eBPF: Elevating Linux Kernel Functionality

Teh Beng Yen
Shih-Hung Tang
{r12922178,r12922042}@csie.ntu.edu.tw
Department of Computer Science and Information Engineering
National Taiwan University

Abstract

Control-flow hijacking attacks pose a significant threat to software security by allowing attackers to redirect a program's execution to malicious code. Enforcing Control Flow Integrity (CFI) in the kernel (kCFI) is a promising defense, but existing approaches often suffer from high overhead and inflexibility due to statically defined policies. This project explores the use of an extended Berkeley Packet Filter (eBPF) to enhance kCFI, leveraging eBPF's dynamic and efficient monitoring capabilities. We conducted a case study on eKCFI, a KCFI approach via eBPF proposed by Jia et al., and extended it with additional features, introducing eKCFI-RET for more fine-grained CFI protection of indirect call and return sites. Our solution is compatible with recent Linux features, such as Retpoline, and introduces minimal overhead when inactive, enhancing security without compromising performance.

1 Introduction

Extended Berkeley Packet Filter (eBPF) is a powerful technology integrated into the Linux kernel that enables the execution of sandboxed programs within a privileged context. This allows for secure and efficient enhancement of kernel functionality without modifying the kernel source code or loading kernel modules. eBPF provides robust capabilities for dynamically monitoring and altering system behavior, making it an ideal tool for improving Kernel Control Flow Integrity (KCFI). Our project is motivated by the flexibility and security of eBPF to enhance KCFI, a critical aspect of maintaining kernel stability and security. Inspired by insights from the eBPF and Networking session at the Linux Plumbers Conference (LPC)[5] in 2023, we aim to replicate and refine the eKCFI[11] system proposed by Jia et al. The implementation of the eKCFI is released on Github[1].

KCFI aims to ensure that the program's execution follows a legitimate control flow graph (CFG), thereby preventing malicious redirections. It is a crucial security mechanism designed to protect against control flow hijacking attacks, which allow adversaries to redirect the execution flow of a program, potentially leading to unauthorized access or control over a system. For instance, in homework 2, a rootkit is used to alter the behavior of syscalls by overwriting syscall table entries, such as hooking the system call to tamper the structures obtained via `getdents64` to hide specific files while executing the `ls` command.

However, current KCFI approaches suffer from high overhead and inflexibility. The LLVM-based approach[16] provides fine-grained forward-edge protection (indirect call/jump) by relying on static policies derived from function prototypes. However, updating these policies requires a kernel rebuild and reboot, making it inefficient. Existing hardware mechanisms, such as BTI[1], offer poor granularity as they only restrict branching to the entry of a function. In other words, it cannot prevent attacks that jump to the injected malicious functions. To address these challenges, eKCFI introduces a novel attachment mechanism for KCFI that leverages eBPF tracepoints to invoke handlers and instrument the trampoline before each indirect call site accurately. Their proposed KCFI approach will be discussed in more detail in Section 3.2.

In this project, we explore how eKCFI works and extend it to safeguard against exploitation of the ret instruction, enhancing overall security against control flow hijacking. Our approach offers the following contributions:

- Protection: Extending the existing framework to safeguard against exploitation of the ret instruction.
- Coverage: Expanding the approach to include trampoline hooking for all indirect call/jmp and ret instructions.
- Refined CFG: Generating a more detailed and fine-grained CFG to encompass all invoked function calls
- Different Handlers: Distinguishing between various instructions by assigning different eBPF policy handlers for each type of control flow transfer

2 Background

2.1 eBPF tracing

eBPF tracing is a robust framework in the Linux kernel that facilitates real-time monitoring and modification of kernel and application behavior. By leveraging eBPF tracing, developers and system administrators can achieve granular insights and control over system performance and security without altering kernel source code or inserting kernel modules. eBPF programs are event-driven, executing when the kernel or an application triggers specific hook points, such as system calls, function entry/exit, kernel tracepoints, and other critical events. Figure 1 illustrates the workflow and architecture of eBPF tracing.

In user space, the process begins with the creation of an eBPF program. This program is compiled into eBPF bytecode and subsequently loaded into the kernel. The kernel's verifier ensures that the bytecode is safe to execute, preventing potential security risks and ensuring system stability. After passing verification, the Just-In-Time (JIT) compiler translates the eBPF bytecode into machine-specific instructions for efficient execution. Once loaded, the eBPF program can attach to various event sources within the kernel, including:

- Kprobes: kernel dynamic instrumentation.
- Uprobes: user-level dynamic instrumentation.
- Tracepoints: kernel static instrumentation.
- Perf events: timed samplings.

