by modifying credentials. Moreover, if the vulnerability enables reading and writing arbitrary memory [53, 54], the attacker could manipulate both user- and kernel-level code. This manipulation can occur even when CFI enforcement is in place by altering page tables that are supposed to be protected by the kernel [20, 23, 45].

**Hypervisor-Based Kernel Integrity Protection.** Recent hardware-based virtualization processes support hypervisors in enforcing strong security policies. Running at a higher privilege than the kernel, the hypervisor utilizes second-level address translation (SLAT), which allows the hypervisor to set page permissions separately from the kernel. Moreover, Mode-Based Execution Control (MBEC) from Intel [38] and Guest Mode Execute Trap (GMET) commercialized by AMD [4] impose further restrictions on code page execution, which is determined by the execution modes designated as either user-mode or supervisor-mode. These features cooperate with the SLAT and enable sophisticated page permissions. Consequently, modern operating systems employ the hypervisor as

<table>
<thead>
<tr>
<th>CFI Implementation</th>
<th>Commodity OS</th>
<th>Forward Edge Policy</th>
<th>Backward Edge Policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft CFG [50] with CET [4, 38]</td>
<td>Windows</td>
<td>Bitmap-based verification</td>
<td>Hardware-based shadow stack</td>
</tr>
<tr>
<td>PaX RAP [71] (open-source version)</td>
<td>Linux</td>
<td>Type-based verification</td>
<td>Type-based verification</td>
</tr>
<tr>
<td>GCC CFI [28] (only CET)</td>
<td>Linux</td>
<td>Hardware-based indirect branch tracking</td>
<td>Hardware-based shadow stack</td>
</tr>
<tr>
<td>Clang/LLVM CFI [17] with CET</td>
<td>Linux, Windows</td>
<td>Type-based verification with hardware-based indirect branch tracking</td>
<td>Hardware-based shadow stack or shadow call stack (only for AArch64) (^1)</td>
</tr>
<tr>
<td>FiniBT [27] (embedded with CET)</td>
<td>Linux</td>
<td>Type-based verification(^2) with hardware-based indirect branch tracking</td>
<td>Hardware-based shadow stack or shadow call stack (only for AArch64)</td>
</tr>
</tbody>
</table>

Table 1: Control-flow integrity implementations currently in use and their policies for commodity OSes

\(^1\)AArch64 of ARM only supports the feature. It was integrated for the x86_64 system but later removed owing to performance overhead and race condition problems [48].

\(^2\)PaX RAP and Clang/LLVM CFI employ a caller-side verification policy, whereas FineIBT employs a callee-side verification policy.
a monitoring and controlling entity with elevated privileges. While the 12th generation CPU products from Intel provide Hypervisor-managed Linear Address Translation (HLAT) to ensure page table integrity for the kernel, SLAT with MBEC or GMET is a widely adopted feature in modern CPUs. Further discussion on this topic is presented in Section 7.

Secvisor [68] and NICKLE [66] can maintain kernel code integrity with hypervisor-level authority, and their implementations in Linux have been demonstrated. The current Microsoft Windows series include a feature called Hypervisor-Protected Code Integrity [52], which provides virtualization-based memory integrity for kernel code. These hypervisor-based mechanisms utilize SLAT to translate the physical addresses used in the guest OS to the physical addresses in the host OS. During this second-level translation process, the CPU ensures that the access to physical pages adheres to the defined page attributes. For instance, if a process in the guest OS attempts to write to an address on a page that is marked as executable only and not writable, an exception occurs during the SLAT process. The hypervisor can interpret this as a code modification attack. Similarly, if an exception occurs when data is written to read-only credentials, the hypervisor may consider it as a credential modification attack. Furthermore, if an exception is raised while executing code in kernel mode and the code is located in a code page that lacks the supervisor-execute permission of MBEC and GMET, the hypervisor may identify it as a control-flow hijack or an unauthorized kernel code injection. For the purpose of data protection, Credential Guard [51] from Microsoft utilizes a virtualized enclave to store security-sensitive data such as credentials. A framework called PrivWatcher [41] was also proposed, which includes a hypervisor-based read-only safe area where credentials can be securely stored, and its prototype was implemented in Linux.

The CFI enforcements in commodity OSes have evolved to incorporate stronger countermeasures against traditional CRAs and page table attacks by leveraging hardware assists and the notion of hypervisors [17, 27, 50, 52, 66, 68].

3 Assumption and Threat Model

We assume our target system is fortified with cutting-edge CFI protection mechanisms. Namely, hardware-assisted CFI policies and hypervisor-based kernel integrity protection prevent control-flow hijacking techniques like ROP and JOP. Additionally, the page-level write protection [33, 52, 66, 68] and read-only safe area scheme [41, 51] of the protection mechanisms thwart unauthorized code alterations, code injection, and unauthorized modifications to kernel credentials. Thus, an attacker has to invoke legitimate kernel functions to perform malicious behaviors such as privilege escalation and spawning a root shell.

We have the same set of assumptions regarding system vulnerabilities and the capabilities of potential attackers as in prior studies [6, 20, 65, 68, 71]. Specifically, the system has an arbitrary kernel memory read and write vulnerability. By exploiting it, an attacker can manipulate page tables, disable KASLR, and manipulate the kernel of the system. The attacker, possessing local user privileges as in previous studies [20, 23, 45], can also execute system calls and programs, thereby collecting system information such as the kernel binary, the size of system memory, and the list of kernel symbol names. Leveraging these capabilities, the attacker may attempt typical control-flow hijacking techniques and kernel modification. However, these trials must be hindered by the presence of CFI policies and the hypervisor running on the system.

