Beyond the Edges of Kernel Control-Flow Hijacking Protection with HEK-CFI

Lukas Maar
Graz University of Technology
Austria

Pascal Nasahl
Graz University of Technology
Austria

Stefan Mangard
Graz University of Technology
Austria

ABSTRACT

Over the past decade, vulnerabilities in the Linux kernel have more than doubled, allowing control-flow hijacking attacks that compromise the entire system. To thwart these attacks, Control-Flow Integrity (CFI) has emerged as state-of-the-art. However, existing kernel CFI schemes are still limited in providing protection against these attacks, e.g., during system events and for return addresses.

In this paper, we introduce Hardware-Enforced Kernel Control-Flow Integrity (HEK-CFI), a novel approach that protects control-flow-related data during system events, as well as function pointers and return addresses, effectively mitigating control-flow hijacking attacks. HEK-CFI leverages Intel CET, specifically write-protected pages used by its shadow stack design, along with signature-based CFI to safeguard this data. To demonstrate its effectiveness, we implement a proof-of-concept and perform a case study on the Linux kernel v5.18. In our case study, HEK-CFI eliminates all illegal backward-edge targets and reduces forward-edge targets by more than 50 % compared to all existing kernel CFI schemes. We evaluate our proof-of-concept on real hardware and observe a performance overhead of 12.3 % for micro benchmarks and 1.85 % for macro benchmarks. In summary, HEK-CFI is the first solution to provide protection for both system events and return addresses. HEK-CFI also generically reduces forward control-flow targets and the performance overhead compared to existing solutions.

CCS CONCEPTS

• **Security and privacy → Operating systems security**.

KEYWORDS

Kernel Control-Data Integrity, Kernel Control-Flow Integrity

ACM Reference Format:
Lukas Maar, Pascal Nasahl, and Stefan Mangard. 2024. Beyond the Edges of Kernel Control-Flow Hijacking Protection with HEK-CFI. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '24), July 1–5, 2024, Singapore, Singapore.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3634737.3661135

1 INTRODUCTION

Over the past decade, the number of vulnerabilities in the Linux kernel has increased dramatically, from 114 CVEs in 2012 to 318 in 2022, according to the NIST NVD. These vulnerabilities can be devastating, allowing to hijack the control flow to gain arbitrary code execution and compromise the entire system.

Despite efforts by processor vendors [12, 19, 47] and kernel developers [20, 32, 33] to mitigate these attacks, adversaries continued to devise more sophisticated code reuse attacks [5, 7, 10, 28]. For instance, Return Oriented Programming (ROP) [7] redirects the control flow to a chain of gadgets, each ending with `ret`. To counter control-flow hijacking attacks, Control-Flow Integrity (CFI) [1] has emerged as a state-of-the-art mitigation by restricting the control flow to an approximated Control-Flow Graph (CFG).

Kernel CFI-based mitigations face two critical challenges in providing security. First, kernel programs must handle system events [6], i.e., context switches, interrupts, exceptions, and syscalls. During these events, the current thread state (which refers to the set of registers that store its run-time information) is stored in memory, ensuring that it can be restored when the thread resumes execution. If these stored states are not fully protected on any of these events, an adversary can corrupt them to hijack the control flow when the state is restored, as described in prior research [16, 23] and exploited by security researchers at Google Project Zero (cf. CVE-2022-42703) [56]. Second, it is difficult to establish an accurate representation of the CFG for kernel programs due to their large size. Coarse approximations leave the system vulnerable to bypass attacks. This is particularly critical for the backward edges, as static determination leaves the system vulnerable to Control-Flow Bending (CFB) [9], re-enabling control-flow hijacking attacks.

Existing kernel CFI-based mitigations [3, 16, 18, 22, 23, 41, 48, 62, 63, 68] are still limited in addressing these challenges as they do not provide sufficient protection for both the thread state during all system events and backward edges, i.e., return addresses. For example, defenses based on ARM's Pointer Authentication (PA), e.g., PAL [62] and Camouflage [18], inadequately protect the stored thread state, allowing attackers to tamper with it and hijack the control flow. Other approaches, e.g., KCoFI [16] and Fine-CFI [41], statically determine backward edges, which makes their systems vulnerable to CFB, bypassing the applied mitigation. Additionally, the Code Pointer Integrity (CPI) [37] solution combined with the user-space memory isolation scheme CETIS [66] safeguards code pointers and return addresses. However, this combination cannot address the first challenge because it is not designed for kernel-level system events that require special thread state handling. Consequently, it cannot protect the kernel during these critical events.

In summary, existing kernel CFI schemes and CPI combined with CETIS have limitations in providing protection for the kernel during system events and for return addresses. As a consequence, adversaries can exploit these limitations to bypass the applied CFI scheme, thereby re-enabling control-flow hijacking attacks.

In this paper, we introduce Hardware-Enforced Kernel Control-Flow Integrity (HEK-CFI) to fill the gap in protecting against kernel

ASIA CCS '24, July 1–5, 2024, Singapore, Singapore

ACM ISBN 979-8-4007-0482-6/24/07.
https://doi.org/10.1145/3634737.3661135

control-flow hijacking attacks. HEK-CFI consists of three key mechanisms. First, HEK-CFI ensures kernel control-data integrity by retrofitting write-protected pages from Intel CET SHadow StacK (SHSTK) for the *supervisor*. It does this by providing write-protected local and global safe areas within the kernel, where control data is securely stored. In particular, our write-protected local safe area extends well beyond the original purpose of Intel CET SHSTK. While SHSTK protects backward edges, our approach safeguards any local control data, such as during low-level environments like system events. Second, HEK-CFI utilizes our control-data integrity scheme to protect the thread state during all system events, including protection for the SHSTK state. This prevents attackers with memory write capabilities from tampering with the thread and SHSTK state. With these two mechanisms, we are the first to provide comprehensive protection for both system events and return addresses. Third, HEK-CFI combines our control-data integrity with signature-based CFI to protect forward control-flow edges, particularly control data, e.g., function pointers. Signature-based CFI efficiently protects function pointers with rare signatures, while control-data integrity fully safeguards any control data, albeit with a potentially higher performance overhead. To balance this trade-off, HEK-CFI automatically selects the optimal scheme for each control data

While HEK-CFI's true contributions rely on efficient control data (including thread state) protection, it demonstrates its practical use with the third mechanism to protect forward edges as well.

We are the first to implement the Intel CET SHSTK for the *supervisor* in the Linux kernel. This allows us to implement a HEK-CFI proof-of-concept, realized as a compiler-assisted software framework consisting of a Linux kernel extension, a code analyzer [24], and an LLVM pass [38]. Our framework automatically hardens the Linux kernel using a user-configurable CFI precision level as input. To demonstrate its effectiveness, we perform a case study where we eliminate all illegal backward edges and reducing forward control-flow edge targets by 99.98 % [16], 93.3 % [3, 46], 76.4 % [23], and 50.4 % [41] compared to existing kernel CFI schemes.

We evaluate HEK-CFI's security, demonstrating strong security guarantees against control-flow hijacking attacks. We run our proof-of-concept on Ubuntu 22.04.1 LTS on a recent Intel Alder Lake processor supporting Intel CET SHSTK. We observe a performance overhead of 12.3±1.5% for the LMbench [45] micro benchmarks. For macro benchmarks, the performance overhead is 1.85 ± 1.02% for Phoronix Test Suite [49] and 0.17 ± 0.23% for SPEC CPU 2017 [15]. In summary, HEK-CFI is the first solution to provide protection for both system events and return addresses. It also generically reduces forward control-flow targets and performance overhead compared to existing CFI schemes. As a result, HEK-CFI improves kernel security to protect against control-flow hijacking attacks.

The main contributions of this work are:

(1) Kernel control-data integrity: We provide kernel control-data integrity, including a secure approach to protect system events and return addresses, establishing ourselves as the first kernel solution to safeguard both.

(2) HEK-CFI: We introduce HEK-CFI, a novel design that combines our kernel control-data integrity with signature-based CFI, ensuring robust protection for control-flow-related data.

(3) POC: We are the first to implement Intel CET SHSTK for the *supervisor* privilege level in the Linux kernel, allowing us to implement a HEK-CFI proof-of-concept as a compiler-assisted framework, both of which we provide open-source.

(4) Case study: We perform a case study to demonstrate the effectiveness of HEK-CFI in protecting forward and backward edges, as well as system events. We show that HEK-CFI provides enhanced efficacy compared to existing solutions.

(5) Evaluation: We evaluate HEK-CFI's security and performance, showing strong security guarantees with an overhead of 12.3 % and 1.85 % for micro and macro benchmarks.

Outline. Section 2 provides background. In Section 3, we present a systematization of existing works. Section 4 introduces HEK-CFI. In Section 5, we implement our proof-of-concept, while Section 6 conducts a case study. Section 7 shows the strong security guarantees. In Section 8, we evaluate the performance overhead. Section 9 discusses related work. Lastly, Section 10 concludes our work.

2 BACKGROUND

This section provides background on control-flow hijacking attacks. At its core, a control-flow hijacking attack comprises two stages. First, an attacker exploits a vulnerability to obtain a Control-Flow Hijacking Primitive (CFHP) [64] allowing them to deviate from the legal Control-Flow Graph (CFG), e.g., memory safety vulnerability to overwrite a function pointer. Second, the attacker utilizes the CFHP to redirect the control flow to an attacker-controlled instruction sequence performing attacker-controlled execution.

CFI. Control-Flow Integrity (CFI) [1] has been established as a state-of-the-art mitigation against these attacks by restricting the control flow to the CFG. However, since complete CFI induces prohibitive runtime overhead, existing CFI schemes restrict the control flow to an approximated CFG. This approximation is achieved through various methods, including static determination (e.g., signature-based [21]) or dynamic techniques (e.g., ARM's Pointer Authentication (PA) [42] or by ensuring integrity of code pointers [37]).

Intel CET. Intel's Control-flow Enforcement Technology (CET) [29] is a recent hardware extension aiming to mitigate scenarios of hijacking attacks. CET consists of the Indirect-Branch Target (IBT) and a SHadow STacK (SHSTK) feature, where CET supports a user and supervisor SHSTK. The IBT feature introduced the `endbr` instruction as a landing pad for indirect branches. When the control flow of an indirect branch is redirected to any other instruction, the hardware raises a control-protection exception. In CET, a shadow stack protects the return path from being maliciously corrupted [58]. On a function call, the hardware pushes the return address onto both the data and the shadow stack. On a function return, both addresses are compared, and a mismatch causes a control-protection exception. Additionally, the hardware pushes sensitive registers (i.e., `rip`, `cs`, and `ssp`) to the supervisor shadow stack on exceptions and interrupts, which are validated on an `iret` instruction. To protect the shadow stack from attackers, it is write-protected using the dedicated page permissions setting dirty and non-writable.

CETIS. CETIS [66] is a user space intra-process memory isolation scheme that uses Intel CET SHSTK's write-protected shadow pages to create a global safe area. This safe area ensures that adversaries cannot tamper with code pointers. By providing write protection for such data, CETIS enhances the efficiency and practicality of Code Pointer Integrity (CPI) [37] on x86_64 systems.

3 THREAT MODEL AND SYSTEMATIZATION

In this section, we first present the threat model and then explore the variety of attack vectors used to obtain a CFHP in the Linux kernel. Additionally, we demonstrate that existing CFI-based mitigations do not fully prevent these CFHPs from violating control-flow restrictions, allowing them to bypass the applied defenses. To illustrate these limitations, we provide example exploits in Appendix 12 and an end-to-end attack exploiting CVE-2019-2215 in Appendix 13.

3.1 Threat Model

We assume that an attacker can arbitrarily execute code in user space. Moreover, the kernel contains a vulnerability that can be exploited to obtain an arbitrary memory read-and-write primitive. This primitive is accessible through the user space without violating the kernel's control-flow integrity. We assume that kernel defense mechanisms are enabled, i.e., the `W^X` [19], SMEP, SMAP [12], and page-table protection [17, 52]. With these mitigations enabled, kernel sections cannot be both writable and executable, the kernel cannot execute or access user space memory, and page tables cannot be manipulated. Our threat model aligns with the threat model of existing kernel CFI-based mitigations [18, 23, 41, 62].

Attack goal. We assume that the primary goal of an attacker is corrupting kernel control-flow-related data discussed in Section 3.2 to hijack the kernel's control-flow.

Out of scope. Data-oriented attacks are another class of attacks that need to be addressed. This work focuses on mitigating control-flow hijacking attacks, while other orthogonal defenses [43, 51, 61] are necessary to address data-oriented attacks. While we acknowledge the existence of side-channel [8, 26, 69], microarchitectural [27, 34], and software fault injection [14, 54] attacks, as well as malicious operating systems, these are out of the scope.