[1] eKCFI: https://github.com/hardos-ebpf-fuzzing/ekcfi

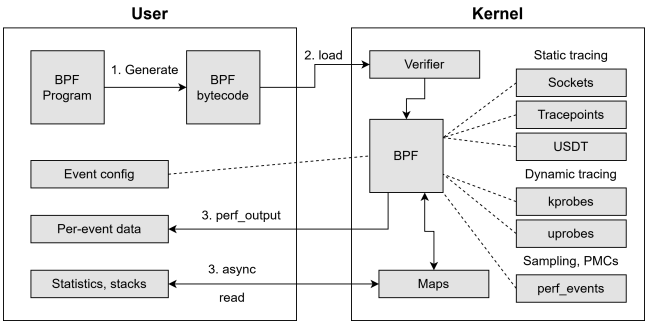

Figure 1: Overview of eBPF tracing

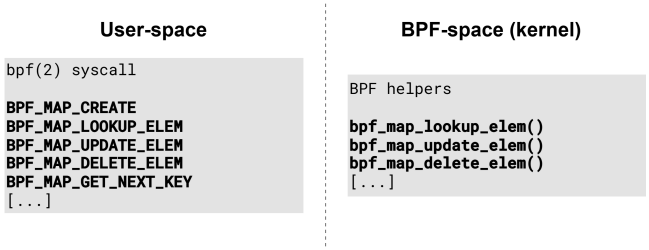

Figure 2: eBPF map operations[10]

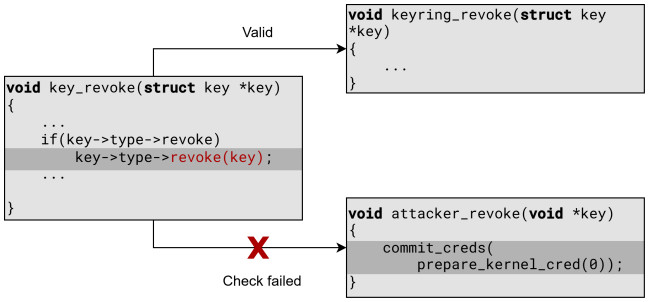

Figure 3: CVE-2016-0728

These connections enable the eBPF program to collect and emit data in real time, which can be processed and analyzed for various purposes such as debugging, performance tuning, and security monitoring. The collected data can be output as per-event data via a perf buffer or stored in eBPF maps for statistical summaries and further analysis.

eBPF tracepoints: The tracepoints are predefined hooks in the Linux kernel that allow eBPF programs to execute custom code at specific kernel events, such as system calls, packet processing, and file operations. This capability is crucial for real-time monitoring and analysis, providing valuable insights without the need for kernel modifications.

eBPF helper functions: The helper functions allow eBPF programs to interact with the kernel environment in a controlled manner, similar to calling kernel functions directly. Helper functions encompass a range of operations, such as printing debugging messages, retrieving system boot time, manipulating network packets, and managing eBPF maps. eBPF programs interact with the kernel environment through a set of helper functions, which enable safe and controlled access to kernel resources. These functions provide various capabilities, such as debugging, time retrieval, network packet manipulation, and managing eBPF maps. For example, the `bpf_printk` helper function allows eBPF programs to print messages to the kernel debug trace file (*/sys/kernel/debug/tracing/trace*), aiding in debugging and development. The helper functions such as `bpf_map_update_elem` and `bpf_map_lookup_elem` enable manipulation of eBPF maps.

eBPF map: eBPF maps serve as versatile key-value stores accessible to eBPF programs. These maps support a wide range of data structures and can store information persistently or transiently. Accessible both from user space via system calls and directly from the helper functions within the kernel, eBPF maps enable programs to maintain stateful information and coordinate effectively with other parts of the kernel. Figure 2 illustrates the interaction between user space and the kernel for eBPF map operations. System calls and eBPF helper functions allow eBPF programs to create, update, and query maps seamlessly, leveraging the efficiency and flexibility of eBPF in various application scenarios.

2.2 Control Flow Integrity

Control Flow Integrity (CFI)[7] is introduced as a security mechanism to mitigate control-flow hijacking attacks. The fundamental concept of CFI is to restrict a program's control flow to its legitimate paths as defined by its control flow graph (CFG). The CFG is a detailed representation of all potential execution paths through a program. CFI enforces that every indirect control flow transfer, such as indirect calls and returns, adheres to the valid paths within the precomputed CFG.