4 Motivation

Contemporary CFI approaches are fortified by the kernel CFI policies, which control indirect branches, and by the hypervisor, which ensures the sanity of the kernel during the second-level address translation process. In other words, direct branches and page-mapping information inside the OS have been out of the spotlight. Our study revisits DOAs against page tables, shedding light on the blind spots.

Kernel Control-Flow Integrity Policies. Most of the CFI-related studies, including hardware-assisted kernel CFI, have primarily concentrated on validating indirect branches, overlooking the direct branch targets. This oversight is attributed to the assumption that the targets reside in read-only code memory. However, if the immutability of code memory cannot be guaranteed, the effectiveness of most CFI implementations must be reassessed.

Non-Writable Code Mechanism. As shown in Figure 2a, the kernel code is protected by SLAT. A normal function, kern_normal(), resides at the guest logical address (GLA) 0xffff1f100, while a security-sensitive function, kern_sensitive(), is located at GLA 0xffff21100. Based on the page table, their corresponding guest physical addresses (GPAs) are translated as 0xffff01000 and 0xffff02000, respectively. These GPAs are then translated again by the hypervisor via SLAT, resulting in the host physical addresses (HPAs) of 0x00001000 and 0x00002000, respectively.

Let us consider a scenario where an attacker attempts to modify the code of kern_normal() and successfully manipulates an attribute of the page in the page table, as shown in the upper-left corner of Figure 2b. However, because the corresponding page in the SLAT table still does not grant write permission, an exception occurs, leading to termination. In the case of page injection, as depicted in the lower-left corner of Figure 2b, an arbitrarily injected page cannot have a corresponding page in the SLAT table or be granted execute permission without authorization from the hypervisor. Consequently, this situation triggers an exception.

The “RWX” access control during SLAT can offer write-protection for code pages; however, it is unable to ensure the integrity of the kernel code. Figure 2c illustrates a sce-
Figure 2: Effectiveness and weakness of the hypervisor-based non-writable code mechanism

(a) Design of the hypervisor-based non-writable code mechanism

(b) Prevention of unauthorized code modification and injection

(c) Weakness of the hypervisor-based non-writable code mechanism

* Read, write, and execute permissions of pages are indicated in page tables and SLAT tables as R, W, and X

...
0xff102400 and 0x00102400, respectively, and their page offset is 0x400. This implies that any gadget with the same page offset can replace the function. Consequently, the attacker can create a new control flow by replacing the page with a call gadget located at the physical address 0x00402400, as shown in the lower-left part of Figure 3.

The call gadgets that were previously gathered are divided into two types: direct and indirect call gadgets. In particular, when using direct call gadgets, additional page remappings are required according to the target logical address. As illustrated in the lower-left part of Figure 3, let us assume that a new control flow has been created over 0xff301200 using the direct call gadget; then, the sensitive function and related data, kern_sensitive() and current_task_cred, respectively, must be remapped to 0xff301200 and 0xff302900, respectively. In the case of indirect call gadgets, these page remappings are not necessary. If a target function is non-essential, such as sub_sensitive(), it can be replaced with a NOP gadget whose physical page is 0x00601600 and is unlinked with the fallback. Furthermore, as shown in the lower-left corner of Figure 3, if arbitrary data, such as modified_cred, are required, the attacker can forcibly inject it into the page table for subsequent exploitation. The attacker can then pass the data to the target system call as an argument.

Page Flushing. POP specifically focuses on remapping page tables in the kernel and does not modify the TLB cache entries that are accessed when invoking a system call. If the original and remapped pages are mixed in the TLB, the exploitation may fail. To this end, POP must be able to flush the TLB entries by executing or accessing kernel pages that trigger the flushing algorithm of the TLB (❸).

At the end of POP, as shown in the upper-middle part of Figure 3, the attacker finally proceeds to invoke a system call.
call with arbitrary arguments that will eventually be chained to the target function, that is, kern_sensitive(), in the kernel (❹). Using the direct call gadget, the attacker passes the first argument, arg1, which specifies the address of modified_cred (0xff401900) holding root credentials via sys_normal() (❹ -a). Otherwise, for the indirect call gadget, the attacker should pass the argument containing the address of kern_sensitive() (0xff201200) along with arg1 through sys_normal() (❹ -b). Consequently, current_task_cred, shown in the lower-right of the figure, is updated, and the attacker gains the administrator privilege, known as root.

Next, we describe the details of our three-staged POP technique, which aims to subvert state-of-the-art CFI protection mechanisms.

### 5.2 Page Carving

Gadgets for POP are categorized into two types based on their purpose: call gadgets (Figure 4a), which chain the system call candidates to the security-sensitive functions, and NOP gadgets (Figure 4b), which unlink non-essential functions if required. Moreover, based on their forms, these gadgets can be classified into function gadgets, which utilize the entire functions, and ROP-like partial gadgets, which only leverage specific code pieces as required. Note that call gadgets with indirect branches are considered partial gadgets because the CFI policy enforces indirect branches within functions.

The page carving step is a process that extracts the candidates for call gadgets, NOP gadgets, and system calls from kernel binaries and creates a list. The kernel binaries are disassembled and converted into assembly code. Function gadgets are then extracted from this result, encompassing all functions within the kernel. Partial gadgets are obtained by examining each page offset, disassembling the code pages of the kernel binary from page offset 0 to 4095, and converting them into assembly code. Owing to disassembling of arbitrary instruction sequences, some of the partial gadgets have invalid instructions. Therefore, they are removed from the list. System call candidates are also identified from the assembly code by utilizing system call-related information, such as the system call table and symbol names. They are included in the list as well. System call candidates can also be considered part of the gadget if they contain direct branches.