3.2 Attack Vectors

In the Linux kernel, various control-flow-related data (we refer as control data) can be exploited to obtain a CFHP.

Function pointers. Function pointers in writable sections are a well-known security concern [13]. The Linux kernel frequently uses function pointers for various tasks, e.g., calling device functions.

Operation table pointers. To reduce the attack surface of overwriting function pointers, the Linux kernel stores multiple function pointers in operation tables mapped as read-only [13]. However, since the operation table pointers are stored in writable sections, an attacker can tamper with the table pointers instead of function pointers [18, 53, 57]. For instance, each `inode` object has an operation table pointer `i_op` pointing to a read-only `inode_operations` struct containing function pointers for `inode` interaction. The attacker can obtain a CFHP by overwriting the table's pointer with a previously crafted `inode_operations` table.

Return addresses. Another way to redirect the control flow is by manipulating the return address stored on the stack. On function return, the kernel interprets the tampered return address as the execution path for the resumption.

Thread state. The thread state refers to the set of registers that store the runtime information about its thread, e.g., `rip` and `rsp` on x86_64, `pc` and `sp` on arm64, as well as general-purpose registers. During system events [6], i.e., context switch, interrupt, exception, and syscall, the current thread state is stored in memory, ensuring that it can be restored later when the thread resumes execution. This allows the kernel to switch between threads and handle event requests. Since these memory locations are writable, an attacker can manipulate them. If the state is restored from the tampered memory, the thread continues execution based on the tampered state, hijacking the kernel's control flow.

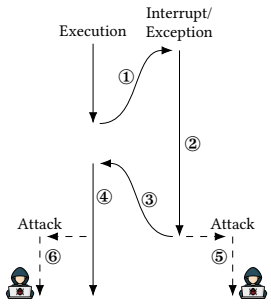

Figure 1: Interrupt or exception disrupts execution.

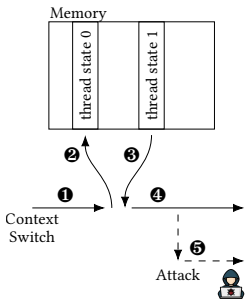

Figure 2: A context switch from thread 0 to thread 1.

Figure 1 shows the system events interrupt and exception, which disrupt the thread's execution and store its state in memory ①. Consequently, the kernel's control flow is legally redirected to handle the invoked request ②. The thread state is restored upon completion ③ to continue its execution ④. By tampering with the stored state, an attacker can perform two attack scenarios. First, they manipulate stored registers, e.g., `rip` for x86_64 and `pc` for arm64, which are interpreted by the return, i.e., `iret` for x86_64 and `eret` for arm64, as the continuing execution ⑤. Second, they tamper with register values later used to redirect the control flow ⑥, e.g., `rax` if the execution performs `call *rax`. We provide motivational exploits in Appendix 12.1. In addition, security researchers at Google Project Zero [56] have exploited CVE-2022-42703 to obtain an uncontrolled arbitrary write, allowing them to corrupt thread state on interrupts and thus hijack control flow.

Another state-changing event is the context switch, where we refer to the Linux kernel design of a context switch: When a thread performs a context switch, it calls into the scheduler to select a new thread to run next, switches the memory descriptor, jumps to the `switch_to` function, and performs a cleanup of the previous thread. The `switch_to` function stores the current thread state in memory and then loads the stored state from the next thread. Figure 2 illustrates the storing and restoring of the state on a context switch ❶-❹. Since the state is stored in writable sections, e.g., the thread's stack frame, an attacker can manipulate it, and gain control of the registers when the state is restored. Taking control of these registers results in hijacking the control flow ❺. For instance, by tampering with callee-saved registers that store a function pointer, the attacker hijacks the control flow on the indirect branch instruction to its tampered register value. We provide motivational exploits in Appendix 12.2 and an end-to-end exploit in Appendix 13.

3.3 Systematization of Existing Works

We investigate the limitations of existing kernel CFI-based mitigations [3, 16, 18, 23, 41, 46, 62, 68] in protecting against kernel

Table 1: Systematization of existing kernel mitigations.

Mitigations	Attack Vector			
	Thread state	Return addresses	Operation table pointers	Function pointers
Ge et al. [23]	○	○	■	●
kCFI [3]	○	○	□	○
Fine-CFI [41]	○	○	■	●
PATTER [68]	○	●	◎	●
Camouflage [18]	○	●	●	○
PAL [62]	○	●	◎	●
FineIBT [46]	○	○	□	○
KCoFI [16]	●	○	□	○
Intel CET SHSTK [29]	○	●	○	○
CPI [37] + CETIS [66]	○	●	◎	●
HEK-CFI	●	●	●	●

● Protection ○ Insufficient protection
■ Implicit protection □ Implicit insufficient protection
◎ Does not protect but can be extended.

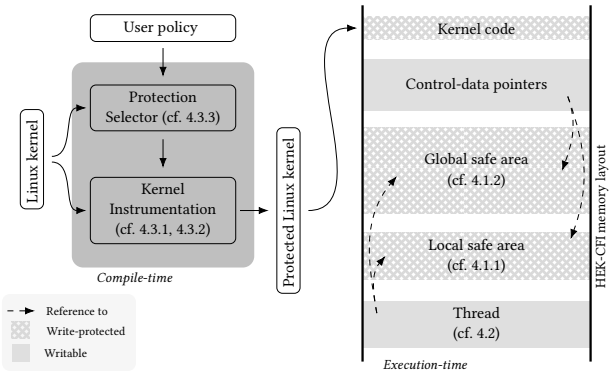

Figure 3: HEK-CFI instruments the kernel to perform runtime validation checks ensuring protection for control data.

control-flow hijacking attacks, where we refer to Section 9 for detailed information. We include Intel CET SHSTK [29] and CPI [37] combined with CETIS [66] to highlight their limitations in protecting specific attack vectors. To illustrate our findings, we use a classification scheme in Table 1. Mitigations marked with ● provide protection for the attack vector, while those marked with ○ provide no protection or can be bypassed by attacks we present in the following. For mitigations that implicitly protect the attack vector or implicitly protect it insufficiently, we use ■ or □, respectively.

Thread state. Various kernel mitigations either do not (i.e., Camouflage [18], FineIBT [46], kCFI [3], and PATTER [68]) or inadequately (i.e., PAL [62], Fine-CFI [41], and the proposal from Ge et al. [23]) protect the stored thread state. As a result, attack scenarios, such as ❺ from Figure 2, and ⑤ and ⑥ from Figure 1, can redirect the control flow and, thereby, bypass control-flow restrictions (see Appendices 12.1 and 12.2 for exploitation examples).

Intel CET SHSTK for supervisor [29] also fails to adequately protect the thread state, as they do not mitigate attack scenarios ⑥ and ❺ from Figure 1 and Figure 2. Moreover, since the shadow stack pointer is part of the thread state, Intel CET SHSTK can be compromised using ❺ as well. CPI [37] combined with CETIS [66] suffers from a similar issue, as it lacks a low-level protection primitive, like protected local storage, to safeguard the thread state.

Return addresses. Kernel mitigations (i.e., kCFI [3], KCoFI [16], Fine-CFI [41], and the proposal from Ge et al. [23]) that solely rely on static analysis to protect backward control-flow edges leave the system vulnerable to the notorious Control-Flow Bending (CFB) [9] attack. CFB involves exploiting dispatcher functions to corrupt the return address using malicious arguments, bypassing the applied CFI protection. As emphasized by Carlini et al. [9] a shadow stack is necessary to fully protect return addresses and mitigate CFB. Besides shadow stack, Lilijestrand et al. [42] emphasized that ARM's Pointer Authentication (PA) [4] could also enhance protection against CFB attacks with dynamic runtime information, such as the current stack pointer, to validate return addresses.

Operation table pointers. Since PA-based kernel mitigations (i.e., PAL [62] and PATTER [68]) do not protect operation table pointers they are susceptible to pointer-to-pointer attacks, where an attacker corrupts operation table pointers instead of function pointers. However, their design can be extended to also protect operation table pointers. Mitigations that provide static control-flow integrity (marked with ■ or □ in Table 1) implicitly protect operation table pointers on indirect branches. Hence, if the function pointers are protected, the operation table pointers are also protected.

Function pointers. Mitigations that provide coarse-grained (i.e., KCoFI [16]) or signature-based (i.e., Fine-CFI [22] and kCFI [3]) protection for forward control-flow edges offer weak security guarantees. The large set of targets matching the function signature in the kernel space enables privilege escalation while not violating the over-approximated CFG, as demonstrated in Appendix 12.3. Moreover, Camouflage [18] only protects a selected set of function pointers, leaving unprotected vulnerable for privilege escalation.

Summary. Existing kernel mitigations exhibit limitations in providing protection for control data, particularly the thread state during system events and return addresses. As a consequence, attackers can exploit these limitations to bypass the applied CFI-based countermeasure. This results in a gap in kernel security.

4 DESIGN

We introduce Hardware-Enforced Kernel Control-Flow Integrity (HEK-CFI) which consists of three key mechanisms. Figure 3 depicts a high-level overview. First, HEK-CFI provides kernel control-data integrity (see Section 4.1) by retrofitting write-protected pages from Intel CET SHSTK for the *supervisor*. This retrofitting enables write-protected local and global safe areas within the kernel where control data is securely stored. Notably, these safe areas extend well beyond the original purpose of Intel CET SHSTK. Second, HEK-CFI utilizes our control-data integrity to protect the thread state (see Section 4.2) during all state-changing system events, i.e., interrupt, exception, syscall, and context switch, as well as protect the SHSTK state. This allows HEK-CFI to mitigate our motivational exploit examples and Google Project Zero's thread state exploit [56]. With these two key mechanisms, we are the first to provide comprehensive protection for both system events and return addresses. Third, HEK-CFI combines our control-data integrity with signature-based

CFI to protect (see Section 4.3) forward control-flow edges, particularly control data, i.e., function pointers and operation table pointers. While signature CFI efficiently protects function pointers with rare signatures, control-data integrity offers full protection for any control data, albeit at a potentially higher performance overhead. To optimize the trade-off, it automatically selects the optimal scheme for each control data based on a user policy as input.

While the real contribution of HEK-CFI lies in the efficient protection of control data (including thread state), the third mechanism demonstrates the practicality of protecting forward edges as well.

4.1 Kernel Control-Data Integrity

Its main concept is to store control data (outlined in Section 3.2) in a hardware-enforced write-protected safe area, preventing any malicious tampering attempt. To access the control data legally, the safe area needs to be designed differently depending on the context, such as local or global. For this reason, HEK-CFI includes a per-thread local and a global safe area storing and protecting the local and global control data, respectively.

Write protection. Our control-data integrity scheme relies on the security of the write-protected safe areas. To achieve this protection, all safe area pages are marked as shadow pages, retrofitting the approach followed by CETIS [66] to the kernel. This enforces Intel CET SHSTK to write-protect them. To legally write to the safe areas, HEK-CFI utilizes the `wrssq` instruction introduced by Intel CET, which permits writes to shadow pages. If a memory write operation to a shadow page is propagated by a non-shadow stack instruction, e.g., `movq`, it causes a control-protection exception raised by Intel CET. There is also no `wrssq`-gadget allowing an attacker to illegally write to shadow pages as we later discuss in Section 7. To conclude, with this approach, it not possible to illegally manipulate control data stored in the safe areas using the arbitrary write primitive.

4.1.1 Local Safe Area. HEK-CFI introduces a local safe area for each thread in kernel space. The local safe area consists of a shadow stack via Intel CET and a Protected Thread Local Storage (PTLS), as illustrated in Figure 4. Intel's hardware feature CET SHSTK utilizes the shadow stack to implicitly push and pop specific control data onto the shadow stack, as explained in Section 2. However, since Intel CET does not provide explicit push or pop operations [30], HEK-CFI introduces the software-based approach PTLS, which is an efficient and secure local storage. The PTLS is located at the bottom of the local safe area and provides `ptls_push` and `ptls_pop` routines corresponding to a safe push and pop operation. Hence, PTLS provides strong protection for locally accessible control data.