Take CVE-2016-0728[8] as an example, where privilege escalation is possible by overwriting a function pointer following a use-after-free vulnerability. In a kernel scenario, consider a `key_revoke` function, which is supposed to call a `key->type->revoke` function stored in a function pointer. With a use-after-free vulnerability, an attacker might corrupt the function pointer and redirect the control flow to their malicious `revoke` function. This malicious function could then invoke kernel credential functions to perform privilege escalation. With CFI deployed, when the attacker attempts to redirect the control flow to their malicious `revoke` function, the CFI checks will detect that this control flow transfer is invalid. Consequently, the system will prevent the attack by terminating the process, thereby maintaining the integrity of the kernel's control flow.

3 Case Study: eKCFI

In our case study, Jia et al. introduced eKCFI[11] that enforces Kernel Control Flow Integrity (KCFI) for Linux throughout its runtime. eKCFI utilizes the flexibility of eBPF to dynamically enable and disable KCFI policies without necessitating kernel recompilation. It protects against attacks that hijack indirect call sites, such as function pointers in the syscall table, and introduces only modest performance overhead when enabled.

With our knowledge, we introduced eKCFI-RET to provide more fine-grained protection for return sites while maintaining the flexibility to activate KCFI policies during runtime. Our approach preserved compatibility with the existing features of newer Linux versions, such as Retpoline[3], which are commonly enabled. eKCFI-RET introduced negligible overhead when disabled.

3.1 Threat Model And Assumptions

In this project, we assume that the kernel is generally benign but may contain vulnerabilities exploitable by attackers through malicious system calls or crafted network packets. Attackers, with sufficient expertise, could hijack the kernel's control flow by manipulating function pointers or employing code-reuse attacks such as Return-Oriented Programming (ROP) and Jump-Oriented Programming (JOP) attacks.

Our approach relies on the eBPF-based Kernel Control Flow Integrity (KCFI) infrastructure, which we assume to be secure. We assume the infrastructure is safely installed and managed by a benign administrator, and there is no access for attackers. By leveraging this trusted infrastructure, our approach aims to dynamically trace and secure control flow transfers, enhancing the overall resilience of the kernel against sophisticated attacks.

3.2 KCFI Design

To enforce control flow integrity[7], one intuitive approach is to check against a static Control Flow Graph (CFG) of a target process. In simple terms, each indirect call has a predefined subset of legitimate call sites. If an indirect call targets a site outside this subset, it violates the policy, and appropriate actions, such as triggering a panic or logging the event, should be taken. As shown in Figure 4, KCFI first (1) retrieves the current control flow transfer information of the target indirect call site, then (2) checks this information against the predefined CFG from the CFG storage. Finally, it (3) applies the corresponding policy based on the result of this verification.

To obtain the current control flow transfer information, we need a method to monitor and intercept target indirect call sites before handling the call. An intuitive way to achieve this is by creating a hooking point before the indirect call sites. There are several ways to implement this: (i) Kprobe-based approach, (ii) Fprobe-based approach, and (iii) Tracepoint-based approach.

Kprobe-based approach: Kprobe[14] allows us to dynamically intercept any kernel routine and trigger custom events. The key idea is similar to a debugger, where the first bytes of the probed instruction, such as an indirect call, are replaced with an interrupt (e.g., `int3` on x86). This causes a trap, invoking the registered pre-handler event. After the pre-handler processes the event, the probed instruction is simulated as a single step. Finally, if a post-handler is defined, it will be invoked after the instruction execution. Although the kprobe-based approach can successfully attach to most indirect calls to retrieve current control flow transfer information, the interrupt causes significant context switch overhead, which impacts performance when handling the event.