When identifying gadgets and system call candidates, we considered the two strategies for exploitation flows, as illustrated in Figure 5: (i) round trip, where the control flow executes a security-sensitive function and then returns to the caller; and (ii) one way, where the control flow executes the security-sensitive function without the need to return.

In the round-trip strategy (Figure 5a), an attacker regains control after exploiting the system via POP. In this case, the attacker can gain escalated privilege and perform additional procedures, such as modifying system files and gaining a root shell within the malicious application, by returning to the exploitation point. In the round-trip strategy, the pair of call and return instructions are required when chaining gadgets.

In the one-way strategy (Figure 5b), the attacker performs similar operations as in the round-trip approach, except that the security-sensitive function must execute attacker-targeted applications, including the malicious application with escalated privilege. Unlike the former, the control flow does not return to the exploitation point. Therefore, the pair of call and return instructions need not be considered.

### 5.3 Page Stitching

The page stitching step, a core component of POP, chains the previously collected gadgets and system call candidates. This step identifies new control flows capable of reaching a security-sensitive function and employs arbitrary memory read and write vulnerabilities to remap the page tables. The step consists of the following four phases: chaining gadgets,
Figure 6: Different methods of chaining gadgets. POP gadgets are shown in gray boxes.

Phase 1 - Chaining Gadgets. This phase involves chaining gadgets that link the system call candidate to the security-sensitive function along with arguments. As shown in Figure 6, we define three types of gadget chains: (i) direct, (ii) indirect, and (iii) direct to indirect calls. These gadget chains cater to both round-trip and one-way exploitation strategies, as discussed in Section 5.2.

(i) Direct call chaining (Figure 6a) consists of call and jump instructions. The gadget can be directly chained to a security-sensitive function if the page offset of the direct branch in it matches the page offset of the sensitive function. Otherwise, the gadgets need to be chained by combining other call gadgets until the chain reaches the sensitive function. Ideally, the system call candidate can also reach the sensitive function if their page offsets are identical, resulting in the shortest gadget chain in POP.

(ii) Indirect call chaining (Figure 6b) typically links only with an indirect branch. POP creates a new control flow by chaining a single indirect call gadget, as it can invoke an arbitrary function allowed by the CFI policy.

(iii) Direct to indirect call chaining (Figure 6c) combines both direct and indirect call chains. It is used if the aforementioned chains are not affordable, although it is a longer path to the sensitive function.

Direct call chaining requires the remapping of additional pages. When the logical address of the security-sensitive function is modified, the physical pages containing subfunctions and data within the sensitive function must be remapped accordingly. If the gadgets used for page stitching involve subfunctions and data, they must also be remapped correspondingly. The remapping process varies depending on the addressing mode employed. For example, we need to identify the base address of the segment or address of the instruction to calculate the exact value when segment- or instruction pointer-based addressing mode is used, respectively. The addressing modes are detailed in Section 6.1.

When exploiting the security-sensitive function with data arguments, the size of a logical address has to be considered along with the gadget chain and system call candidate. Let us assume that the system call candidate is invoked with a 4-byte argument, which contains the logical address of the malicious credentials to be passed to the sensitive function. In this case, the upper 4 bytes of the argument will be trimmed, and only the lower 4 bytes will be delivered to the sensitive function, even though an 8-byte argument is passed through the system call. Consequently, the physical page of the malicious credentials needs to be remapped to an address within the 4 GB address space, and the 4-byte argument is passed to the sensitive function.

Phase 2 - Avoiding the Remapping Explosion. For reliable exploitation, page remappings should be minimized, which causes code changes in the logical address space. This can be achieved by prioritizing the selection of gadgets with indirect calls and shorter lengths, without subfunctions and data references, and those on a single physical page. However, if an exploitation method other than direct call chaining is not applicable, hundreds of physical pages may need to be remapped depending on the structure of a security-sensitive function. The importance of subfunctions and data must be determined to prevent this remapping explosion, and non-essential ones must be remapped using NOP gadgets to unlink function calls, as shown in Figure 6. Data also need to be remapped to dummy physical pages. Note that employing the unlinking technique, which replaces the fall-through with the NOP gadget, can be used to weaken the security features of the sensitive function.

Phase 3 - Modifying Page Tables. The vulnerability of arbitrary memory read and write is leveraged to remap page tables with the chained gadgets and data. As described in Section 3, we assumed that an attacker can exploit kernel vulnerabilities to obtain useful symbol information from the kernel. Consequently, the attacker can traverse the process list, obtain the page tables of the malicious application, and modify them according to the gadget chain.

If the attacker forcibly modifies the page tables of the kernel code area, it may cause side effects on other processes that
share the same kernel code. To resolve this issue, we utilize private page tables to prevent the propagation of changes to all processes. Whenever we need to modify the original page tables, we allocate private page tables using the following free page allocation methods of POP. The pages are initialized by copying data from the original page tables, and the copied pages replace them. For instance, in the x86_64 Linux system, the top-level private page table is set to the CR3 register of the CPU by modifying the target process context within the kernel, such as the task_struct and mm data structures. Lower-level private tables replace the original ones by modifying related entries in higher-level page tables [40]. Additionally, commodity OSes typically employ large pages, such as 2 MB and 1 GB pages, for the linear address space. In that case, private page tables have to be allocated by extending the large page to 4 KB pages.

**Phase 4 - Allocating Free Physical Pages.** New physical pages have to be allocated to create private page tables. They are also required even when passing data as arguments to the system call candidate during the exploitation process. Securing these new physical pages can be achieved by (i) allocating pages from the free physical page pool in the kernel or (ii) obtaining them from the process heap area.