Listing 1 illustrates the `ptls` struct (PTLS) and the associated initializing, pushing, and poping routines. At its core, all of these routines use the macros `rd_ptls` (Line 7) and `wr_ptls` (Line 5) to obtain the `ptls` struct's location and to write to the underlying shadow page memory of the `ptls` struct, respectively. The macro `rd_ptls` does so by performing a logical AND operation of the read shadow stack pointer (Line 8) with `(LSA_SZ-1)`, where `rdsspq` reads the current shadow stack pointer, and `LSA_SZ` represents the local safe area size. The macro `wr_ptls` executes the `wrssq` instruction to write an 8 byte word to the top of the `ptls`. The `ptls_init` routine initializes the `ptls` struct by obtaining its current location with `rd_ptls`. Then, `ptls_init` sets the top of stack

```
1  struct ptls {
2    u64 tos; /* top of stack */
3    u64 data[]; /* data array */
4  };
5  #define wr_ptls(tos, d) \
6    asm("wrssq %0,(%1)"::"r"(d),"r"(tos))
7  #define rd_ptls(ptls) \
8    asm("rdsspq %0\n" \
9      "andq $~(LSA_SZ-1),%0":"=r"(ptls))
10
11 void ptls_init(void) {
12   struct ptls *ptls;
13   rd_ptls(ptls);
14   wr_ptls(&ptls->tos, (u64)ptls->data);
15 }
16 void ptls_push(u64 data) {
17   struct ptls *ptls;
18   rd_ptls(ptls);
19   wr_ptls(ptls->tos, data);
20   wr_ptls(&ptls->tos, ptls->tos+8);
21 }
22 u64 ptls_pop(void) {
23   struct ptls *ptls;
24   rd_ptls(ptls);
25   wr_ptls(&ptls->tos, ptls->tos-8);
26   return *ptls->tos;
27 }
```

Listing 1: PTLS's provided routines.

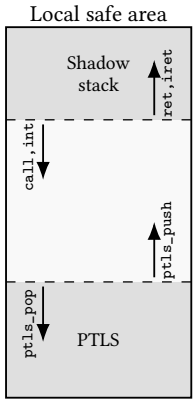

Figure 4: Memory layout of the local safe area for each thread, where Intel CET SHSTK implicitly resides on the top and PTLS on the bottom.

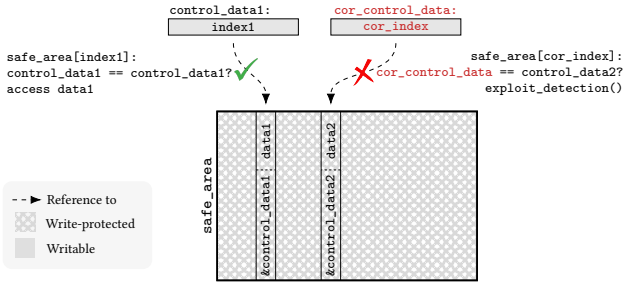

Figure 5: Two accesses from a control data stored on the global `safe_area`. HEK-CFI validates that the address of the accessed control data matches the stored address referenced by the index. Since the validation of `control_data1` succeeds, `data1` access is granted. Contrary, the validation of the corrupted `cor_control_data` fails, detecting an exploit attempt.

member variable `ptls->tos` to the `&ptls->data[]`, indicating that the `ptls` is empty and is ready for the first push operation. The other two routines, `ptls_push` and `ptls_pop`, perform push and pop operations to or from the `ptls`, respectively.

4.1.2 Global Safe Area. HEK-CFI presents a global safe area for control data that are accessible in a global context. These globally accessible control data store an index referencing the global safe area where the actual data is stored. We design our global safe area as a one-to-one mapping, meaning that each globally accessible control data has its own safe storage within the global safe area.

HEK-CFI provides allocation and deallocation routines to allocate and deallocate a global safe storage for the control data. To mitigate forgery attempts, we uniquely bind the control data to its safe

storage by storing its address along with the actual protected data within the safe storage on allocation. On deallocation, we unbind the control data with the safe storage and mark the safe storage as free. Figure 5 exemplifies a globally accessible control data with `control_data1`, storing an `index1` index that references the safe storage (`safe_area[index1]`) within the global safe area. This storage comprises the control data address `&control_data1` and the actual protected data `data1`.

HEK-CFI provides read and write routines to access the global protected data. It ensures protection against malicious access by verifying the control data's integrity before the access. This verification involves checking whether the accessed control data's address matches the stored address of the referenced safe storage. Access is granted on match; otherwise, it is considered an exploitation attempt. We illustrate both cases in Figure 5, where the validation check for the valid control data `control_data1` succeeds, while it fails for the corrupted control data `cor_control_data`.

4.2 Thread State Protection

Compared to user space programs, the kernel processes synchronous (i.e., exceptions and syscalls) and asynchronous (i.e., interrupts) system events which store the thread state in memory and handles invoked execution request (see Sections 4.2.1 and 4.2.2). Furthermore, at each context switch (see Section 4.2.3), the kernel stores the thread state in memory while restoring the state of the next thread to execute. By not fully protecting the stored state, an attacker can perform attack scenarios described in Section 3.2, as demonstrated in the examples given in Appendix 12 and the end-to-end attack exploiting CVE-2019-2215 in Appendix 13. To mitigate these scenarios, HEK-CFI uses our kernel control-data integrity to effectively protect the stored thread state. More precisely, HEK-CFI protects the thread state for interrupts, syscalls, and exceptions by storing it within the local safe area and for the context switch within both the global and local safe area.

4.2.1 Interrupt and exception. The hardware pushes `ss`, `rsp`, `rflags`, `cs`, and `rip` to the current stack or value specified in the Interrupt Stack Table (IST) on kernel entrance and re-entrance. We refer to the current stack and the value specified in the IST as data stack. The kernel then pushes the general-purpose registers (except `rsp`) to the data stack. With Intel CET SHSTK enabled, the hardware pushes `cs`, `rip`, and `ssp` to the shadow stack atomically to the data stack push. On `iret` instructions, the hardware validates that `cs` and `rip` are equal to the ones stored on the shadow stack, where a mismatch causes a control-protection exception.

Protecting `cs` and `rip` is insufficient to ensure protection as an attacker may tamper general-purpose registers stored on the data stack, illustrated in attack scenario ⑥ in Figure 1. To mitigate this, HEK-CFI stores the register values within the PTLS on interrupt and exception entrances before pushing them to the data stack. On interrupt and exception exits, the registers are popped from the data stack and then compared with those stored in the PTLS. HEK-CFI interprets a mismatch as an exploitation attempt.

To mitigate potential TOCTTOU attacks, HEK-CFI never stores the registers to unprotected memory during state storing (① in Figure 1). HEK-CFI protects all register values, as shown in Listings 2 and 3. On the request entrance routine `rq_entry`, HEK-CFI

```
1  rq_entry:
2   /* safe store r15 */
3   wrssq r15, R15(gs)
4   /* r15 = ptls->tos */
5   rdsspq r15
6   andq  $~(LSA_SZ-1), r15
7   movq  r15, (r15)
8   /* store all regs to ptls */
9   wrssq r14, R14(r15)
10  wrssq r13, R13(r15)
11  ...
12  wrssq rdi, RDI(r15)
13  /* store r15 to ptls */
14  wrssq r14, R14(gs)
15  movq  r15, r14
16  movq  R15(gs), r15
17  wrssq r15, R15(r14)
18  movq  R14(gs), r14
```

Listing 2: Safe stores all registers to the current `ptls`.

```
1  rq_exit:
2   /* safe store r15 */
3   wrssq r15, R15(gs)
4   /* r15 = ptls->tos */
5   rdsspq r15
6   andq  $~(LSA_SZ-1), r15
7   movq  r15, (r15)
8   /* validate all regs */
9   cmpq  r14, R14(r15)
10  jne   .fault
11  ...
12  cmpq  rdi, RDI(r15)
13  jne   .fault
14  /* validate r15 */
15  movq  r14, R14(gs)
16  movq  r15, r14
17  movq  R15(gs), r15
18  cmpq  r15, R15(r14)
19  jne   .fault
20  movq  R14(gs), r14
```

Listing 3: Validates all registers to be equal to the stored ones on the `ptls`.

first temporarily stores `r15` to a per-CPU storage (Line 3), also write-protected with Intel CET SHSTK. Temporarily storing `r15` is essential because HEK-CFI requires one register for the upcoming execution. HEK-CFI then interprets the bottom of the current local safe area as `ptls` (Lines 5-6) and loads `ptls->tos` to `r15` in Line 7. Between Lines 9-12 all registers are stored to the `ptls`. Lastly, HEK-CFI protects register `r15` (Lines 14-18).

On request exit `rq_exit`, HEK-CFI performs a validation process that requires `r15` for the upcoming execution. Between Lines 9-13, HEK-CFI compares all registers with the protected register values stored within the `ptls`, jumping to `.fault` on a validation failure. We relax the constraints on interrupts and exceptions from user space: We guarantee to return to the exact location in user space but do not protect general-purpose registers. Hence, we apply `rq_entry`/`rq_exit` only if the thread was disrupted during kernel-space execution. Crucially, our relaxation is stricter than comparable CFI-based mitigations [23, 41, 62].

4.2.2 Fast syscall. The `syscall` instruction invokes a fast syscall to request kernel-space execution with supervisor privileges. The hardware saves the current `rip` to `rcx` and `rflags` to `r11` on this instruction. Next, it loads the `rip` from `MSR_IA32_LSTAR`, indicating the syscall entry location. On entry, the kernel software stores both register values, `rcx` and `r11`, on the data stack, while on completion, it restores both values and executes `sysret`. This instruction returns to user space by loading `rip` from `rcx`, `rflags` from `r11`, and user `cs` and `ss` from `MSR_IA32_STAR`.

To protect `rcx` and `r11` from being tampered with, HEK-CFI also stores both values within our provided PTLS, on kernel entrance. During kernel execution, when these user registers are legally modified, HEK-CFI also modifies the protected one. On completion, HEK-CFI validates that `rcx` and `r11` have not been tampered with.

In rare cases, the Linux kernel returns from a fast syscall to user space with `iret` instead `sysret`. HEK-CFI stores, for these cases, the `USER_CS` and `rcx` which represents `cs` and `rip`, to the shadow

stack. On `iret` Intel CET SHSTK compares both, `cs` and `rip`, from the data and shadow stack, where a mismatch raises an exception. Crucially, HEK-CFI never trusts any value stored in unprotected memory, as `USER_CS` is a constant and `rcx` was protected within PTLS. Since an attacker cannot corrupt either register, HEK-CFI protects the fast syscall event from being exploited.

4.2.3 Context switch. When a thread performs a context switch, it calls into the scheduler that selects a new thread to run next. Since the Linux kernel does not have a dedicated scheduler, the current thread switches the memory descriptor (including `cr3`) with the new descriptor and jumps to `switch_to`. This function stores the current and restores the next stack pointer, callee-saved registers, `fs`/`gs` registers (if required), and additional non-general-purpose registers, e.g., for debugging. Storing and restoring the instruction pointer is not needed as it was implicitly stored on the stack when the context switch function was called and is restored on return.

HEK-CFI also mitigates the attack scenario outlined ❻ from Figure 2 by protecting the shadow stack pointer with control-data integrity within the global safe area. Hence, the attacker cannot forge the shadow stack pointer. Since the control data of the data stack must match the control data on the shadow stack on every `ret` and `iret`, and the shadow stack pointer's integrity is ensured, corrupting the data stack pointer cannot be exploited.

In addition to the shadow stack pointer, HEK-CFI protects callee-saved registers as well as `fs` and `gs` within our PTLS upon a context switch event. HEK-CFI does not require explicit protection for instruction pointers on a context switch because Intel CET SHSTK implicitly protects the stored instruction pointer as a return address.

4.3 Control-Flow Integrity

In this section, we explain how HEK-CFI combines kernel control-data integrity with function signature CFI to protect the thread state, return addresses, and control data pointers, i.e., operation table and function pointers. Function signature CFI (see Section 4.3.2) provides efficient protection for function pointers with rare signatures. In contrast, control-data integrity (see Section 4.3.1) offers full protection for any control data, albeit with a potentially higher performance overhead. To optimize the trade-off, we present the control-data protection selector (see Section 4.3.3), which automatically selects what to protect with signature CFI and what with control-data integrity based on a user policy as input. This way, we achieve strong security without compromising performance.

4.3.1 Safe Area Usage. To protect control data pointers with the kernel control-data integrity approach, HEK-CFI instruments the kernel as follows: When control data pointers are generated, HEK-CFI allocates a safe storage and binds it to the control data. On control data destruction, the corresponding safe storage is unbound and deallocated. Depending on the context of the control data (i.e., global or local), HEK-CFI utilizes either the global or local safe area as safe storage. For the global safe area, HEK-CFI uses the provided allocation and deallocation routines directly. For the local safe area, HEK-CFI pushes a local safe storage to the PTLS, consisting of the actual protected data and the address of the local control data pointer, to uniquely bind the local safe storage to the control data. On deallocation, HEK-CFI pops the local safe storage from the PTLS.

When accessing control data pointers, HEK-CFI validates whether the index stored in the control data has been tampered with. For the global safe area, HEK-CFI uses the provided read and write routines directly, as shown in Figure 5. For the local safe area, HEK-CFI uses read and write routines similar to the global ones, but they reference the PTLS. We illustrate the instrumentation in Appendix 14.