Fprobe-based approach: Another method is using the fprobe[4] mechanism. Fprobe mitigates significant overhead by using a synchronous call instead of an interrupt. It instruments handlers at

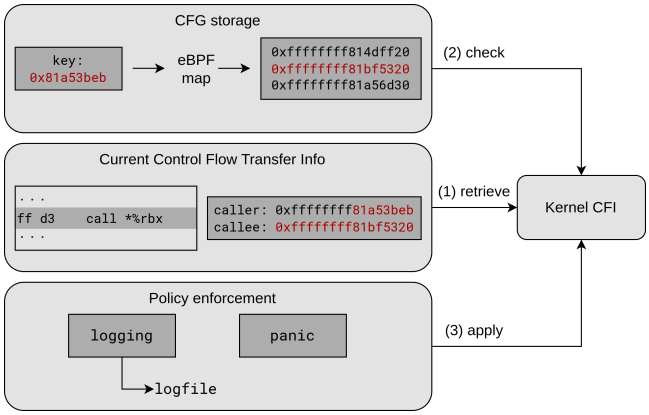

Figure 4: eKCFI Design

multiple function entries and exits. However, the fprobe-based approach has less coverage compared to LLVM-KCFI[16], as it does not instrument certain functions, such as those with `noinstr` and `notrace` attributes, as well as tracing subsystem and library functions. Additionally, the fprobe-based approach introduces performance overhead whenever a probed function is invoked because fprobe cannot distinguish between direct and indirect calls. In other words, the handler is invoked on each function entry, regardless of the call type.

Tracepoint-based approach: To preserve the same coverage of protection provided by LLVM-KCFI while maintaining modest performance overhead, Jia et al. observed that inserting a tracepoint-style attachment approach, which can also invoke handlers synchronously without requiring an interrupt. They introduced a new method similar to the fprobe-based approach by instrumenting each handler, also known as a '*trampoline*', before each indirect call site. To utilize the flexibility of dynamically attaching policy programs, they instrumented `nop` instructions as placeholders that can be dynamically replaced with trampolines as needed.

eKCFI allows dynamic application of CFI policies to target indirect call sites by replacing the instrumented `nop` instruction with its own trampoline. The trampoline is responsible for retrieving the current control flow information and passing it to the attached policy program for verification. To protect return sites, eKCFI-RET uses the same mechanism by instrumenting a `nop` instruction before each return site, which can also be attached to a trampoline. This will be detailed further in the next section.

3.3 KCFI Implementation

3.3.1 Kernel Instrumentation. As shown in Figure 5, we use LLVM to statically instrument a 5-byte `nop` instruction, which will later be replaced by a trampoline. eKCFI employs synchronous calls to eliminate the overhead of kprobe interrupts. Since synchronous calls require a 5-byte instruction, we use a `nop` of the same length for replacement. Indirect calls can target any general-purpose register determined by the compiler. While decoding the current text address to obtain the target register information is one approach, it introduces additional instruction overhead. To minimize the cost between trapping to the trampoline and invoking the policy program,

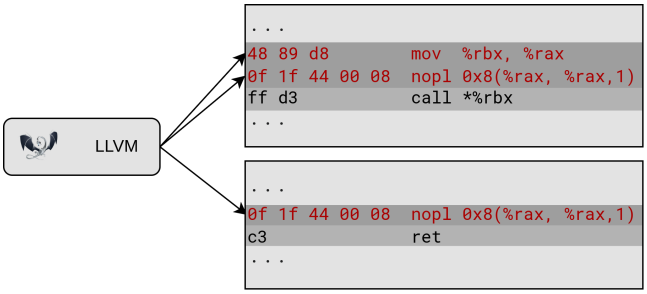

Figure 5: LLVM-based Kernel Instrumentation for Indirect Call Sites and Return Sites

eKCFI adds a `mov` instruction to move the target register to the `rax` register if it is not already `rax`. In eKCFI-RET, the return address of the caller function is placed on the stack. Therefore, eKCFI-RET can simply obtain the current transfer information from the stack, eliminating the need for an additional `mov` instruction.

3.3.2 eKCFI module. To dynamically replace the `nop` with a trampoline on the fly, a privileged program, such as a Linux kernel module, is required. We assume the kernel module is benign and installed safely by a trusted root administrator. The root administrator can then dynamically invoke the module to overwrite the `nop` instruction, as presented in Algorithm 1. The eKCFI module first invokes `text_gen_insn` to generate a synchronous call, then uses `text_poke_queue` and `text_poke_finish`, which are kernel text patching mechanisms, to replace the target `nop` with a direct call to the trampoline.