The kernel manages a pool of free physical pages to handle system memory efficiently and sequentially allocates pages when necessary. Allocating pages from the kernel exploits this characteristic. When free pages are needed, the method arbitrarily obtains them from the end of the pool to the beginning with a simple counter. It is a straightforward process and does not need memory allocation functions of the kernel. However, when the system memory is exhausted, the kernel will overwrite the allocated pages.

On the other hand, the heap is an isolated area for each process. Therefore, obtaining free pages from the heap can result in individual availability of free pages. However, the method requires traversing the page tables of the process to determine their physical pages.

### 5.4 Page Flushing

The remapped addresses resulting from the page stitching step may interfere with previously cached addresses in a TLB. Therefore, it is crucial to flush the TLB prior to exploitation. In commodity OSes, the TLB entries are frequently flushed as their memories are exhausted by applications, system services, and interrupts. This implies that TLB entries associated with the pages that were forcibly remapped are flushed out in adequate time after page stitching. Unfortunately, in x86_64 systems, the page table has a global bit that delays the flushing. This global bit is used for kernel code pages that are used in process-wide sharing. As such, to accelerate TLB flushing, the page flushing step unsets the global bits of the entries that are remapped through the page stitching step. Moreover, it leverages the CPU affinity to perform exploitation and page flushing on the same core because of the independent TLB for each CPU core.

### 6 Evaluation

POP is not only a practical but also an effective CRA under state-of-the-art CFI enforcement. To demonstrate its effectiveness, we revisited a real-world page-remapping vulnerability, CVE-2013-2595 [53], along with its exploit code [64] on a Linux system. The vulnerability allows arbitrary remapping of physical pages to the userspace. Consequently, an attacker can read and write arbitrary kernel memory and conduct typical page modification attacks, such as altering kernel code or credentials. However, hypervisor-based kernel integrity protection can prevent these typical page modifications, as discussed in Section 3. To show the page-level CRA still functions under the CFI enforcement, we rewrote the exploit code only with the arbitrary memory read and write capability and POP technique. Additionally, we assessed the distribution of system call candidates and gadgets across various kernel versions and configurations to demonstrate the feasibility of POP.

The evaluation was conducted on an HP Victus 16 laptop, which features an Intel i7-12700H processor and 16 GB of memory. To enable Clang/LLVM CFI with CET and FineIBT, we employed Linux kernel versions 6.1.12 and 6.2.8, respectively, and compiled these kernels using LLVM 6.0.0 on Ubuntu 22.04.2. To leverage the SLAT and MBEC features, we extended an open-source lightweight hypervisor [33] and integrated hypervisor-based kernel integrity protection. We ported CVE-2013-2595 to our vulnerable kernel driver and executed it for evaluation. The code and data we used are accessible via our GitHub repository [32].

At the time of writing, we evaluated POP against our testbed, and no attacks were successful.

### 6.1 Proof-of-Concept Exploitation

We constructed a PoC exploit for the Linux kernel version 6.2.8 with FineIBT. Another exploit for Clang/LLVM CFI with CET is not significantly different, except for the address values. As several CRA studies [3, 8, 15, 19] have shown the effectiveness of indirect call gadgets, we developed a round-trip type exploitation scenario to demonstrate the distinctive differences and characteristics of POP. One of the strengths of POP is the ability to provide attackers with the flexibility to select arbitrary entry points with arguments. To emphasize this advantage, we intentionally chose a system call candidate whose direct branch aligns with the page offset of the security-sensitive function and whose argument has a length of 64
Table 2: Important symbol names and offset values in the kallsyms_offsets table. The offset values indicate distances from the start of the .text section.

<table>
<thead>
<tr>
<th>Symbol Name</th>
<th>Offset Value</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>sys_call_table</td>
<td>0x1400400</td>
<td>Breaking</td>
</tr>
<tr>
<td>_x64_sys_read</td>
<td>0x46fda0</td>
<td>KASLR</td>
</tr>
<tr>
<td>clear_tasks_mm_cputask()</td>
<td>0xeb800</td>
<td>Identifying kernel data structures</td>
</tr>
<tr>
<td>prepare_kernel_cred()</td>
<td>0x1257f0</td>
<td></td>
</tr>
<tr>
<td>__set_task_comm()</td>
<td>0x476f10</td>
<td></td>
</tr>
<tr>
<td>pgd_alloc()</td>
<td>0x6e840</td>
<td>Performing POP</td>
</tr>
<tr>
<td>init_task</td>
<td>0x201b00</td>
<td></td>
</tr>
<tr>
<td>page_offset_base</td>
<td>0x19d70078</td>
<td></td>
</tr>
<tr>
<td>__per_cpu_offset</td>
<td>0x19d9e0</td>
<td></td>
</tr>
<tr>
<td>commit_creds()</td>
<td>0x1253b0</td>
<td></td>
</tr>
</tbody>
</table>

Preparation of Attack Primitives and Page Carving. To perform POP, an attacker must acquire symbol information and data structures from the kernel. In Section 3, we assumed that the attacker could run arbitrary programs locally and exploit a kernel memory read and write vulnerability. As a result, we obtained the kernel binary from the boot directory of the system and disassembled it using the objdump tool. Within the kernel binary, we located the kallsyms_offsets table to access symbol information. The table contains the offset of each kernel symbol and is composed of hundreds of 4-byte values in a continuously growing format. We identified the table by searching for its distinctive feature within the kernel binary. To complement symbol names to the table, we extracted them from the /proc/kallsyms file because its symbol list originates from the table [42]. We then added symbol information to the kernel assembly code. Finally, we collected system call candidates and gadgets through static analysis of the assembly code we generated, as discussed in Section 5.2. We employed the Python scripting language for it and compiled a list for page stitching.