Since HEK-CFI uses a one-to-one mapping to bind the control data pointers to the safe storage uniquely, functions such as `memcpy` may cause false positive exploitation attempts, as the copied stored index mismatches with its referenced address. To prevent false positives, we provide an instrumentation routine performing `realloc`. We carefully designed this routine to only copy the control data if the old safe storage is validated. Since the old safe storage was validated, this routine cannot be used to forge a safe storage.

4.3.2 Function-Signature Control-Flow Integrity. HEK-CFI provides control-flow transfer restriction with a function signature granularity by applying FineIBT [22] to the kernel. FineIBT leverages Intel IBT for coarse-grained control-flow integrity and builds a software-based control-flow restriction with function signature granularity on top of it. It stores the hash of a function pointer's signature into a register before redirection and validates the hash on function entry. A hash mismatch is interpreted as an exploitation attempt.

4.3.3 Control-Data Protection Selector. How much performance overhead is acceptable and how much is prohibitive depends on the use case. For high-security context, i.e., protecting all control data with control-data integrity, an elevated performance overhead may be acceptable. On the other hand, high-efficiency systems hardly accept any performance overhead. Since we envision HEK-CFI being suitable for all use cases, we provide a user policy that allows to choose the desired CFI precision level and, hence, overhead.

To accomplish this, our control-data protection selector analyzes the Linux kernel to identify control data pointers, i.e., function and operation table pointers, that require protection with control-data integrity based on the user policy. Dynamically allocated and global control data pointers are stored in the global safe area (see Section 4.1.2), while local control data pointers are stored in the PTLS (see Section 4.1.1). Regardless of the user policy, protection for the thread state and return addresses is always enabled.

User policy. The user policy comprises the number of permitted control-flow targets with function signature granularity. Our selector analyzes the Linux kernel and determines the number of possible control-flow targets matching the signature for each control data pointer. For function pointers, if the determined number exceeds the permitted control-flow targets, the selector annotates to protect the pointer with control-data integrity. For operation table pointers, the selector annotates to protect the pointer if one of its containing function pointers exceeds the permitted targets.

5 IMPLEMENTATION

In this section, we present the proof-of-concept implementation of HEK-CFI. We extend the Linux kernel and implement an LLVM pass [38] for instrumentation and the control-data protection selector using CodeQL [24] and Python. Our selector and instrumentation work automatically based on a user policy.

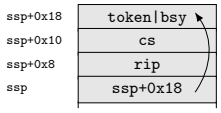

Figure 6: Required shadow stack layout for a valid `iret`.

5.1 Linux Kernel

We first enhance the Linux kernel to support all instrumentation routines. Next, we integrate write protection for the safe areas enforced with CET SHSTK. Lastly, we provide thread state protection.

Kernel instrumentation. We implement thread-safe functions for all kernel instrumentation routines described in Sections 4.1.1, 4.1.2 and 4.3.1, except for the read-access routines, as our compiler extension directly inserts an instruction sequence for the read-access routines, as we explain in Section 5.3. Moreover, we implement a routine that handles all fault scenarios of HEK-CFI, where our proof-of-concept detects and prevents exploitation attempts.

Protection through shadow stack. At the time of writing, there was no Linux kernel patch for the *supervisor* shadow stack[1]. Therefore, we integrate shadow stack as follows: Intel CET SHSTK supports per-thread supervisor and per-CPU IST shadow stacks. The supervisor shadow stack is used with most executions, while the IST shadow stack is used when the system requires the IST stack. To enable shadow stack usage, we first set the per-CPU IST shadow stacks (`MSR_INT_SSP_TAB`) to a table of shadow stacks and the supervisor shadow stack (`MSR_PL3_SSP`) to the per-thread shadow stack page. We then set bit `X86_CR4_CET` in the `cr4`, and `CET_SHSTK_EN | CET_WRSS_EN` in the `MSR_S_CET`. Next, we prepare all shadow pages for the `setssbsy` instruction accordingly [31], which sets the shadow stack pointer register to the value specified in the `MSR_PL0_SSP`. The `setssbsy` instruction requires the shadow stack to be not busy, while it marks the stack as busy. The busy flag is stored in the memory location `MSR_PL0_SSP` points to.

To mark a page as a shadow stack, its permissions must be dirty and non-writable. If a page is marked as shadow stack, only shadow stack write instructions, e.g., `wrssq`, are permitted to write this page. Otherwise, Intel CET raises a control-protection exception on a write operation propagated from a non-shadow stack instruction, e.g., `movq`. Moreover, if a shadow stack instruction writes to a non-shadow page, Intel CET also raises a control-protection exception.

Our proof-of-concept enables the shadow stack during kernel initializations before SMP is enabled, and only the `init_task` runs. Hence, the shadow stack is only set for the `init_task`. Enabling the shadow stack must occur in the early stage of kernel initialization, as our control-data integrity scheme relies on its security mechanism.

We extend the user thread creation routine to allocate one shadow stack page, mark its permission to dirty and non-writable, initialize the PTLS, and prepare the shadow stack for the `iret` to start execution in user space. Figure 6 depicts the required shadow stack layout to perform a valid `iret` [30], where `ssp` is the shadow stack pointer, `token` is the supervisor token, and `bsy` is the busy flag indicating whether the shadow stack is in use. On `iret`, the hardware validates that `rip` and `cs` are equal between data and shadow stack. If they are equal, the hardware sets the `rip` and `cs` accordingly. It then sets the shadow stack pointer to `ssp+0x18` and resets the `bsy` flag on the user space transition. We also extend the kernel thread creation, which closely assembles the user thread creation.

Since each thread has its shadow stack, the shadow stack pointer has to be stored and restored on every context switch. Hence, we implement the context switch in assembly as described in Section 4.2.3. When restoring the next shadow stack pointer, we store a non-busy token temporarily in the next shadow stack, which is used to switch the shadow stack. The switch occurs with the `setssbsy` instruction. Afterward, we reverse the temporary storing of the token.

Protection through IBT. We use the inofficial FineIBT patch [22, 46] for the control-flow restriction with signature granularity.

Interrupts and exceptions. We implement a safe storing routine for all general-purpose registers on interrupt and exception entries, as illustrated in Listing 2. Crucially, `rq_entry` is executed before the Linux kernel pushes the registers to the data stack via `PUSH_ALL_REGS`. Since `rq_entry` requires the register `r15` for the upcoming execution, HEK-CFI temporarily stores it on a per-CPU page in the `per_cpu` section. The per-CPU page's permissions are set as a shadow stack page. To support nested interrupts, HEK-CFI increases the `ptls->tos` by `sizeof(struct regs)` after both `rq_entry` and `PUSH_ALL_REGS`. We implement the validation routine `rq_exit` on interrupt and exception exits, shown in Listing 3.

Fast syscalls. Intel CET resets the supervisor shadow stack pointer on a privilege level change from user to kernel space via a fast syscall (i.e., `syscall` instruction) [31]. Hence, we extend the kernel entrance for a fast syscall to set the shadow stack accordingly. On rare occasions where the kernel returns via the `iret` instruction instead of `sysret` to the user space, HEK-CFI prepares the shadow stack to perform a valid `iret`, shown in Figure 6.

5.2 Control-Data Protection Selector

Our control-data protection selector has two main components: A code analyzer and a parser. We use CodeQL [24] as our code analyzer and a Python script as our parser. CodeQL compiles the Linux kernel to create a database that stores essential meta information. We run our CodeQL queries using the database to retrieve relevant information for our mitigation. Then, our parser interprets the query results based on the user policy and generates an output file containing all the information for kernel instrumentation.

Code analyzer. Our CodeQL queries first find all control data pointers, e.g., members within a struct or its containing struct or standalone pointers. The queries then determine the number of functions matching the function pointer's signature or any within an operation table. Each function pointer retrieves a score of matching functions, while each operation table pointer retrieves the highest number of its containing function pointers. The queries also determine all occurrences of control data pointers via allocation, deallocation, global variable, or local variable.

Parser. Our parser inputs the query results and the user policy (i.e., maximum permitted control-flow targets with signature granularity). It then filters out control data pointers with fewer targets than the user policy allowed and saves the remaining pointers to a file. This file represents all pointers protected with control-data integrity. At this point, a user can modify the file if necessary.

[1]Xen hypervisor v4.15-unstable [11] has integrated Intel CET SHSTK for supervisor, but it significantly varies from the Linux kernel integration.

5.3 Compiler Extension

Our compiler extension (LLVM pass) inserts validation checks into the kernel, ensuring HEK-CFI's functionality.

Control-data integrity. To ensure control-data integrity, the LLVM pass first examines kernel code for operations such as allocations, deallocations, reallocation, reads, and writes of control data pointers, annotated in the file generated by our control-data protection selector. Subsequently, the pass modifies the code to include instrumentation routine accessing the control data, where Appendix 14 demonstrates the actual instrumentation. For allocation, deallocation, reallocation, and write access, the pass inserts function calls provided by our kernel extension. The code is modified for read access to perform a performance-trimmed instruction sequence executing the safe storage read-access. To protect global variables, HEK-CFI identifies all protected global control data pointers, whether standalone or within structs, and allocates their safe storage during the early stage of kernel initialization.

Similar to global variables in kernel code, global variables within modules must also be initialized so that protected control data pointers reference a safe area. Hence, HEK-CFI identifies these global variables and inserts initialization routines on module insertion.

Signature CFI. We utilize the inofficial FineIBT patch [46] in our LLVM pass to ensure control-flow restriction with signature granularity. This patch instruments the caller and callee sites involved in each indirect forward-edge transfer.

6 CASE STUDY

This section demonstrates HEK-CFI's effectiveness in reducing forward control-flow targets. HEK-CFI achieves this by protecting operation table pointers and function pointers with a common function signature from being overwritten through our kernel control-data integrity scheme (see Section 4.1). Additionally, it restricts the control-flow targets of function pointers with rare function signatures at the granularity of signatures. By reducing the permitted targets of control-data-integrity-protected pointers to 1 and limiting signature-restricted pointers to the set of functions matching the pointers' signature, HEK-CFI effectively lowers the overall average forward control-flow targets. This case study demonstrates that HEK-CFI achieves a forward target reduction of more than 50 % over comparable kernel CFI schemes [3, 16, 23, 41, 46] while having a lower performance overhead than these schemes, as we later evaluate in Section 8. Notably, HEK-CFI is also the first to protect return addresses and thread states comprehensively.

In the following, we describe how we compute HEK-CFI's target reduction with other kernel CFI schemes. We first discuss the *Average Indirect targets Allowed* (*AIA*) metric, which allows us to quantify CFI precision levels. We then determine the *AIA* for various CFI precision levels, including HEK-CFI. Finally, we demonstrate that HEK-CFI outperforms the security guarantees of forward control-flow target reduction compared to CFI schemes.

Quantify CFI precision level. Zhang et al. [70] proposed the metric *Average Indirect Reduction* (*AIR*) to evaluate the effectiveness of CFI-based mitigations. However, for large binaries, e.g., the Linux kernel, the *AIR* may be misleading [9, 23] because in these cases, an *AIR* of more than 99 % stills permits 10 k of control-flow targets for each indirect control-flow redirection. Subsequently,

Table 2: Function pointers protected with kernel control-data integrity (see Section 4.1).

	Total	Within operation tables[1]	Plain[2]
Total	6487	3033	3454
Protected	1662	1235	427

Stored in [1]read-only sections and [2]writable sections.

Ge et al. [23] proposed *AIA* as an improved evaluation metric to quantify the effectiveness of CFI-based mitigations.

$$AIA = \frac{1}{n}\sum_{i=1}^{n} |T_i| \tag{1}$$

Equation (1) shows the quantifier, where n is the number of indirect calls within the entire binary, and T is the set of permitted control-flow targets. We consider three cases for our CFI precision analysis: Coarse-grained (reachable function granularity) and fine-grained (function signature granularity) CFI policy, and our HEK-CFI.

Analysis. The setup of our case study is the Linux kernel v5.18 running with the configuration of Ubuntu 22.04.1 LTS. We perform a manual analysis revealing that the entire kernel code, including all modules, comprises 6487 function pointers (outlined in Table 2). 3454 of them are stored in writable sections and 3033 are stored within read-only operation tables, whereas the kernel has 300 operation table pointers. We determine that the kernel comprises a total of 140534 control-flow targets for coarse-grained CFI (without any unaligned `endbr64`). Next, we obtain information for each indirect call and the function pointer's permitted targets with signature granularity. Using these insides and Equation (1), we calculate a coarse-grained AIA_{cg} of 140534 and a fine-grained AIA_{fg} of 325.