Algorithm 1 eKCFI Module: Text Patching

```
1:  trampoline ←addr of EKCFI_TRAMPOLINE  // Algorithm 2
2:  function EKCFI_ENABLE_ENTRY(addr)
3:      Lock text_mutex
4:      // Generate call instruction
5:      inst ← TEXT_GEN_INSN(CALL_OP, addr, trampoline)
6:      // Replace 5-bytes nop as trampoline
7:      TEXT_POKE_QUEUE(addr, inst, 5, NULL)
8:      TEXT_POKE_FINISH()
9:      Unlock text_mutex
10: end function
```

3.3.3 eKCFI trampoline. Algorithm 2 presents the pseudocode for the trampoline algorithm for handling trap before each indirect call site. When a trap occurs, the trampoline first stores the current register information and then retrieves the caller and callee information. Since the trampoline is a direct call, the caller can be retrieved from the stack, and, as mentioned earlier, the callee information can be obtained from the `rax` register. Before triggering the attached eBPF policy program, interrupts should be disabled and a lock should be acquired to prevent race conditions. This ensures that only one CPU enters the trampoline at a time, avoiding recursion and preventing multiple processes from repeatedly entering the verification process. Finally, after the eBPF policy program completes its check, the trampoline restores the register information and continues execution from the trap point.

Algorithm 2 eKCFI Trampoline

```
1:  in_check ← false
2:  function EKCFI_TRAMPOLINE(void)
3:      Store register
4:      Load caller and callee info
5:      PREEMPT_DISABLE()
6:      if THIS_CPU_READ(in_check) then
7:          go to 15
8:      end if
9:      THIS_CPU_WRITE(in_check, true)
10:     result ← Call attached eBPF program // Algorithm 3
11:     if result == EKCFI_RET_DENY then
12:         // Do something
13:     end if
14:     THIS_CPU_WRITE(in_check, false)
15:     PREEMPT_ENABLE()
16:     Restore register
17: end function
```

eKCFI-RET: In eKCFI-RET, the algorithm is similar to the trampoline algorithm for indirect call sites. The difference is that both the return address to the trap point (*callee*) and the return address of the `ret` instruction (*caller*) can be retrieved from the current stack. However, the return site coverage can be more extensive in kernel text compared to indirect call sites. If we try to make a "*full*" instrumentation coverage of each return site, a problem arises. As shown in Algorithm 2, before triggering the attached eBPF program, the trampoline disables interrupts and acquires a lock. However, each of these operations is a kernel function and thus has a return site in kernel text. This causes recursion before we can prevent it. A straightforward solution is to blacklist certain kernel functions from being instrumented. Similar to kprobe, we use `within_kprobe_blacklist` to prevent trampoline instrumentation of the functions specified in the kprobe blacklist.

3.3.4 eBPF program.

- Policy program: As shown in Algorithm 3, the attached eBPF program for KCFI retrieves the caller and callee information passed from the trampoline. It then checks this information against the pre-stored CFG from the BPF map using `bpf_map_lookup_elem`. If the current control flow transfer is a valid path, execution continues. Otherwise, appropriate actions, such as logging or panicking the kernel, will be taken.
- Trace Program: As previously mentioned, we assume that eKCFI has a predefined CFG for a kernel process. Several binary analysis tools can be used to construct this CFG. In eKCFI, since the root administrator can define their own eBPF program, Jia et al. introduced a trace eBPF program similar to the policy program. Specifically, the trace program logs the current caller and callee information using `bpf_printk`. The root administrator can then retrieve and parse these logs to update the BPF map program accordingly.

Algorithm 3 eBPF program

```
1:  call_map ← predefined CFG
2:  NR_CALLEES ← size of call_map
3:  function EBPF_EKCFI_CHECK(ctx)
4:      caller_key ← ctx.caller & 0xFFFFFFFF
5:      callees ← BPF_MAP_LOOKUP_ELEM(call_map, caller_key)
6:      for i = 0 to NR_CALLEES do
7:          if ctx.callee == callees[i] then
8:              return EKCFI_RET_ALLOW
9:          end if
10:     end for
11:     return EKCFI_RET_DENY
12: end function
```

3.4 Compatible with Retpoline

Retpoline[3] is a software mitigation technique implemented in modern operating systems to defend against Spectre V2[12] attacks. These attacks exploit the speculative execution of indirect branch predictions to leak sensitive data. As shown in Figure 6, Retpoline mitigates this by redirecting indirect calls to return instructions. This is achieved by pushing the target address onto the stack and utilizing a return instruction to transfer control. Since indirect calls are eliminated, eKCFI no longer functions as intended. Mitigating Spectre attacks and enforcing control flow integrity are independent kernel features, and they can complement each other.