To access the kernel area from the user level and break KASLR, we first read the /proc/meminfo file to obtain the system memory size. Then, we exploited CVE-2013-2595 to map all physical memory to the user area and searched the start address of the kernel code by matching its signature in every 4 KB page. Once we found the start address of the kernel, we could read and write arbitrary kernel memory using this address along with symbol information. KASLR loads the kernel into a random location each time the system boots. To defeat it, we needed to obtain the address of an arbitrary kernel function. The kernel stores the addresses of system call functions in the sys_call_table array, and the first system call is sys_read(). Therefore, the first value in sys_call_table contains the address of _x64_sys_read() (0xffffffffbbefda0). By subtracting the offset value of _x64_sys_read() (0x46fda0) within the kallsyms_offsets table from it, we obtained the base address of the kernel code (0xffffffffbba00000) and broke KASLR. Using the base address, we consequently mapped kernel virtual addresses to userspace virtual addresses. Hereafter, we refer to kernel addresses calculated based on the default kernel address (0xffffffff81000000).

We identified kernel structures by analyzing the kernel assembly code with the symbols to trace runtime kernel data. For example, we obtained each offset of the field in the task_struct and cred data structures by analyzing clear_tasks_mm_cputask(), prepare_kernel_cred(), and __set_task_comm() that modify the fields. We also reconstructed the mm_struct data structure from pgd_alloc() to manipulate page tables. Using the data structures, symbol information, and the base address of the kernel, we traversed the init_task variable (0xffffffff8301bb00) with the well-known technique that traces linked lists [25] to obtain the task_struct data for the malicious application. We also accessed the top-level page table information for page stitching with the pgd field of the mm_struct data structure within the task_struct data. Furthermore, by reading the page_offset_base variable (0xffffffff829d7008), we obtained the direct mapping area (0xffffffff888000000000) where system memory is mapped one-to-one, then used it to allocate free physical pages. The __per_cpu_offset table (0xffffffff829d9e0) was also identified to access the per-CPU data of CPU 0 (0xffffffff8846f600000). The symbol information we utilized for POP is summarized in Table 2.

Page Stitching. The control and data flows of the security-sensitive function, commit_creds, are shown in Figure 7. To clarify, we merge duplicated data references and function calls into a single entity. In line 3, the function starts with the endbr64 instruction that allows indirect branches to execute it. This implies indirect call gadgets can reach it even under CET enforcement. In line 8, the rbx register stores the sum of the base address of the gs segment, the value of the rip register, and 0x7ef0c776 to access the task_struct data of the currently running process, that is, current_task. The instruction employs a combination of the segment- and instruction pointer-based addressing modes. The value of the rip is 0xffffffff811253ca. When it is added to 0x7ef0c776, the result represents an offset of 0x31b40 (percpu_hot) from the base address of the gs segment. The base address of CPU 0 in __per_cpu_offset variable is 0xffffffff88846f600000. Consequently, by adding the offset and the base address, we can obtain the final address of the current_task, 0xffffffff88846f631b40. In line 13, the suid_dumpable variable is also accessed via the rip register, and the address of the value is 0xffffffff846178a8.

The most important part of commit_creds() is in line 26, where credentials are updated. Other functions are not essential and can be remapped or replaced with NOP gadgets. A straightforward approach is to replace these
non-essential functions with NOP gadgets, which helps us avoid the potential exploitation caused by remapping. However, when it comes to page remapping, it is important to note that if a function shares the same page as commit_creds() or if two or more functions are located on the same page, those functions cannot be replaced with NOP gadgets. While developing the PoC exploit, we identified four functions: inc_rlimit_ucounts(), dec_rlimit_ucounts(), key_fsuid_changed(), and key_fsgid_changed(), which could not be replaced with NOP gadgets. The first two functions had no subfunctions, while the latter two had multiple ones. Upon further analysis, we discovered that their subfunctions were not called when the thread keyring field of the cred data structure was zero. With this information, we remapped these functions and modified the thread keyring field accordingly.

Finding gadgets with specific direct branches is a straightforward process, and we manually searched gadgets from the gadget list using a simple text search tool, the grep tool. To perform page stitching with minimal gadgets, we chose __x64_sys_removexattr(), which included a direct branch at 0xffffffff814af3b0. The page offset of the system call matched the page offset of commit_creds() at 0xffffffff811253b0 with a displacement of 0x38a000. We also chose the functions and data listed in Table 3. They were either remapped for the system call or replaced by NOP gadgets, as discussed in Section 5.3.

Free physical pages were allocated in reverse order, beginning from 0xffffffff80003ff000, which is 16 GB away from the value of page_offset_base. One free page was allocated for the malicious credentials, which were copied from the credentials of init_task. The thread_keyring field of the malicious credentials was set to zero to skip the subfunctions. Additionally, ten free pages were utilized to maintain private page tables.

### Page Flushing and Exploitation

Before page flushing, we attached the malicious application to CPU 0 with the taskset tool and cleared all global bits of the remapped pages from the page tables. Subsequently, we waited until the remapped pages were flushed from the TLB. Through our experiments, 60 seconds were sufficient to wait in our evaluation environment. Finally, the application successfully obtained root privilege by passing the malicious credentials (0xffffffff880003ff000) as an argument to the __x64_sys_removexattr system call.