Target reduction with HEK-CFI. We configure HEK-CFI via user policy to enforce an $AIA_{hek-cfi}$ that is lower than comparable kernel CFI schemes [3, 16, 23, 41, 46]. To achieve this, a developed tool (see Appendix 15) determines the user policy as 190 maximum permitted control-flow targets. With this user policy, HEK-CFI's control-data protection selector automatically determines that out of 3454 function pointers stored in writable sections, 427 exceed the permitted control-flow targets of 190. Hence, HEK-CFI protects these function pointers with control-data integrity, as outlined in Table 2. Moreover, among the 300 operation table pointers, 88 contain at least one function pointer exceeding 190 targets. Consequently, HEK-CFI ensures the integrity of these 88 operation table pointers. Since the operation tables referenced by the 88 protected pointers are read-only, their containing 1235 function pointers are implicitly protected from manipulation. Overall, our case study protects 1662 function pointers reducing their control-flow target to 1.

$$AIA_{hek-cfi} = \frac{1}{n}\left(\sum_{i=1}^{p} 1 + \sum_{i=p+1}^{n} |T_i|\right) \tag{2}$$

We adapt Equation (1) to Equation (2), where p is the number of protected function pointers, i.e., 1662. We then compute $AIA_{hek-cfi}$, resulting in 22 average permitted control-flow targets. To evaluate HEK-CFI's effectiveness, we compute the improvement over coarse- and fine-grained CFI, showing a permitted target reduction of 99.98 % over coarse- and 93.3 % over fine-grained CFI.

Table 3: Permitted control-flow target reduction of HEK-CFI over state-of-the-art mitigations.

	AIA	*AIR*	HEK-CFI's improvement[1]
	-	%	%
KCoFI [16]	140534	-	99.98
FineIBT [46], kCFI [3]	325	-	93.3
Ge et al. [23]	92	-	76.4
Fine-CFI [41]	-[2]	99.999740	50.4
HEK-CFI	22	99.999871	-

[1] HEK-CFI's control-flow target reduction over the other mitigation.

[2] Not enough information provided to compute the *AIA*.

Comparison to existing CFI schemes. As described by Ge et al. [23], directly comparing CFI precision metrics is inaccurate for several reasons, such as varying kernels and kernel configurations. Hence, for the following comparison, we relatively compare various designs [3, 16, 23, 41, 46] as if their mitigation would have protected our kernel binary, with the results shown in Table 3.

KCoFI provides coarse-grained CFI, resulting in an AIA_{KCoFI} of 140534. Due to our $AIA_{hek-cfi}$ of 22, we reduce targets by 99.98 %. In both fine-grained mitigations, FineIBT and kCFI, average permitted targets (AIA_{fg}) are 325, while our mitigation reduces these targets by 93.3 %. Ge et al. claimed that their proposal eliminates 71.8 % of signature-based CFI schemes. We assess the CFI precision level of their proposal by reducing the AIA_{fg} by 71.8 %, resulting in an AIA_{ge} of 92. With an $AIA_{hek-cfi}$ of 22, we improve by 76.4 %.

Since Fine-CFI only provides *AIR* as precision level, we compute the *AIR* for HEK-CFI and Fine-CFI for our kernel binary.

$$AIR_{hek-cfi} = \frac{1}{n}\sum_{i=1}^{n}\left(1 - \frac{|T_i|}{S}\right) \tag{3}$$

$$AIR_{fine-cfi} = 1 - \frac{|T|}{S \cdot n} \tag{4}$$

We use Equation (3) [70] to compute HEK-CFI's $AIR_{hek-cfi}$ and their adapted Equation (4) [41] for Fine-CFI's $AIR_{fine-cfi}$, where T is the set of permitted control-flow targets, n is the number of indirect calls, and S is the kernel binary size. The results are 99.999 871 % for $AIR_{hek-cfi}$ and 99.999 74 % for $AIR_{fine-cfi}$, showing HEK-CFI's permitted target reduction of 50.4 %.

PAL [62] provides too little information to perform a reasonable comparison. Since we neither have any information about their *AIA* nor *AIR*, a direct comparison with our case study is not possible.

Manual efforts. In rare cases, LLVM's frontend may not store variable type information, which can cause our implemented LLVM pass to miss control data pointer uses. We manually inserted validations into the kernel code to address these rare false negatives (less than 0.3 %). We emphasize that these false negatives come from neither our design nor our implementation but rather from the limitations of the LLVM frontend. Therefore, we still refer to our framework as automated.

Even if the compiler missed inserting an instrumentation routine, this would result in a fault as the control data contains an index instead of the data. Similarly, if the protection selector failed to find a control-data instance, the instrumentation routines would interpret the control data as an index, resulting in a false exploit detection. During our evaluation, we encountered no such scenarios except for the false negatives caused by the LLVM frontend, which we manually fixed. Even if false negatives were present, we anticipate they could not be exploited, as they would result in a fault.

7 SECURITY DISCUSSION

Improvement over existing works. HEK-CFI improves security over existing works, as outlined in Table 1. For instance, only KCoFI provides comprehensive thread state protection among the kernel CFI schemes. Thus, all schemes except KCoFI fail to mitigate all of our motivational exploitation attacks presented in Appendix 12, as well as the Google Project Zero's thread state exploit (cf. CVE-2022-42703) [56]. In contrast, HEK-CFI successfully mitigates them. Although a combination of solutions such as KCoFI, Fine-CFI, and Intel CET SHSTK could theoretically offer similar protections for thread state and return addresses, HEK-CFI still improves forward edge protection (see Table 3) with substantially lower performance overhead. In particular, HEK-CFI reduces the CFI targets by over 50 % (see Section 6) compared to the combined forward CFI precision of these mitigations, represented by Fine-CFI's enhanced precision. Due to the unavailability of these solutions for empirical comparison, we estimate their combined overhead to be significantly higher than that of HEK-CFI. Specifically, HEK-CFI's performance overhead for macro benchmarks is about 1.85 %, as we will evaluate later in Section 8, whereas the individual overheads for KCoFI and Fine-CFI are around 10 % each. Similarly, when comparing HEK-CFI with a theoretical combination of KCoFI, Ge et al.'s solution, and SHSTK, HEK-CFI reduces targets by more than 76 %, representing the enhancement over Ge et al.'s solution. Like the previous example, we expect the combined overhead of these solutions to be significantly higher, with KCoFI and Ge et al.'s solutions individually contributing overheads of 10 % and 2 %, respectively.

Attacking control-data integrity. An attacker attempts to perform the following attack scenarios on kernel control-data integrity. Firstly, they overwrite the index stored in place of control data with an index referencing other control data. When the control data is accessed, HEK-CFI verifies that the control data's address matches the stored address in the safe storage. HEK-CFI interprets a mismatch as an exploit attempt. Secondly, they overwrite the index to reference outside the safe area. HEK-CFI performs a bounds check and, hence, detects the exploit attempt. Thirdly, they tamper with the global or local safe areas directly. However, the safe area is write-protected by Intel CET and any write access by non-shadow stack operations results in a control-protection exception.

Confused deputy attack. A confused deputy attack [39] tricks a high-privilege routine to perform a write operation to protected data. Since only shadow stack instructions, i.e., `wrssq`, are permitted for writing to shadow stack pages, we identify the following high-privilege routines that may be targeted: Instrumentation routines for the safe areas, i.e., `rq_*` and `ptls_*`. However, these routines cannot be exploited for a deputy attack as they only write to the safe area if the access was validated. Furthermore, these routines only write to the current shadow stack or per-CPU storage. Since an attacker can neither control the shadow stack pointer nor the per-CPU storage is mapped, these routines can also not be used for a deputy attack. Overall, HEK-CFI effectively mitigates the confused deputy attack, preventing the existence of a `wrssq`-gadget.

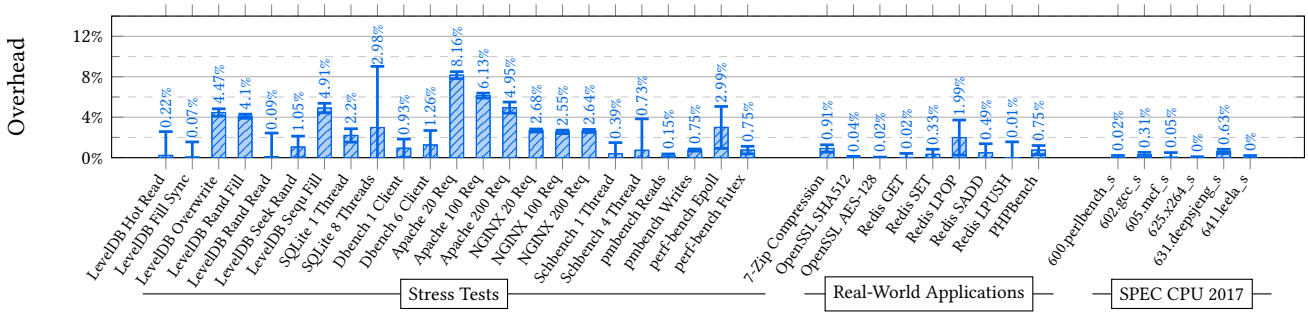

Figure 7: Macro benchmark results.

Thread state. The Linux kernel processes system events that store the thread state in memory, as illustrated in Figures 1 and 2. Directly overwriting the state, e.g., `rip` or `cs`, on the data stack ⑤ will cause a control-protection exception on `iret` as Intel CET validates that the `rip` on data and shadow stack is equal. Directly tampering with pages marked as shadow stack also leads to a control-protection exception. Moreover, an attacker may manipulate general-purpose registers ⑥ stored in memory. For instance, the attacker overwrites a register stored on the data stack that will be used for a control-flow transfer later during execution. However, HEK-CFI stores all general-purpose registers within the PTLS and validates the registers to be equal on state restoration. Hence, HEK-CFI detects the tampering attempt. In attack scenario ❺, an attacker attempts to tamper with the thread state restored on a context switch event. However, since HEK-CFI protects the stored thread state within PTLS and the shadow stack state to the global safe area, each corruption attempt is detected. By manipulating any of these safe areas, Intel CET raises a control-protection exception which HEK-CFI interprets as an exploitation attempt.

Architectural-defined control data. The Linux kernel has various architectural-defined control data that may be targeted for control-flow hijacking attacks. x86 CPUs contain Interrupt Descriptor Tables (IDT), Global Descriptor Tables (GDT), and Local Descriptor Tables (LDT) storing security-critical system configurations. The IDT stores sensitive information, such as interrupt entry locations, CPU privilege level on interrupt entry, and used stack. However, corrupting the IDT is not possible on recent Linux versions because the kernel maps it as read-only. Both descriptor tables, GDT and LDT, store information about memory segmentations, such as the code segment. The code segment determines the CPU privilege level on interrupt (and exception) entry and exit, as well as the base address used by indirect and direct control-flow transfers [23]. By tampering with the descriptor tables, an attacker may change data of segments, e.g., code segment. HEK-CFI maps the GDT as a shadow stack page to prevent descriptor table corruption and uses the `wrssq` instruction for write operations. The LDT could be protected similarly to the GDT, with its address protected with control-data integrity. However, in our proof-of-concept, we compiled the Linux kernel with `CONFIG_MODIFY_LDT_SYSCALL` reset, not supporting LDT. This may affect compatibility with older user applications, but we encountered no issues during the evaluation.

Control-Flow Bending. Control-Flow Bending (CFB) [9] is an attack that exploits so-called dispatcher functions that corrupts their own return address using malicious arguments. Even with a fully-precise static CFI scheme [9], commonly used functions have a large set of permitted backward edges that can be used for a CFB attack. Thus, CFI-based mitigations that do not fully protect return addresses can be bypassed by CFB attacks [23]. However, HEK-CFI includes a shadow stack, which prevents dispatcher functions from corrupting their return address and effectively mitigates CFB.

Pointer-to-pointer corruption. HEK-CFI is designed to protect control data from being corrupted. However, an attacker may corrupt a non-control data pointer to a control data, e.g., the attacker may corrupt a pointer within a list of inodes (`list_head`) to forge an `inode`. Fortunately, our control-data integrity scheme provides generic protection that can also be applied to protect non-control data pointers. We acknowledge that identifying an appropriate set of non-control data pointers to protect is a subject that requires further research and is an area we plan to explore in future work.

Stack tampering. Xu et al. [67] recently presented their novel WarpAttack attack targeting CFI schemes. They demonstrated that compiler optimizations may introduce double-fetch vulnerabilities when registers containing control data are spilled to the stack, making them vulnerable to corruption attacks. Fortunately, HEK-CFI's PTLS can prevent against WarpAttack by storing these registers within PTLS instead of pushing them to the stack. While WarpAttack protection is possible, it is currently out of scope as extending the compiler stage would require significant engineering effort.