A straightforward approach is to apply eKCFI-RET, given that indirect calls are replaced with returns, allowing us to instrument only one `nop` in the thunk for all indirect calls. However, this method causes all indirect calls redirected to this thunk to trap to a trampoline, even if the call is not the target call. This scenario effectively results in "*full*" instrumentation, reducing the flexibility to instrument specific indirect call sites and incurring additional cost of traps. As shown in Figure 6, we adopt the approach of instrumenting the trampoline before calling the indirect thunk. In x86_64 systems, the target address of each indirect call is placed in the `r11` register by default. This approach preserves the flexibility to instrument specific indirect call sites and eliminates the need for the `mov` instruction, as the target register is already set to `r11`.

eKCFI-RET: Retbleed[17] is another Spectre variant that bypasses Retpoline by targeting the return instructions. To mitigate this attack, similar to Retpoline, all return sites are redirected to a single return site. As explained above, if we just instrument before that return site which is similar to a "*full*" instrumentation. Hence, eKCFI-RET applied the same approach which instrument the trampoline before calling the return thunk.

4 Evaluation

We evaluated the performance of LLVM-KCFI, eKCFI, and eKCFI-RET on virtual machine (VM) using QEMU. We tested LMBench[15] as microbenchmark performance on the VM. All configurations were tested using the Linux v6.5 kernel on Ubuntu 22.04. The kernels were compiled with standard protection features, including Retpoline. The VM was configured with 4 virtual CPUs and 16GB of RAM. We utilized host passthrough mode on a host system with a 20-core, 12th Gen Intel i7-12700K, 4.90GHz processor.

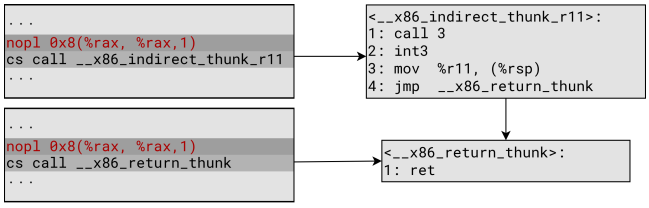

Figure 6: Indirect call sites and return sites are replaced with indirect call thunks and return thunks by Retpoline

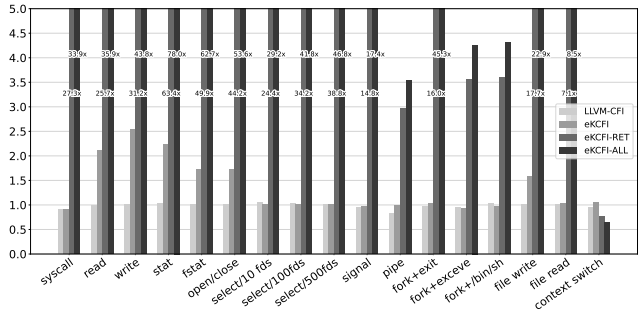

Figure 7: Microbenchmark Performance - KCFI

Our evaluation includes our case study, eKCFI[11], originally published by Jia et al. We built upon their work and implemented new feature, eKCFI-RET, and support with Retpoline. It is important to note that our evaluation results for the eKCFI will differ from theirs, as we did not have access to their specific modifications and configurations of the Linux kernel and LLVM.

Figure 7 presents the performance of applications running on the VM. The results are normalized against those of the same applications running on the VM without KCFI protection. We evaluated LLVM-KCFI and eKCFI with three configurations: (1) eKCFI, (2) eKCFI-RET, and (3) eKCFI-ALL. All configurations are set to "*full*" instrumentation respectively. Configuration (3) incorporates both (1) and (2), providing CFI protection for both indirect call sites and return sites.

The performance evaluation shows that eKCFI presents modest performance overhead on about half of the benchmarks compared to LLVM-KCFI. LLVM-KCFI, in turn, incurs negligible overhead on most of the benchmarks compared to the vanilla kernel without KCFI protection. For eKCFI, benchmarks related to the file system, such as `read/write` and `open/close` operations, incur significant overhead (1.5x-2.5x). This is because Linux uses the Virtual File System (VFS[9]) layer, which introduces extensive indirect calls to various filesystem operations. Each of these indirect calls requires validation by eKCFI, leading to increased overhead in these specific benchmarks.

eKCFI-RET and eKCFI-ALL demonstrate significant overhead across all benchmarks. As previously mentioned, return sites cover all functions, and each function return requires a trap to the trampoline and subsequent validation. We observed that when eKCFI-RET is combined with eKCFI (eKCFI-ALL), the performance overhead multiplies. We believe this is due to increased task rescheduling.