### 6.2 Branch and Gadget Distributions

The Linux kernel consists of a static-linked vmlinux file and kernel modules. Kernel modules have a significant amount of code base but are selectively loaded only when necessary. Therefore, they are not always available for exploitation. Commodity OSes typically tailor their kernels based on the default configuration of the Linux kernel. Consequently, we focused on the gadget list using a simple text search tool, the grep tool. To perform page stitching with minimal gadgets, we chose __x64_sys_removexattr(), which included a direct branch at 0xffffffff814af3b0. The page offset of the system call matched the page offset of commit_creds() at 0xffffffff811253b0 with a displacement of 0x38a000. We also chose the functions and data listed in Table 3. They were either remapped for the system call or replaced by NOP gadgets, as discussed in Section 5.3.

Free physical pages were allocated in reverse order, beginning from 0xffffffff80003ff000, which is 16 GB away from the value of page_offset_base. One free page was allocated for the malicious credentials, which were copied from the credentials of init_task. The thread_keyring field of the malicious credentials was set to zero to skip the subfunctions. Additionally, ten free pages were utilized to maintain private page tables.

### Page Flushing and Exploitation

Before page flushing, we attached the malicious application to CPU 0 with the taskset tool and cleared all global bits of the remapped pages from the page tables. Subsequently, we waited until the remapped pages were flushed from the TLB. Through our experiments, 60 seconds were sufficient to wait in our evaluation environment. Finally, the application successfully obtained root privilege by passing the malicious credentials (0xffffffff880003ff000) as an argument to the __x64_sys_removexattr system call.

### 6.2 Branch and Gadget Distributions

The Linux kernel consists of a static-linked vmlinux file and kernel modules. Kernel modules have a significant amount of code base but are selectively loaded only when necessary. Therefore, they are not always available for exploitation. Commodity OSes typically tailor their kernels based on the default configuration of the Linux kernel. Consequently, we focused on the gadget list using a simple text search tool, the grep tool. To perform page stitching with minimal gadgets, we chose __x64_sys_removexattr(), which included a direct branch at 0xffffffff814af3b0. The page offset of the system call matched the page offset of commit_creds() at 0xffffffff811253b0 with a displacement of 0x38a000. We also chose the functions and data listed in Table 3. They were either remapped for the system call or replaced by NOP gadgets, as discussed in Section 5.3.

Free physical pages were allocated in reverse order, beginning from 0xffffffff80003ff000, which is 16 GB away from the value of page_offset_base. One free page was allocated for the malicious credentials, which were copied from the credentials of init_task. The thread_keyring field of the malicious credentials was set to zero to skip the subfunctions. Additionally, ten free pages were utilized to maintain private page tables.

### Page Flushing and Exploitation

Before page flushing, we attached the malicious application to CPU 0 with the taskset tool and cleared all global bits of the remapped pages from the page tables. Subsequently, we waited until the remapped pages were flushed from the TLB. Through our experiments, 60 seconds were sufficient to wait in our evaluation environment. Finally, the application successfully obtained root privilege by passing the malicious credentials (0xffffffff880003ff000) as an argument to the __x64_sys_removexattr system call.
on the .text section of the vmlinux file and compared kernels built with commodity and kernel default configurations to show the feasibility of POP.

**Distribution of System Call Candidates.** POP can utilize both 64-bit and 32-bit system calls, and we identified all system call candidates with prefixes such as __x64 and __ia32 across kernel versions. While extracting system call candidates using the common rules described in Section 5.2, we applied additional strict rules. (i) The first direct branch of the system call has to be explicitly reached without any control-flow diversions and exceptions. System calls were excluded if they contained conditional branches, floating-point instructions, or undefined instructions before the first call or jump instruction. (ii) At least, the first argument (rdi) has to remain controlled before the first call or jump instruction. We also applied these rules to function and partial gadgets. Table 4 lists the total number of system calls and the number of system call candidates that were identified. Even with the strict rules, we could identify more than 220 system call candidates out of 992 system calls. Kernels built with default configurations had fewer system call candidates than commodity configurations. The default configuration did not retain frame pointers for functions, resulting in shorter system call functions. Consequently, some system call candidates were excluded because their branch targets pointed to the pages where their branches are located.

While analyzing system call candidates, we discovered that page offsets of direct branch targets were aligned to a 16-byte boundary. The alignment characteristic was observed in both Clang/LLVM CFI and FineIBT. Thus, the first gadget following a system call candidate must be 16-byte aligned. However, this does not limit the capability of POP because function gadgets are also aligned to it. Hereafter, we refer to 16-byte aligned as aligned and 16-byte unaligned as unaligned.

**Distribution of Function and Partial Gadgets.** We applied three rules while identifying all functions and partial gadgets. First, we searched all functions in the kernel for function gadgets, selecting only complete function forms. Tail-call functions that call other functions with jump instructions were excluded because they require additional gadgets. Although system call candidates could also be used as function gadgets, we did not include them in our evaluation. Second, we disassembled code units of every 32 bytes across all page offsets to identify partial gadgets. The size of the partial gadgets was determined based on two reasons: (i) the branch targets of system call candidates are aligned by 16 bytes. A 32-byte size allows for address overlapping and is adequate for collecting gadgets. (ii) We aimed to keep the gadget size and search space small for simplicity. After identifying the function and partial gadgets, we selected call gadgets from them if the target of the first branch was reachable and valid. Specifically, we considered six arguments: rdi, rsi, rdx, rcx, r8, and r9, which could be passed to the system call candidates for indirect call gadgets. Finally, we extracted NOP gadgets in which no call and jump instructions go outside of the gadgets. All gadgets were considered for both the round-trip and one-way exploitation strategies.