8 PERFORMANCE EVALUATION

We evaluate our proof-of-concept's runtime overhead by performing micro benchmarks with LMbench [45], and macro benchmarks with Phoronix Test Suite [49] and SPEC CPU 2017 [15]. We observe an overhead of 12.3 ± 1.5% for micro benchmarks, while macro benchmarks from Phoronix and SPEC increase the overhead by 1.85 ± 1.02% and 0.17 ± 0.23%. We run our proof-of-concept on Ubuntu 22.04.1 LTS on the Intel Alder Lake processor i7-12700k, supporting Intel CET. We also evaluate the compile time, binary size, and analyzer overhead in Appendix 16.

Micro benchmarks. We use LMbench to evaluate the latency and bandwidth overhead. We consider the Linux kernel v5.18 baseline and our proof-of-concept HEK-CFI enhanced one. We run each benchmark 80 times and compute the geometric mean and standard deviation. Table 4 illustrates the evaluation results for HEK-CFI, with the geometric mean being 12.3 ± 1.5%.

Phoronix Test Suite. We split our benchmarks from Phoronix Test Suite into stress tests and real-world applications, as illustrated in Figure 7. Among the stress tests are three database, two webserver, one scheduler, one virtual paging, and one performance tool benchmarks. Since the webserver and database benchmarks use a lot of control-data pointers with common signatures, e.g., `void (*)(struct sk_buff *)` and `void (*)(struct inode *)`, these benchmarks elevate the overhead between 0.01 % to 8.16 %. The Schbench benchmark illustrates that HEK-CFI has little impact on scheduling performance as the caused overhead is between 0.39 % to 0.73 %. Among the real-world applications are one in-memory database, and three user applications, having a performance overhead between 0.01 % to 1.99 %. The computed geometric mean of all Phoronix Test Suite macro benchmarks is 1.85 ± 1.02%.

We observe an elevated standard deviation for benchmarks, particularly multi-threaded ones. However, since these are present in both kernel binaries, the baseline and HEK-CFI-enhanced, they are caused by the noisy kernel properties, e.g., interrupts.

SPEC CPU 2017. We perform various speed SPEC CPU 2017 macro benchmarks, as illustrated in Figure 7. The resulting overheads are with a geometric mean of 0.17 ± 0.23%, in line with the results of the real-world applications from Phoronix Test Suite.

9 RELATED WORK

FineIBT [22, 46] provides effective protection for function pointers with rare function signatures, as they enforce control-flow integrity with function signature granularity while having little impact on the performance [22, 59]. However, FineIBT does not protect return addresses. kCFI [3] protects forward-edge control-flow transfers similarly to FineIBT, but instead of relying on Intel's IBT hardware feature, it uses Clang's software solution. PATTER [68] uses ARM's Pointer Authentication (PA) [4] to sign and authenticate function pointers and return addresses and, hence, protects against tampering of these two control data but does not protect operation table pointers. Camouflage [18] protects selected function and operation table pointers, and return addresses, with ARM PA. All four proposed mitigations [3, 18, 46, 68] do not protect the thread state.

KCoFI [16] is a coarse-grained CFI scheme with protection for the thread state. For forward-edge control-flow transfers, they only reduce the targets by 98.18 %, and for backward transfers, KCoFI enforces that a function return must land at one of the call sites that could have called it. Moreover, KCoFI has a performance overhead of above 10 % and 100 % for macro and micro benchmarks, respectively.

Ge et al. [23] proposed a method of retrofitting kernel software to provide fine-grained CFI. They demonstrated that their FreeBSD proof-of-concept reduces performance overhead and control-flow targets compared to a function signature-based CFI scheme. Their proof-of-concept implementation has a reasonable performance overhead of around 2 % for macro benchmarks. To protect against attack scenarios ⑤ and ⑥ from Figure 1, they disable preemption during kernel space execution. The kernel may also raise a page fault exception on common occasions, e.g., `copy_from_user`, which they address by storing the `rip` to an unused debug register on exception entry and validating on exit. However, as this debug register is stored in writable memory on a context switch (and they do not protect the thread state on a context switch ❺), an attacker can corrupt the memory to gain control of the register on restoration. Subsequently, the instruction pointer is set to the corrupted debug register (⑤) on exception exit, hijacking the control flow.

Li et al. [41] proposed Fine-CFI that reduces the indirect control-flow targets over previous CFI schemes [16, 23]. It induces the overhead by about 10 % and 8 % for macro benchmarks from Phoronix and SPEC, respectively. Moreover, Fine-CFI insufficiently protects the thread state on an `iret` instruction as it only validates the `rip` and `cs`. This leaves the stored state unprotected for attack scenarios ⑥ and ❺ from Figures 1 and 2, respectively.

PAL [62] is a kernel CFI-based defense that uses ARM PA to protect function pointers and return addresses with 1 % to 5 % overhead for macro benchmarks. However, their measured overheads must be interpreted cautiously, as they fluctuate by up to 200 %, as reported in their appendix. PAL does not protect operation table pointers and insufficiently protects the thread state. On a preemption, the kernel stores the registers in memory, and PAL computes a signature of all registers, including a time-based nonce. Then, PAL stores the nonce and signature in memory. After preemption, it verifies the registers to ensure they have not been corrupted. We identify three security concerns. First, it only validates the registers on preemption, not on a context switch event. Second, PAL is susceptible to replay attacks, where an attacker manipulates the nonce, signature, and register values to match a previously authenticated version. Third, storing the registers in memory before signing them makes PAL vulnerable to TOCTTOU attacks, where an attacker corrupts the stored register values before the signature is computed. Overall, PAL is vulnerable to attack scenarios ⑤, ⑥, and ❺.

Carlini et al. [9] demonstrated that CFB can bypass CFI schemes that determine the backward edges statically. Since these mitigations, except PATTER, Camouflage, and PAL, do so, their scheme is vulnerable to CFB, re-enabling control-flow hijacking attacks.

While CFI is a powerful approach, there are other approaches [2, 25, 35, 36, 44, 50, 65] to prevent kernel control-flow hijacking attacks. Although we donot discuss these, their existence highlights the ongoing efforts to enhance kernel security.

10 CONCLUSION

In this paper, we introduced HEK-CFI, which provides hardware-enforced protection for control data, effectively mitigating control-flow hijacking attacks. Our HEK-CFI was established as the first kernel CFI-based countermeasure to provide protection for both thread state during system events and return addresses. At the same time, it generically reduces the forward control-flow targets and performance overhead compared to existing kernel mitigations. Overall, HEK-CFI increases kernel security.

11 ACKNOWLEDGEMENTS

We thank the anonymous reviewers and shepard for their valuable feedback. This project has received funding from the Austrian Research Promotion Agency (FFG) via the SEIZE and AWARE project (FFG grant number 888087 and 891092). Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2005. Control-Flow Integrity. In *CCS*.

[2] Muhammad Abubakar, Adil Ahmad, Pedro Fonseca, and Dongyan Xu. 2021. SHARD: Fine-Grained Kernel Specialization with Context-Aware Hardening. In *USENIX Security Symposium*.

[3] Android. 2022. Kernel Control Flow Integrity. https://source.android.com/docs/security/test/kcfi

[4] ARM. 2022. Arm Architecture Reference Manual for A-profile architecture.

[5] Tyler K. Bletsch, Xuxian Jiang, Vincent W. Freeh, and Zhenkai Liang. 2011. Jump-oriented programming: a new class of code-reuse attack. In *AsiaCCS*.

[6] Daniel P Bovet and Marco Cesati. 2005. *Understanding the Linux Kernel*. O'Reilly Media, Inc.

[7] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. 2008. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *ACM Conference on Computer and Communications Security (CCS)*.

[8] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. 2020. KASLR: Break It, Fix It, Repeat. In *AsiaCCS*.

[9] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. 2015. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity.. In *USENIX Security*.

[10] Nicholas Carlini and David A. Wagner. 2014. ROP is Still Dangerous: Breaking Modern Defenses. In *USENIX Security*.

[11] Andrew Cooper. 2021. Xen CET Supervisor Shadow Stacks. https://xenbits.xen.org/people/andrewcoop/Xen-CET-SS.pdf

[12] Jonathan Corbet. 2012. Supervisor mode access prevention. https://lwn.net/Articles/517475/

[13] Jonathan Corbet. 2015. Kernel security: beyond bug fixing. https://lwn.net/Articles/662219/

[14] Jonathan Corbet. 2016. Defending against Rowhammer in the kernel. https://lwn.net/Articles/704920/

[15] Standard Performance Evaluation Corporation. 2017. SPEC CPU 2017. https://www.spec.org/cpu2017/

[16] John Criswell, Nathan Dautenhahn, and Vikram Adve. 2014. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *S&P*.

[17] Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2017. PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables. In *NDSS*.

[18] Rémi Denis-Courmont, Hans Liljestrand, Carlos Chinea, and Jan-Erik Ekberg. 2020. Camouflage: Hardware-assisted CFI for the ARM Linux kernel. In *DAC*.

[19] Jake Edge. 2011. Extending the use of RO and NX. https://lwn.net/Articles/422487/

[20] Jake Edge. 2013. Kernel address space layout randomization. https://lwn.net/Articles/569635/

[21] Reza Mirzazade Farkhani, Saman Jafari, Sajjad Arshad, William Robertson, Engin Kirda, and Hamed Okhravi. 2018. On the Effectiveness of Type-Based Control Flow Integrity. In *ACSAC*.

[22] Alexander J. Gaidis, Joao Moreira, Ke Sun, Alyssa Milburn, Vaggelis Atlidakis, and Vasileios P. Kemerlis. 2023. FineIBT: Fine-grain Control-flow Enforcement with Indirect Branch Tracking. *arXiv:2303.16353* (2023).

[23] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. 2016. Fine-Grained Control-Flow Integrity for Kernel Software. In *Euro S&P*.

[24] GitHub. 2021. CodeQL. https://codeql.github.com/

[25] Spyridoula Gravani, Mohammad Hedayati, John Criswell, and Michael L. Scott. 2019. IskiOS: Lightweight Defense Against Kernel-Level Code-Reuse Attacks. *arXiv:1903.04654* (2019).

[26] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*.

[27] Daniel Gruss, Michael Schwarz, and Moritz Lipp. 2018. Meltdown: Basics, Details, Consequences. In *Black Hat USA*.

[28] Ralf Hund, Thorsten Holz, and Felix C. Freiling. 2009. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In *USENIX Security*.

[29] Intel. 2016. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture.

[30] Intel. 2017. Control-flow Enforcement Technology Preview. Revision 2.0.

[31] Intel. 2019. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 4: Model-Specific Registers.

[32] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. 2014. ret2dir: Rethinking kernel isolation. In *USENIX Security*.

[33] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. 2012. kGuard: Lightweight Kernel Protection against Return-to-User Attacks. In *USENIX Security*.

[34] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *S&P*.

[35] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ziegler, Wolfgang Schröder-Preikschat, Daniel Lohmann, and Rüdiger Kapitza. 2013. Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring. In *NDSS*.

[36] Anil Kurmus and Robby Zippel. 2014. A Tale of Two Kernels: Towards Ending Kernel Hardening Wars with Split Kernel. In *CCS*.

[37] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. 2014. Code-Pointer Integrity. In *OSDI*.

[38] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *IEEE / ACM International Symposium on Code Generation and Optimization – CGO*.

[39] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Yi Chien, Felipe Huici, Nathan Dautenhahn, and Pierre Olivier. 2022. Assessing the Impact of Interface Vulnerabilities in Compartmentalized Software. In *NDSS*.

[40] Guoren Li, Hang Zhang, Jinmeng Zhou, Wenbo Shen, Yulei Sui, and Zhiyun Qian. 2023. A Hybrid Alias Analysis and Its Application to Global Variable Protection in the Linux Kernel. In *USENIX Security*.

[41] 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* (2018).

[42] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. 2019. PAC it up: Towards Pointer Integrity using ARM Pointer Authentication. In *USENIX*.

[43] Lukas Maar, Martin Schwarzl, Fabian Rauscher, Daniel Gruss, and Stefan Mangard. 2023. DOPE: DOmain Protection Enforcement with PKS. In *ACSAC*.

[44] Derrick McKee, Yianni Giannaris, Carolina Ortega Perez, Howard Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. 2022. Preventing Kernel Hacks with HAKC. In *NDSS*.

[45] Larry McVoy and Carl Staelin. 1996. Lmbench: Portable Tools for Performance Analysis. In *USENIX ATC*.

[46] Joao Moreira. 2022. Kernel FineIBT Support. https://lwn.net/Articles/891976/

[47] James Morse. 2015. arm64: kernel: Add support for Privileged Access Never. https://lwn.net/Articles/651614/

[48] PaX Team. 2015. Rap: Rip rop.