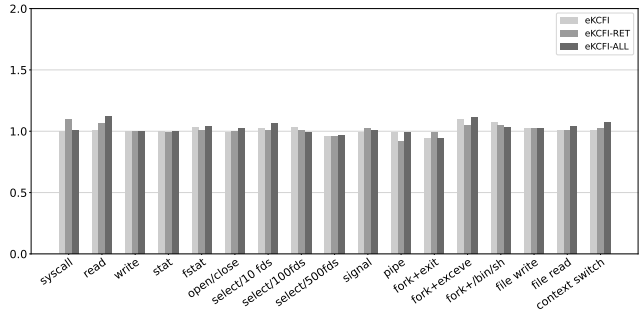

Figure 8: Microbenchmark Performance - NOPs

The frequent traps and validations interfere with task execution times, causing additional delays from preemption, scheduling, and memory allocation. The CFG of the tracing `ls` command, as shown in Figure 9 that placed in Appendix A, demonstrates that the `ls` command incurs extensive nodes, including global functions for memory allocations, scheduling, and other operations.

Figure 8 demonstrates that the NOPs overhead remains consistently low among the configurations. Notably, some benchmarks, such as `pipe` and `exit`, show better performance compared to vanilla. However, since we use QEMU to run the benchmarks, these performance results may not be entirely accurate.

4.1 Security Analysis

Both LLVM-KCFI and eKCFI prevent attackers from hijacking indirect call sites to an arbitrary address. Therefore, even if an attacker injects malicious code, they cannot overwrite function pointers to redirect execution to the malicious code. We evaluated this by implementing a rootkit that overwrites function pointers in the syscall table with a crafted function. Both LLVM-KCFI and eKCFI successfully detected and reported this vulnerability.

eKCFI-RET prevents attackers from hijacking return sites by manipulating return addresses to arbitrary addresses. It mitigates code reuse attacks such as Return-Oriented Programming (ROP). We evaluated this by implementing a backdoor module with an existing out-of-bounds (OOB) vulnerability. An attacker could exploit this OOB vulnerability to overflow the kernel stack and overwrite the return address. In this scenario, LLVM-KCFI failed to detect the vulnerability, while eKCFI-RET successfully detected and reported it.

5 Discussions and Limitations

5.1 Runtime Flexibility and Integration with Hardware CFI Mechanisms

Although eKCFI-RET shows the significant overhead in Figure 7, as previously mentioned, one of our goals is to allow both eKCFI[11] implementations to be effectively attached or detached during runtime. Additionally, "*full*" instrumentation is not typically necessary, and LLVM-KCFI and existing hardware mechanisms for CFI such as Intel CET[6] or Arm PAC[13] can complement eKCFI. For example, a critical use case is when a zero-day vulnerability occurs in a specific function or process. In such cases, a benign root administrator can instantly generate a fine-grained CFG and live-patch eKCFI to the victim function as a temporary mitigation without needing to tear down the system.

5.2 Limitations of eBPF in KCFI Context

While eBPF is a powerful tool in the context of Kernel Control Flow Integrity (KCFI), several limitations and challenges must be addressed. Firstly, our approach requires trusting the eBPF subsystem; in our threat model, if an attacker can corrupt kernel memory, they might also corrupt the memory of helper functions or maps. This highlights the need for additional protection mechanisms. Hardware-based solutions like Intel Memory Protection Keys (MPK)[2] could be useful for safeguarding eBPF maps, but protecting kernel helper functions remains difficult due to their deep integration within the kernel. Similarly, as previously mentioned, we assume a benign administrator, but an attacker might exploit the kernel to escalate their privileges and gain control over the eKCFI module. Another significant limitation involves the Control Flow Graph (CFG) tracing method, which can exhibit randomness and potentially lead to system crashes. Lastly, eBPF programs themselves are not protected, which may expose additional attack surfaces.

6 Conclusion

In this project, we conducted a comprehensive analysis of the functionality of eKCFI and introduced eKCFI-RET to enhance protection for return sites. Despite eKCFI-RET incurring significant overhead due to its comprehensive coverage of all functions, the flexibility of allowing runtime attachment and detachment of protection mechanisms enabled immediate responses to zero-day vulnerabilities through live-patching, without requiring full instrumentation or system teardown. Overall, our enhancements to eKCFI substantially improved kernel security while maintaining an acceptable performance trade-off.