The results of analyzing the gadgets for each kernel version are listed in Table 5. The code size represents the size of the .text section extracted with the objdump tool from the vmlinux file. The numbers at the top are the count of aligned gadgets, while the bold numbers in parentheses indicate the sum of aligned and unaligned gadgets. Kernels with commodity configurations had more gadgets than default configurations, except for call gadgets that use jump instructions. The reason was that the default configuration frequently utilized tail calls with jump instructions. The table reveals that the numbers of aligned direct call and NOP gadgets are sufficient for performing POP. However, the number of aligned indirect call gadgets is noticeably smaller compared to the others.

To find a way to reach unaligned indirect call gadgets, we analyzed the distribution of branch targets in aligned direct call gadgets. Table 6 lists the number of aligned direct call gadgets whose branch targets can jump to unaligned addresses. The table reveals that at least over 8.7% of total gadgets have unaligned branch targets. Figure 8 illustrates the distribution of page offsets for these unaligned branch targets. Each dot on the map represents gadgets within a 16x16 address range, indicating the presence of multiple gadgets in this range. Kernels built with Clang/LLVM CFI, as shown in Figures 8a and 8b, exhibit a large number of unaligned branches within the ranges of 0x4ea1 to 0x51f and 0x5c1 to 0x5cf, respectively, resulting in horizontal lines. Conversely, kernels built with FineIBT, shown in Figures 8c and 8d, do not display such patterns. Furthermore, all kernel versions display concentrated areas at the upper-left and lower-right corners. Although gadgets are gathered in specific offsets, unaligned branch targets are evidently distributed across various page offsets. In conclusion, unaligned indirect call gadgets can be reached utilizing unaligned target branches, as shown in Table 5.

### 7 Discussion and Mitigation

Although POP has the capability to generate new control flows through the DOA technique even in the presence of strong
Table 5: Numbers of call and NOP gadgets in Linux kernels. Code size indicates the .text section size of the kernel binary. The numbers at the top of function and partial gadgets represent the number of aligned gadgets. The bold numbers in parentheses represent the sum of aligned and unaligned gadgets.

<table>
<thead>
<tr>
<th>Version</th>
<th>Configuration</th>
<th>Code Size (KB)</th>
<th>Function Gadgets</th>
<th>Partial Gadgets</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Direct Call</td>
<td>Partial Gadgets</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Call</td>
<td>NOP</td>
</tr>
<tr>
<td>6.1.12 (Clang/LLVM CFI with CET)</td>
<td>Commodity</td>
<td>18,440.6</td>
<td>6,447 (6,466)</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Kernel Default</td>
<td>18,444.8</td>
<td>5,495 (5,503)</td>
<td>2</td>
</tr>
<tr>
<td>6.2.8 (FineIBT)</td>
<td>Commodity</td>
<td>20,480.0</td>
<td>6,500 (6,507)</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Kernel Default</td>
<td>18,432.0</td>
<td>5,504 (5,506)</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 6: Number of aligned direct call gadgets and unaligned branch targets in them.

<table>
<thead>
<tr>
<th>Kernel Version</th>
<th>Configuration</th>
<th>Aligned Direct Call Gadgets</th>
<th>Unaligned Branch Target</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Total</td>
<td>Unaligned Branch Target</td>
</tr>
<tr>
<td>6.1.12 (Clang/LLVM CFI with CET)</td>
<td>Commodity</td>
<td>78,231</td>
<td>8,889</td>
</tr>
<tr>
<td></td>
<td>Kernel Default</td>
<td>74,385</td>
<td>8,540</td>
</tr>
<tr>
<td>6.2.8 (FineIBT)</td>
<td>Commodity</td>
<td>80,845</td>
<td>7,018</td>
</tr>
<tr>
<td></td>
<td>Kernel Default</td>
<td>74,282</td>
<td>6,430</td>
</tr>
</tbody>
</table>

CFI policies, it also has certain limitations stemming from its dependence on page table modifications. These limitations can be leveraged as mitigations to effectively prevent the success of POP attacks.

Limitations of Page-Oriented Programming. The remapping of functions and gadgets has the potential to disrupt the exploitation flow. For instance, if memory copy functions such as strcpy() and memcpy() are replaced with gadgets, recursive calls may occur and lead to failure when the security-sensitive function executes these functions. Furthermore, if the subfunctions of the sensitive function are spread over multiple physical pages and remapped, they may unintentionally modify the execution flow of other kernel functions, resulting in their failures. However, these limitations can be alleviated by (i) substituting unused functions in the exploitation flow with gadgets, (ii) unlinking non-essential subfunctions instead of remapping them, and (iii) recovering all remapped pages to the original pages after exploitation.

Page table modification requires multiple read-and-write operations, exploiting an arbitrary memory access vulnerability in the kernel. These attempts may lead to system crashes during context switches or be affected by other CPU events, depending on the characteristics of the vulnerability. Even when leveraging CVE-2013-2595, the system can still crash because page table modification is not an atomic operation. Nonetheless, the risk can be reduced by setting the top-level table of private page tables to the pgd field of the mm_struct data structure after thoroughly preparing the private page tables, as described in Section 5.3. This approach can mitigate
the side effects of partially-linked private tables and allow for atomic-like modification.

**Mitigations.** Several mechanisms, including Secvisor [68], HyperSafe [76], TrustZone-based Real-time Kernel Protection (TZ-RKP) [6], Secure Kernel-level Execution Environment (SKEE) [7], kCoFI [18], and the work of Song et al. [69], have been proposed to protect page tables by escorting page updates. These methods are effective in detecting and preventing unauthorized modifications of them. However, frequent page updates can lead to notable performance overhead. Data-flow integrity (DFI) [11] and software fault isolation (SFI) [12, 74] can prevent POP by limiting arbitrary memory read and write vulnerabilities. They impede paths to the kernel DOAs, but runtime overhead is not negligible. In contrast, PT-Rand [20] and Microsoft Windows employ a strategy of randomizing page tables, thus aiming to hide their locations. Randomizing page tables incurs lower performance overhead; however, completely concealing them from all execution paths within the kernel is challenging [67].