[49] Phoronix. 2022. OpenBenchmarking. https://openbenchmarking.org

[50] Marios Pomonis, Theofilos Petsios, Angelos D Keromytis, Michalis Polychronakis, and Vasileios P Kemerlis. 2017. kR^ X: Comprehensive Kernel Protection against Just-In-Time Code Reuse. In *EuroSys*.

[51] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P. Kemerlis, and Michalis Polychronakis. 2020. xMP: Selective Memory Protection for Kernel and User Space. In *S&P*.

[52] Samsung Knox News. 2016. Real-time Kernel Protection (RKP). https://www.samsungknox.com/de/blog/real-time-kernel-protection-rkp

[53] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *S&P*.

[54] Mark Seaborn. 2015. Exploiting the DRAM rowhammer bug to gain kernel privileges. http://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html

[55] INetCop Security. 2016. New Reliable Android Kernel Root Exploitation Techniques. http://powerofcommunity.net/poc2016/x82.pdf

[56] Seth Jenkins. 2022. Exploiting CVE-2022-42703 - Bringing back the stack attack. https://googleprojectzero.blogspot.com/2022/12/exploiting-CVE-2022-42703-bringing-back-the-stack-attack.html

[57] Seth Jenkins. 2023. Analyzing a Modern In-the-wild Android Exploit. https://googleprojectzero.blogspot.com/2023/09/analyzing-modern-in-wild-android-exploit.html

[58] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS*.

[59] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. 2019. Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity. In *HASP*.

[60] Di Shen. 2017. Defeating Samsung KNOX with Zero Privilege. https://infocondb.org/con/black-hat/black-hat-usa-2017/defeating-samsung-knox-with-zero-privilege

[61] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William R. Harris, Taesoo Kim, and Wenke Lee. 2016. Enforcing Kernel Security Invariants with Data Flow Integrity. In *NDSS*.

[62] Yoo Sungbae, Park Jinbum, Kim Seolheui, Kim Yeji, and Kim Taesoo. 2022. In-Kernel Control-Flow Integrity on Commodity OSes using ARM Pointer Authentication. In *USENIX Security*.

[63] Zhi Wang and Xuxian Jiang. 2010. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *S&P*.

[64] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. 2019. KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities. In *USENIX Security*.

[65] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. 2018. FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities. In *USENIX Security*.

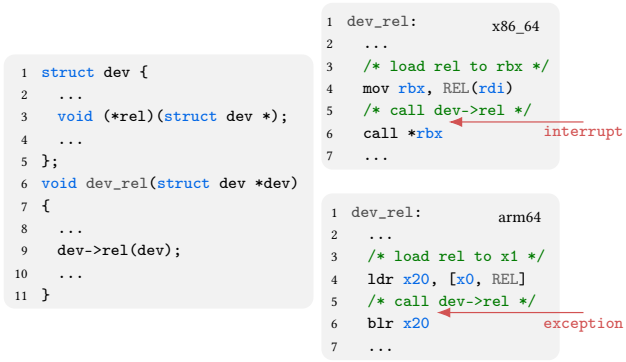

Figure 8: `dev_rel` performs an indirect branch (for x86_64 and arm64). If an interrupt/exception is triggered shortly before the branch, the register (`rbx` or `x20`) is stored to writable memory. By overwriting this memory location, an attacker can gain control over the register, resulting in a CFHP.

```
1  ret_from_fork:
2    mov rax, rdi
3    call schedule_tail
4    test rbx, rbx
5    jne 1f
6  2:
7    mov rsp, rdi
8    call syscall_exit_to_user_mode
9    jmp __irqentry_text_end
10 1:
11   mov r12, rdi
12   call *rbx
13   jmp 2f
```

Listing 4: For x86_64.

```
1  ret_from_fork:
2    bl schedule_tail
3    cbz x19, 1f
4    mov x0, x20
5    blr x19
6  1:
7    get_current_task tsk
8    mov x0, sp
9    bl asm_exit_to_user_mode
10   b ret_to_User
```

Listing 5: For arm64.

Figure 9: Instruction sequence `ret_from_fork` that can be exploited for a CFHP by tampering with the callee-saved register, `rbx` for x86_64 and `x19` for arm64.

[66] Mengyao Xie, Chenggang Wu, Yinqian Zhang, Jiali Xu, Yuanming Lai, Yan Kang, Wei Wang, and Zhe Wang. 2022. CETIS: Retrofitting Intel CET for Generic and Efficient Intra-Process Memory Isolation. In *CCS*.
[67] Jianhao Xu, Luca Di Bartolomeo, Flavio Toffalini†, Bing Mao, and Mathias Payer. 2023. WarpAttack: Bypassing CFI through Compiler-Introduced Double-Fetches. In *S&P*.
[68] 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:1912.10666* (2019).
[69] Yuval Yarom and Katrina Falkner. 2014. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security*.
[70] Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *USENIX Security*.
[71] Xiaochen Zou. 2022. CVE-2022-27666 Writeup. https://etenal.me/archives/1825

APPENDIX

12 MOTIVATIONAL EXPLOIT EXAMPLES

In this section, we demonstrate various attack scenarios where an attacker obtains a Control-Flow Hijacking Primitive (CFHP), allowing them to deviate from the legal Control-Flow Graph (CFG) in an attacker-controlled manner. In the cases of Appendices 12.1 and 12.2, these CFHP scenarios also allow adversaries to bypass the control-flow restrictions imposed by applied kernel CFI-based countermeasures. As a result, attackers can redirect the control flow to an attacker-controlled code location. Furthermore, in Appendix 12.3, we show that even being within the approximated CFG with signature granularity is insufficient to mitigate this attack, leaving the system vulnerable to compromise by adversaries. Our approach, HEK-CFI, addresses these attack scenarios by protecting the thread state during all system events and control-data pointers, i.e., function and operation table pointers.

12.1 Attacking Thread State on Exceptions and Interrupts

Suppose an interrupt or exception request disrupts a thread. In that case, the system stores the thread state, including general-purpose registers that may hold control data, to a writable memory location, e.g., thread stack frame. This allows the system to resume the thread's execution once the interrupt or exception handling is completed. However, there is a security issue. If an attacker tampers with the memory holding control data in the saved thread state, they gain control over the control data when the system restores the thread state. As a result, the attacker obtains a CFHP.

Figure 8 illustrates this attack scenario by pivoting the `dev_rel` function (shown in C, x86_64 assembly, and arm64 assembly). This function makes an indirect branch to the address stored in `dev->rel`. To exploit `dev_rel` as a CFHP, an attacker can initiate an asynchronous interrupt or exception, such as using the high-precision `hrtimer` interface in the Linux kernel. This interrupt or exception may happen right before the assembly code's indirect branch in Line 4.If the interrupt or exception occurs precisely at this point, the attacker can manipulate the register `rbx` or `x20` stored on the stack frame. When the interrupt or exception handling finishes, the execution continues with the manipulated register. Hence, the indirect branch redirects the control flow to the value stored in the tampered register, effectively hijacking the control flow.

In Figure 8, we exemplify this attack on the thread state during the system events exception and interrupts with the function `dev_rel`. This can be generalized for every function, performing an indirect branch which can be disrupted by an exception, e.g., page fault on `copy_from/to_user`, or interrupt, e.g., timer interrupt.

12.2 Attacking Thread State on Context Switch

Exploiting instruction sequence of `ret_from_fork`. In this attack scenario, an attacker tampers with callee-saved registers (part of the thread state) of a thread currently not running to obtain a CFHP. In Figure 9, we illustrate the assembly instruction sequence, which is executed as first instruction sequence for a just created process using `fork` to return to the user space. For x86_64, the kernel executes the `ret_from_fork` function as shown in Listing 4. If an attacker tampers with the callee-saved registers between the `fork` and execution of `ret_from_fork`, they gain control over `rbx` and `r12` (which will be `rdi`). Since `rbx` is not zero (Line 4), the indirect call at Line 12 is made with the attacker-controlled registers. This control over the indirect call is what allows the attacker to achieve a CFHP. The function `ret_from_fork` for arm64 (see Listing 5) closely resembles the one for x86_64. By corrupting the memory

```
1  void rq_qos_wait(...)
2  {
3    struct rq_qos_wait_data data = {
4      ...
5      .cb = acquire_inflight_cb,
6    };
7    ...
8    do {
9      /* break out of while loop */
10     if (data.got_token)
11       break;
12     /* perform indirect branch */
13     data.cb(...);
14     ...
15     /* perform context switch */
16     schedule();
17   } while (1);
18 }
```

Listing 6: Code pattern example that can be exploited for a CFHP. An attacker tampers with the callee-saved register, containing the function pointer `data.cb` (Line 5), during the `schedule` function (Line 16) to have control over the indirect branch (Line 13).

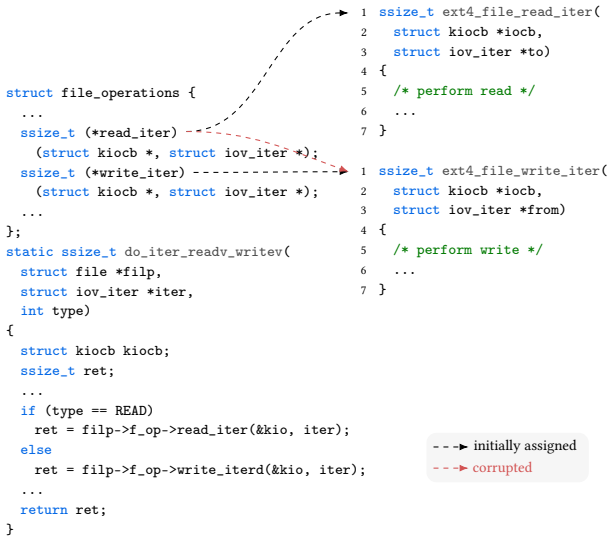

Figure 10: Exploitation to break signature-granular CFI. By overwriting the function pointer `read_iter` with `ext4_file_write_iter` an attacker enforce to call the write function on a read request.

location, where `x19` is stored, (i.e., `taxk_struct`) an attacker gains control over it, which is then used for a CFHP in Line 5.

For a more detailed exploitation, we refer to Appendix 13, where we exploit the CVE-2019-2215 to perform an end-to-end attack.

Exploiting generic code patterns. Another way to achieve a CFHP is through code patterns that involve storing a function pointer in a callee-saved register and then calling functions leading to a context switch, such as `schedule` or `mutex_lock`. On the context switch, the thread state including the callee-saved register is stored in memory, and the thread is marked as not running. Similar to the previous example, an attacker interferes with the memory location where the callee-saved register is stored while the thread is not running. This manipulation allows the attacker to gain control over the function pointer, resulting in a CFHP when the thread resumes execution to call the function pointer.

We illustrate in Listing 6 an example of such a code pattern. The function `rq_qos_wait` loads a function pointer into a register in Line 5 and uses it in an indirect branch in Line 13. In between, it may perform a context switch in Line 16. Crucially, whether the function uses a callee-saved register or not depends on the compiler. For our observations, we compiled the Linux v5.18 with gcc version 10.2.1 for both x86_64 and arm64 architectures using the default configuration of Ubuntu 22.04.1 LTS. In both cases, the kernel used callee-saved registers to store the function pointer, resulting that `rq_qos_wait` can be exploited to obtain a CFHP.

12.3 Attacking Signature-based CFI

CFI with signature granularity provides weak security guarantees for the Linux kernel because the set of control-flow targets matching the signature is too large. In Figure 10, we demonstrate an example of how an attacker can bypass signature-granular CFI. To explain, the attacker begins by opening a file where they have read- but not write-permission, e.g., `/etc/passwd`. This action causes the kernel to allocate a `file` object containing an operation table pointer to a `file_operations` object. Within the `ext4` filesystem, its members `read_iter` and `write_iter` point to functions `ext4_file_read_iter` and `ext4_file_write_iter`. By performing a read or write operation to or from the opened file, the kernel calls `do_iter_readv_writev`.Since both signatures match, the attacker can overwrite `read_iter` member with the address of `ext4_file_write_iter` and still resides in the over-approximated CFG determined based on signatures. However, this manipulation causes a read operation to perform a write operation. Consequently, the attacker calls the `read` syscall to write content to the file where they do not have write-permissions, e.g., `/etc/passwd`. This results in a persistent privilege escalation, which CFI with signature granularity cannot prevent.

In fact, since the operation table `file_operations` in Figure 10 is mapped as read-only, an attacker cannot directly manipulate the function pointer `read_iter`. However, they can forge an operation table and tamper with the operation table pointer of the `file` object, pointing to the forged operation table. In this way, they can bypass the read-only mapping and perform this privilege escalation attack.