7 Acknowledgments

We extend our deepest gratitude to the researchers whose groundbreaking work laid the groundwork for this project. We would like to thank the contributions of eKCFI[11], presented by Jinghao Jia, Michael V. Le, Salman Ahmed, Dan Williams, Hani Jamjoom, Tianyin Xu.

References

[1] [n. d.]. Providing protection for complex software. https://developer.arm.com/documentation/102433/latest/.
[2] 2015. Memory protection keys. https://lwn.net/Articles/643797/.
[3] 2018. *White paper: Retpoline: A Branch Target Injection Mitigation.* Technical Report Revision 003. Intel. 22 pages. https://www.intel.com/content/dam/develop/external/us/en/documents/retpoline-a-branch-target-injection-mitigation.pdf
[4] 2022. fprobe: Introduce fprobe function entry/exit probe. https://lwn.net/Articles/881019/.
[5] 2023. Linux Plumbers Conference 2023. https://lpc.events/event/17/sessions/155/#20231113.
[6] 2023. *White paper: Complex Shadow-Stack Updates (Intel® Control-Flow Enforcement Technology).* Technical Report. Intel. 14 pages. https://www.intel.com/content/www/us/en/content-details/785687/complex-shadow-stack-updates-intel-control-flow-enforcement-technology.html
[7] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)* 13, 1 (2009), 1–40.
[8] Common Vulnerabilities and Exposures. 2016. CVE - CVE-2016-0728. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-0728.

[9] Richard Gooch. [n. d.]. Linux Virtual File System. https://docs.kernel.org/filesystems/vfs.html.
[10] Brendan Gregg. 2021. BPF Internals. USENIX Association.
[11] Jinghao Jia, Michael V. Le, Salman Ahmed, Dan Williams, and Hani Jamjoom. 2023. Practical and Flexible Kernel CFI Enforcement using eBPF. In *Proceedings of the 1st Workshop on EBPF and Kernel Extensions* (New York, NY, USA) *(eBPF '23)*. Association for Computing Machinery, New York, NY, USA, 84–85. https://doi.org/10.1145/3609021.3609293
[12] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2020. Spectre attacks: Exploiting speculative execution. *Commun. ACM* 63, 7 (2020), 93–101.
[13] 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 *28th USENIX Security Symposium (USENIX Security 19)*. 177–194.
[14] Ananth Mavinakayanahalli, Prasanna Panchamukhi, Jim Keniston, Anil Keshavamurthy, and Masami Hiramatsu. 2006. Probing the guts of kprobes. In *Linux Symposium*, Vol. 6. 5.
[15] Larry W McVoy, Carl Staelin, et al. 1996. Lmbench: Portable tools for performance analysis.. In *USENIX annual technical conference*. San Diego, CA, USA, 279–294.
[16] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing {Forward-Edge}{Control-Flow} integrity in {GCC} & {LLVM}. In *23rd USENIX security symposium (USENIX security 14)*. 941–955.
[17] Johannes Wikner and Kaveh Razavi. 2022. {RETBLEED}: Arbitrary speculative code execution with return instructions. In *31st USENIX Security Symposium (USENIX Security 22)*. 3825–3842.

A Appendix

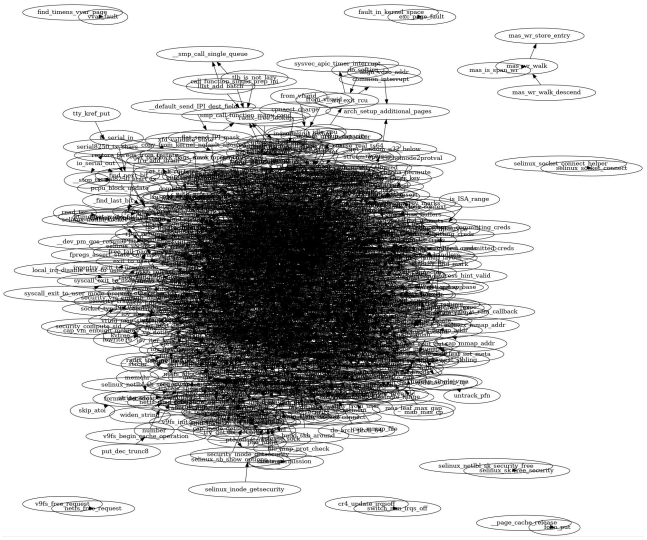

Figure 9: The Control Flow Graph (CFG) of tracing the 1s command via eKCFI-RET approach