Recent studies on kernel compartmentalization and domain isolation [47, 49, 65] leverage special hardware features such as pointer authentication code with memory tagging extension and SLAT table switching called Extended Page Table pointer (EPTP) switching of the hardware-based virtualization technology. Although these mechanisms require kernel changes in commodity OSes to collaborate, they provide robust compartment or domain isolation through hardware and access policies. Therefore, they can prevent page remappings with lower overhead by isolating page tables from unauthorized accesses.

The 12th generation processors from Intel have recently introduced a specialized extension called HLAT as part of the Virtualization Technology-Redirect Protection (VT-rp). This enhancement aims to mitigate page remapping attacks. At the time of writing, Microsoft Windows 11 ensures support for the HLAT extension [37], while Linux does not. With HLAT, the guest OS is required to invoke hypercalls before and after safely updating the page table information. However, this may potentially hinder performance and cause synchronization issues in multicore environments. Furthermore, HLAT is only available on Intel CPUs starting from the 12th generation, and it is uncertain whether all CPU vendors will support this feature. As demonstrated, POP remains a viable and effective attack technique for most current systems, and legacy systems still need other mitigations we discussed.

### 8 Related Work

**Control-Flow Integrity.** Heuristic-based CFI research, such as ROPGuard [26], kBouncer [63], and ROPPecker [14], have focused on analyzing return traces to determine the characteristics of ROP attacks with reasonable performance overhead. Other studies, such as Bin-CFI [79], CCFI [78], opaque CFI (O-CFI) [57], kCoFI [18], kCFI [29], MCFI [61], and indirect function-call checks (IFCC) [72], employed static analysis on source code or binaries to investigate control flow deviations using CFGs generated from the analysis.

In contrast, pCFI [62], PathArmor [73], PittyPat [22], µCFI [35], CFI-LB [44], and OS-CFI [43] aimed to generate precise CFGs by combining dynamic information such as execution flow, execution path, and pointer origin information, to enforce stringent CFI policies. Additionally, hardware-based techniques for preventing deviations in indirect branches were explored, including hardware-enforced CFI (HCFI) [16], Transactional Synchronization Extensions-based CFI (TSX-CFI) [59], and PAL [77]. These techniques leveraged customized or commodity hardware such as Intel Transactional Synchronization Extensions (TSX) [38] and ARM Pointer Authentication (PA) [5].

Previous CFI research has made substantial contributions to enforcing CFI policies by reducing the target set of the indirect branch and leveraging hardware support. However, a limitation of these approaches is that they primarily concentrate on enforcing policies over indirect branches. Although kCoFI and kCFI considered page table protection, it is still uncertain whether they can effectively prevent POP in commodity OSes with negligible performance overhead.

**Bypassing Control-Flow Integrity.** Several research groups, including Carlini and Wagner [10], Davi et al. [21], Göktaş et al. [30, 31], Evans et al. [24], and Carlini et al. [9] have investigated the weaknesses of CFI policies. They demonstrated that several CFI studies based on heuristics and static CFGs are still bypassed using techniques such as flushing return traces and exploiting target sets of indirect branches.

Instead of exploiting the weaknesses, other studies by Chen et al. [13], Morton et al. [58], and Hu et al. [34] proposed DOAs that rely only on data to escalate privileges without deviating from the CFG. Data-oriented programming (DOP) [36] and block-oriented programming (BOP) [39] proposed CRA techniques along valid execution paths. Additionally, several studies have targeted the kernel [23, 45, 46] and demonstrated code modification and privilege escalation by modifying non-control data, such as page tables and credentials. They exemplified that DOAs are applicable even beyond the user level.

POP can be compared to DOP and BOP in the sense that it achieves the CRA without modifying any code. However, POP is not quite restricted by CFI policies because it exploits direct branches. Consequently, POP does not require complex computations, making it a more straightforward and intuitive technique.

**Kernel Integrity Protection.** In x86_64 environments, previous research on kernel integrity protection with hardware has employed the virtualization technology (VT) of the CPU. Diverse approaches, including Secvisor [68], NICKLE [66], and Shadow-box [33], have utilized the SLAT of VT to ensure that kernel code remains non-writable. Secvisor, in particular, proposed a page update mechanism using the hypervisor, but
its performance overhead was more than double because all page faults were handled by it. In contrast, PrivWatcher [41] focused on a lightweight protection mechanism for credentials. It leveraged SLAT to establish a safe region inside the kernel to prevent unauthorized modification of credentials.

SeCage [47] and xMP [65] utilized the SLAT-related feature, EPTP switching, and supported memory isolation with access policies. Their isolation mechanisms could isolate page tables from unauthorized accesses with kernel changes. Recently, a new hardware feature, HLAT, has been introduced by Intel to hinder page remapping attacks. Although the latest CPUs may mitigate performance overhead and prevent POP through it, protecting legacy systems with older CPUs or without support for the feature is an open problem.

9 Conclusion

We introduced a novel CRA technique called POP and demonstrated its PoC on an up-to-date system. This attack is practical and capable of creating arbitrary control flows, effectively bypassing the current CFI implementations by undermining their critical assumption of code memory immutability. While the attack can be mitigated on cutting-edge hardware products, other existing systems still remain susceptible to this attack at present.

References