13 MOTIVATIONAL END-TO-END EXPLOIT

This section provides a motivational end-to-end attack that exploits the CVE-2019-2215 vulnerability. The vulnerability allows an arbitrary read-and-write primitive that matches our threat model in Section 3.1. This end-to-end attack shows the severity of not protecting the thread state, as this attack can bypass almost all kernel CFI schemes (mitigations that do not protect the thread state, see Table 1) including the one used by Android [3], i.e., kCFI. Therefore,

```
1  void find_process(void)
2  {
3    size_t kernel_base = leak_kaslr();
4    printf("[*] looking for the process \"t1\"...\n");
5    size_t task = INIT_TASK_OFFSET + kernel_base;
6    size_t task_stack;
7    while (1) {
8      arb_read(task + TASKS_OFFSET, (size_t)&task);
9      task = task - TASKS_OFFSET;
10
11     char name[8] = {0};
12     arb_read(task + COMM_OFFSET, (size_t)name);
13     if (!strcmp((char *)name, "t1")) {
14       arb_read(task + STACK_OFFSET, (size_t)&task_stack);
15       printf("[+] we found the process at %lx stack %lx\n",
             task, task_stack);
16       break;
17     }
18   }
19 }
```

Listing 7: Obtaining `task_struct`'s kernel address and the kernel stack address of the process named `"t1"`.

this attack can be used through Android kernel exploitation to perform control-flow hijacking attacks, which often relied on using the arbitrary read-and-write primitive to tamper with global control data pointers [40], e.g., `ptmx_fops` [55, 60] or `modprobe_path` [71]. The high-level exploitation strategy is a three-step plan as follows.

As the *first* step, we create a new process by calling `fork`, which prompts the kernel to allocate a `task_struct` and an associated kernel stack. The `task_struct` is allocated via the dedicated allocation cache `task_struct_cachep`, while the kernel stack is allocated via `vmalloc`, both of which reuse freed objects for future allocations. We change the name of the just created process to a unique one, i.e., `"t1"`, and use our read primitive to obtain the address of its `task_struct`. The way to do this is by leaking `init_task` (Line 5 of Listing 7), which is the `task_struct` of the initial, i.e., first, high-privilege process. Since the Linux kernel stores all processes as a linked list (via the `struct list_head tasks` member), we iterate through all processes at Line 8 and use the arbitrary read to obtain the process's names at Line 12. We check whether its name (via the `char comm[TASK_COMM_LEN]` member) is the same as the unique one at Line 13, we previously set to `"t1"`. If the stored name is the same, we leak the stored stack (via the `void *stack` member variable) at Line 14[2].

As the *second* step, we kill the process named `"t1"` prompting the kernel to free its `task_struct` and kernel stack. By creating a new process via `fork` called `"t2"` shortly after we reclaim both the previously leaked `task_struct` and kernel stack[3]. Immediatly after this `fork`, we use our arbitrary write primitive to overwrite the stored callee-saved registers (as part of the thread state) stored either on the kernel stack for x86_64 or the `task_struct` for arm64 (denoted as `THREAD_CPU_CXT`). For x86_64 these are `rbp`, `rbx`, and `r12-15`, while for arm64 `x19-29` and `x9`. If process

[2]Leaking the kernel stack's address is only necessary for x86_64 as arm64 stores the callee-saved registers to its `task_struct`

[3]To bypass the free-list randomisation of the kernel heap allocator and reclaim the leaked `task_struct`, we massage the dedicated allocator `task_struct_cachep`. For the sake of simplicity, we assume that we successfully reclaim it.

```
1  __switch_to_asm:
2  // Save callee-saved regs
3    pushq rbp
4    pushq rbx
5    pushq r12
6    pushq r13
7    pushq r14
8    pushq r15
9
10 // switch stack
11   movq rsp, TASK_sp(rdi)
12   movq TASK_sp(rsi), rsp
13
14 // restore callee-saved regs
15   popq r15
16   popq r14
17   popq r13
18   popq r12
19   popq rbx
20   popq rbp
21
22   jmp  __switch_to
```

Listing 8: For x86_64.

```
1  cpu_switch_to:
2    mov  x10, #THREAD_CPU_CXT
3    add  x8, x0, x10
4    mov  x9, sp
5
6  // store callee-saved regs
7    stp  x19, x20, [x8], #16
8    stp  x21, x22, [x8], #16
9    stp  x23, x24, [x8], #16
10   stp  x25, x26, [x8], #16
11   stp  x27, x28, [x8], #16
12   stp  x29, x9, [x8], #16
13   str  lr, [x8]
14   add  x8, x1, x10
15
16 // restore callee-saved regs
17   ldp  x19, x20, [x8], #16
18   ldp  x21, x22, [x8], #16
19   ldp  x23, x24, [x8], #16
20   ldp  x25, x26, [x8], #16
21   ldp  x27, x28, [x8], #16
22   ldp  x29, x9, [x8], #16
23   ldr  lr, [x8]
24
25 // restore stack pointer
26   mov  sp, x9
27   msr  sp_el0, x1
28   ret
```

Listing 9: For arm64.

Figure 11: Context switch assembly code of the Linux kernel.

`"t2"` is scheduled, it loads these manipulated register values from the stack or `task_struct` as shown with `__switch_to_asm` in Listing 8 and `cpu_switch_to` in Listing 9, respectively. Since it is the first time scheduled, the Linux kernel prompts this process to execute `ret_from_fork` shown in Figure 9. With control over the register, we just created a CFHP with control over its arguments (Lines 11 and 12 for x86_64 and Lines 4 and 5 for arm64), allowing an adversary to perform a powerful control-flow hijacking attack.

As the *third* step, we perform a classical control-flow hijacking attack such as executing an instruction sequence equivalent to `commit_creds(prepare_kernel_cred(0))` to escalate privileges. This powerful control-flow hijacking attack depends on a race window, to tamper with the process's (`"t2"`) kernel stacks or `task_struct` before it gets scheduled. However, this race window is large and does not affect the practicality of this attack.

14 INSTRUMENTATION

This section demonstrates the instrumentation process of HEK-CFI, where we illustrate how HEK-CFI ensures control-data integrity for globally and locally accessible control data.

Globally accessible control data. Listing 10 depicts an allocation and deallocation of `dev`, and a read from and write to its member variable `void (*rel)(struct dev *)`. We assume that HEK-CFI provides protection for the function pointer `rel`. Although this example protects a function pointer, the same applies to protected operation table pointers. For the instrumentation, we refer to the function `ss_alloc`, `ss_free`, `ss_rd`, and `ss_wr`, to globally accessible safe storage allocation, deallocation, read, and write.

```
struct dev {
  /* protected fn pointer */
  void (*rel)(struct dev *);
  ...
};
void dev_rel(struct dev *);
void function(void) {
  struct dev *dev;
  dev = kmalloc();
  ...
  dev->rel = &dev_rel;
  ...
  if (dev->rel)
    dev->rel(dev);
  ...
  kfree(dev);
}
```

Listing 10: Original code of accessing a struct dynamically allocated.

```
void function(void) {
  struct dev *dev;
  dev = kmalloc();
+ ss_alloc(dev->rel);
  ...
- dev->rel = &dev_rel;
+ ss_wr(dev->rel, &dev_rel);
  ...
- if (dev->rel)
-   dev->rel(dev);
+ reg = ss_rd(dev->rel);
+ if (reg)
+   reg(dev);
  ...
+ ss_free(dev->rel);
  kfree(dev);
}
```

Listing 11: Instrumented code of accessing a struct dynamically allocated.

```
struct delayed_call {
  /* protected fn pointer */
  void (*fn)(void *);
  void *arg;
};
void dc_fn(void *);
void function2(void) {
  struct delayed_call dc;
  ...
  dc.fn = &dc_fn;
  ...
  dc.fn(dc.arg);
  ...
}
```

Listing 12: Original code of a local struct.

```
void function2(void) {
  struct delayed_call dc;
+ ptls_alloc(dc.fn);
  ...
- dc.fn = &dc_fn;
+ ptls_wr(dc.fn, &dc_fn);
  ...
- dc.fn(dc.arg);
+ reg = ptls_rd(dc.fn);
+ reg(dc.arg);
  ...
+ ptls_free(dc.fn);
}
```

Listing 13: Instrumented code of a local struct.

Listing 11 demonstrates how HEK-CFI protects the function pointer `rel` when the struct it belongs to (`dev`) is dynamically allocated. HEK-CFI inserts `ss_alloc` to allocate a safe storage for `rel` shortly after dynamically allocating `dev`. HEK-CFI replaces the write access to `rel` with `ss_wr`, which writes to the protected data within the allocated safe storage. On read access, HEK-CFI replaces the read with `ss_rd`, which reads the protected data from the safe storage and stores it in the register `reg`. HEK-CFI inserts `ss_free` shortly before freeing the `dev` to free the safe storage.

Locally accessible control data. Listing 12 demonstrates the local variable `delayed_call` containing a function pointer `void (*fn) (void *)` protected by HEK-CFI. For the instrumentation, we refer to the functions `ptls_alloc`, `ptls_free`, `ptls_rd`, and `ptls_wr`, to locally accessible safe storage allocation, deallocation, read, and write, respectively. As illustrated in Listing 13, HEK-CFI inserts `ptls_alloc` and `ptls_free` on function entry and exit. Additionally, HEK-CFI replaces the write and read access to or from the protected function pointer `fn` with `ptls_wr` and `ptls_rd`.

Table 4: Micro benchmark results.

	Benchmarks	Baseline	Overhead in %
Latency in µs	fcntl lock	1.51	18.9 ± 0.9
	pagefault	0.17	18.3 ± 1.4
	proc call	0.0027	−1.9 ± 3.3
	proc fork	81.5	8.7 ± 1.9
	proc fork+exec	90.5	8.9 ± 1.5
	proc shell	359	7.6 ± 0.8
	signal install	0.15	15.4 ± 1.9
	signal catch	1.12	8.8 ± 0.3
	signal fault	0.48	8.1 ± 1.0
	syscall null	0.079	25.8 ± 0.3
	syscall open+close	1.19	18.7 ± 2.3
	syscall read	0.12	20.6 ± 1.6
	syscall write	0.095	24.8 ± 3.9
	syscall stat	0.44	9.4 ± 1.1
	syscall fstat	0.15	18.2 ± 0.9
Bandwidth in MB/s	pipe 1 k	0.13	24.3 ± 4.2
	pipe 128 k	3.74	20.8 ± 0.2
	unix 1 k	337	11.2 ± 0.3
	unix 128 k	82	11.3 ± 0.8
	file rd 4 k o2c	2.69	15.4 ± 1.1
	file rd 1 M o2c	15	6.2 ± 0.7
	file rd 4 k ip	7.71	13.6 ± 0.8
	file rd 1 M ip	16.2	7.5 ± 0.4
	mmap rd o2c 4 k	1.34	10.3 ± 0.4
	mmap rd o2c 1 M	14.8	6.3 ± 0.5
	mmap rd 4 k	43.7	0.1 ± 0.2
	mmap rd 1 M	41.6	1.4 ± 2.0
	geo mean	-	12.3 ± 1.5

15 DETERMINE IDEAL USER POLICY

We developed a tool that takes the average permitted control-flow targets value, i.e., $AIA_{hek-cfi}$ illustrated in Equation (2), as input and determines the user policy, i.e., maximum permitted forward control-flow targets, as output. It does so by first finding the value of p, the number of permitted function pointers of Equation (2), to match the given input. This value p resides within the range of 1 to n. After determining p, the tool identifies the set $|T_i|$ with the largest number of permitted control-flow targets. This largest number is the output value of the user policy.

16 DETAILED EVALUATION RESULTS

In this section, we illustrate the results of our micro benchmark performance evaluation in Table 4, performed with LMbench [45]. Moreover, we evaluate the binary and compile-time overhead of the Linux kernel enhanced with HEK-CFI. Lastly, we evaluate the time taken by our code analyzer and parser.

Binary size and compile time overhead. Since HEK-CFI inserts validation checks to ensure its functionality, the inserted checks increase both the binary size and the compile time. To illustrate their increase, we compile an unmodified Linux kernel v5.18 with clang version 15.0.0 as a baseline. We then compile our modified Linux kernel with our LLVM pass and the user policy described in our case study. The results are that the binary size and compile-time increase by 0.97 % and 47.2 ± 1.6%, respectively.

Code analyzer and parser. We measure the time taken to generate a database and query requests for CodeQL. Moreover, we measure the required time for parsing the results from CodeQL to output the file for our LLVM pass. We perform each measurement eight times and calculate the geometric mean and standard deviation, where the results are 2h53m39s±6s, 59m4s±6s, and 17.8s±0.4s for database generation, database queries, and parser, respectively.